Une architecture pour intégrer des composants de contrôle de la coopération dans un atelier distribué

Manuel Munier

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Une architecture pour intégrer des composants de contrôle de la coopération dans un atelier distribué

THÈSE

présentée et soutenue publiquement le 15 janvier 1999

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Manuel Munier

Composition du jury

Président :	M. Claude Godart, Professeur à l'Université Henri Poincaré, Nancy I, ESSTIN
Rapporteurs :	M. Roland Balter, Professeur à l'Université Joseph Fourier, Grenoble M. Claude Chrisment, Professeur à l'Université Paul Sabatier, Toulouse M. Jacques Guyard, Maître de Conférence à l'Université Henri Poincaré, Nancy I
Examinateurs :	M. Jean-Marc Andreoli, Xerox Research Center Europe, Grenoble M. Khalid Benali, Maître de Conférence à l'Université Nancy 2

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

Résumé

Cette thèse présente un nouveau modèle de transactions avancé permettant non seulement la coopération entre activités par le biais d'échanges d'informations en cours d'exécution, mais également leur distribution et l'hétérogénéïté de leurs relations de coopération. Notre objectif est de décentraliser le contrôle de l'application dans les activités la composant, c'est-à-dire:

- chaque activité est responsable de ses interactions avec les autres activités,
- les contrôles réalisés localement assurent, implicitement, la synchronisation globale du système.

Pour cela, nous avons développé deux critères de correction distribués, à savoir la *D*-sérialisabilité et la *DisCOO*-sérialisabilité. Ils assurent des propriétés globales équivalentes à celles des critères de correction classiques (la sérialisabilité et la *COO*-sérialisabilité) mais en ne se basant que sur des contrôles locaux effectués par chacune des transactions du système.

Outre la décentralisation du contrôle des interactions, nous proposons également des mécanismes permettant aux transactions de négocier les règles (schémas de coopération) à respecter lors de leurs échanges.

Mots-clés: coopération, distribution, transaction, schéma de coopération, négociation.

Abstract

In this thesis we detail a new advanced transaction model that not only supports cooperation between activities through intermediate results exchanges while executing, but also allows them to be distributed and autonomous and to use possibly different cooperation schema to share their data. Our main objective was to decentralize the control of interactions towards the activities themselves, ie:

- each activity coordinates its own interactions with other activities,
- these controls performed locally implicitly ensure the synchronization of the whole system.

With that in mind, we defined two distributed corectness criteria: the *D*-serializability and the the *DisCOO*-serializability. They provide the same global properties than classical corectness criteria (serializability and *COO*-serializability) but they only rely on local controls performed by each transaction.

Besides the decentralization of the control of interactions, our second objective was to define mechanisms to let transactions negotiate the rules (cooperation schemas) they want to verify on their data exchanges.

Keywords: cooperation, distribution, transaction, cooperation schema, negotiation.

Remerciements

Avant de vous présenter les travaux réalisés dans le cadre de ma thèse, je voudrais tout d'abord adresser mes plus sincères remerciements aux personnes qui m'ont apporté leur aide et leur soutien pour que ce travail puisse aboutir.

A Claude Godart, mon directeur de thèse, qui m'a accueilli au sein de l'équipe ECOO, encadré et dirigé dans mes recherches tout au long de ces trois années.

A Claude Chrisment et à Roland Balter, mes rapporteurs, qui ont accepté de relire et d'évaluer mon manuscrit et qui ont formulé des remarques constructives.

A Jacques Guyard, qui a bien voulu assumer la fonction de rapporteur interne et qui s'est intéressé à ma thèse.

A Jean-Marc Andreoli, pour l'intérêt qu'il a montré à ce travail et pour ses critiques qui ont permis de l'affiner.

A Khalid Benali, qui m'a encadré et dirigé dans mes recherches, dont les critiques m'ont permis de progresser dans mon travail et ma rédaction, et avec qui j'ai eu grand plaisir à collaborer.

Aux membres de l'équipe, aux thésards et aux stagiaires qui m'ont aidé et conseillé dans mon travail, avec une pensée particulière pour Karim Baina.

A toutes les personnes du Loria qui travaillent dans l'ombre mais qui répondent toujours présentes quand on a besoin d'elles, et particulièrement à Danielle Marchand, à Vireack Ul, à Antoinette Courrier, et à l'équipe des "Moyens Informatiques".

A ma famille, bien évidemment, qui m'a fourni, à chaque instant, un soutien moral sans faille qui m'a permis d'achever ce travail dans les meilleures conditions.

A mes amis, pour leur présence et leur bonne humeur.

"Dans la vie, il n'y a pas de solutions.
Il y a des forces en marche: il faut les créer et les solutions suivent."

(Antoine de SAINT-EXUPERY)

Table des matières

Chapitre 1

Introduction

1.1 Contexte de l'Etude

La démocratisation des moyens de communication tels qu'internet (réseau à l'échelle mondiale, facilité de raccordement, faibles coûts de connexion) ouvre de nouvelles perspectives en informatique, notamment du point de vue du **partage de l'information**. Cependant, le réseau internet est souvent assimilé au web, c'est-à-dire à un réseau où chacun peut publier, de manière non structurée et non organisée, ses propres informations. Les autres internautes sont alors libres d'y accéder et de les utiliser comme ils le désirent. En particulier, ils n'ont aucune obligation de prendre en compte les mises-à-jour apportées à un document auquel ils ont accédé plusieurs mois auparavant et qu'ils désirent utiliser pour produire de nouveaux documents. Dans le cas présent, ces derniers risquent ainsi de faire référence à des informations périmées.

Notre vision du partage de l'information consiste plutôt à utiliser ces moyens de communication pour déployer de nouveaux systèmes d'informations dans lesquels les différents acteurs pourraient travailler de manière coordonnée, éventuellement en même temps et sur les mêmes informations, bien que se trouvant à des endroits géographiquement éloignés. C'est le cas par exemple lorsque plusieurs entreprises coopèrent pour former une **entreprise virtuelle**. Le concept d'entreprise virtuelle désigne le fait que de nombreuses applications distribuées sont le résultat de la **coopération** de plusieurs acteurs remplissant différents rôles. Cette coopération peut être continue, comme par exemple entre un constructeur automobile et ses sous-traitants, ou éphémère comme par exemple les différents corps de métier dans la construction d'un bâtiment. Ces acteurs forment un système relationnel structuré autour d'un objectif commun pour une durée généralement limitée à la durée d'un projet. Par exemple, lors de la construction d'un immeuble, différents partenaires avec des compétences complémentaires (architecte, bureau de contrôle, entreprise de construction, électricien, charpentier, ...) s'associent en une entreprise "virtuelle" pour la durée de construction de l'immeuble. Ce concept d'**entreprise-projet** est très représentatif du domaine des applications coopératives distribuées à large échelle auquel nous nous intéressons.

Au cours d'un tel projet, les différents partenaires alternent des périodes d'isolation avec des périodes d'interaction. Au cours de ces dernière, ils vont être amenés à coopérer

en se communiquant des informations à la fois partielles[1] et intermédiaires[2] pour pouvoir réaliser leur tâche. En outre, dans le cas d'une activité de conception relativement complexe telle que la construction d'un centre commercial, il n'existe pas de partenaire qui possède à lui seul la maîtrise et la connaissance intégrale de l'activité globale exécutée (de la structure porteuse du bâtiment au nombre de prises de téléphone par cellule commerciale). Il est donc extrêmement difficile pour un partenaire d'appréhender "manuellement" toutes les conséquences qu'une modification effectuée par un des partenaires pourrait avoir sur les autres informations partagées.

Ce sont autant de raisons pour lesquelles il est intéressant de profiter de la migration d'une coopération basée sur des outils traditionnels (téléphone, fax, papier, disquettes) vers une coopération électronique via internet pour intégrer des mécanismes de coordination et de communication permettant d'organiser et de contrôler les échanges d'informations entres les différents partenaires.

1.2 Coopération et Applications Distribuées

Dans le cadre d'une telle entreprise virtuelle (ou encore entreprise-projet), les mécanismes mis en œuvre pour supporter la coopération doivent cependant respecter un certain nombre de contraintes vis-à-vis des différentes entreprises associées, en particulier celles liées à leur besoin d'autonomie et à leur répartition géographique (sites éloignés). En effet, les environnements d'aide à la coopération habituellement utilisés pour coordonner un groupe de personnes travaillant sur un réseau dédié ne répondent pas entièrement aux besoins de la coopération dans le cadre des applications distribuées à large échelle:

- Les solutions organisées autour d'une base de données centralisée ne sont pas adaptées. D'une part, ceci limite considérablement l'autonomie des partenaires. D'autre part, il est nécessaire de définir un "administrateur" à qui tous les partenaires du projet confieront la gestion des données qu'ils désirent partager (mais peut-être pas avec tous leurs partenaires). Notre objectif es de distribuer le contrôle des échanges de données entre partenaires tout en assurant implicitement une certaine cohérence globale.
- Il ne doit pas s'agir d'un système propriétaire imposant à tous les partenaires du projet d'utiliser des logiciels identiques pour pouvoir coopérer. Les mécanismes mis en œuvre doivent au contraire pouvoir s'interfacer avec les outils de conception et les autres logiciels déjà en place dans les entreprises. En fait, il s'agit de distinguer, au niveau des utilisateurs, les outils de coordination des outils de production proprement dits.
- Le coût, tant au niveau de l'infrastructure matérielle/logicielle à mettre en œuvre que des changements des habitudes de travail des utilisateurs, doit être minimisé. En effet, les entreprises participant à un tel projet n'ont pas forcément les moyens

1. Ce sont des éléments d'information permettant aux autres partenaires du projet de démarrer leur activité au plus tôt sans devoir attendre que les informations complètes soient publiées.

2. Ces échanges ayant lieu en cours d'activité, les informations partagées sont potentiellement sujettes à de nouvelles modifications.

d'investir dans des réseaux dédiés, soit du fait de leur taille (il s'agit souvent de petites ou moyennes entreprises), soit du fait de la durée limitée de la coopération (le temps du projet).

Parmi les environnements d'aide à la coopération les plus couramment utilisés figurent les gestionnaires de configurations. Leur rôle se limite toutefois à la gestion des versions et configurations successives des différentes données partagées, c'est-à-dire à assurer la cohérence d'un référentiel commun (généralement centralisé) soumis aux accès concurrents des différents partenaires. Aucun contrôle n'est réalisé sur la manière dont ces partenaires s'échangent les données.

Les environnements centrés procédés proposent quant à eux de contrôler la coopération entre les différents partenaires en se basant sur la connaissance des procédés exécutés. Cela nécessite toutefois de connaître, à priori, les différentes actions qui seront accomplies par les partenaires. Or, comme nous l'avons indiqué, déterminer le procédé de l'activité globale peut devenir extrêmement difficile dans le cas d'une activité de conception relativement complexe.

Dans le domaine des bases de données, les systèmes transactionnels garantissent que si chaque activité (encapsulée dans une transaction), prise individuellement, s'exécute correctement, alors l'exécution entremêlée de plusieurs activités (due à leurs accès concurrents aux données partagées) est également "correcte" (par rapport à un critère de correction). Les modèles de transactions habituellement utilisés supposent toutefois que les activités accèdent à un référentiel commun, qu'elles sont isolées les unes des autres (les résultats intermédiaires d'une activité ne sont pas visibles) et qu'elles sont de courte durée (application bancaire par exemple). D'autres modèles de transactions, dits avancés, ont été développés pour relâcher l'isolation entre les transactions (ex: ConTracts). Toutefois, ces modèles ont été définis pour répondre à des besoins bien spécifiques et n'ont le plus souvent pas de base formelle au sens critère de correction des exécutions.

Une dernière catégorie d'environnements, les outils d'aide au travail coopératif (*"Computer Supported Cooperative Work"*) ou collecticiels, est plus orientée vers les aspects "humains" de la coopération tels que la gestion d'un groupe de personnes, les mécanismes de notification, les techniques de communication (messagerie électronique, vidéo-conférences, ...). L'objectif des collecticiels est d'intégrer différents outils pour assister les utilisateurs lorsqu'ils coopèrent. Du point de vue de la coordination, les mécanismes utilisés par les collecticiels sont ceux développés dans d'autres domaines tels que les systèmes distribués ou les gestionnaires de configurations.

1.3 Objectifs de la Thèse

Dans le cas des applications coopératives distribuées, la programmation explicite des interactions entre les différentes activités est généralement difficile à maîtriser puisque cela nécessite de prévoir toutes les interactions possibles. Afin de masquer cette complexité et de décharger au maximum les programmeurs d'applications des problèmes liés aux interactions entre activités concurrentes, le projet *COO* développé au sein de l'équipe ECOO aborde le problème de la coopération selon une approche transactionnelle. Parmi les résultats de ce projet figurent notamment la définition d'un nouveau modèle de transactions, les

COO-transactions, ainsi qu'un nouveau critère de correction, la *COO*-sérialisabilité. Celui-ci relâche la propriété d'isolation entre les transactions, leur permettant ainsi de coopérer en s'échangeant, au cours de leur exécution, des résultats intermédiaires au travers d'un référentiel commun. Le projet *COO* reste toutefois basé sur une architecture centralisée qui n'est pas adaptée aux entreprises virtuelles. D'une part, ceci limite considérablement l'autonomie des partenaires puisqu'ils sont ainsi dépendants d'un site central. D'autre part, il est nécessaire de définir un "administrateur" à qui tous les partenaires du projet confieront la gestion des données qu'ils désirent partager (mais peut-être pas avec tous leurs partenaires). L'approche développée dans cette thèse consiste à distribuer à la fois le stockage des objets partagés et le contrôle des échanges de données entre partenaires, tout en assurant implicitement une certaine cohérence globale.

Nous considérons ainsi une application coopérative distribuée comme étant un ensemble d'activités représentant les différents partenaires du projet. Chacune de ces activités dispose de son propre référentiel local dans lequel elle stocke une copie des objets qu'elle partage avec les autres activités. Deux activités pourront alors coopérer par le biais de tranferts de données entre leurs référentiels locaux respectifs. Dans le cas général, les activités ne sont pas isolées les unes des autres, ce qui signifie que ces échanges pourront éventuellement avoir lieu en cours d'exécution, c'est-à-dire que les activités auront la possibilité de s'échanger des résultats intermédiaires.

En ce qui concerne la coordination de ces échanges, l'idée que nous avons développée est que chaque activité doit être responsable de ses propres échanges de données avec les autres activités (architecture d'égal-à-égal). En d'autres termes, **nous voulons assurer un contrôle global équivalent aux critères de correction classiques (sérialisabilité, *COO*-sérialisabilité) mais en ne nous basant que sur des contrôles locaux (*D*-sérialisabilité, *DisCOO*-sérialisabilité) effectués par chacune des transactions du système.** Pour cela, deux activités désirant partager un certain objet devront tout d'abord négocier un schéma de coopération. A partir de ce moment, à chaque fois qu'un échange de données concernant cet objet devra avoir lieu entre ces deux activités, chacune d'entre elles vérifiera que toutes les règles fixées par le schéma de coopération sont respectées au niveau de son propre référentiel local. Si les deux activités valident cet échange, l'opération de tranfert entre leurs référentiels respectifs est alors exécutée, sinon elle est refusée. Ainsi chaque activité contrôle elle-même ses interactions avec les autres activités et un échange de données ne peut avoir lieu que si les deux activités impliquées dans cet échange donnent leur accord.

Dans les systèmes transactionnels classiques, y compris *COO*, le contrôle des interactions repose sur un seul et unique critère de correction appliqué à toutes les transactions. Cela signifie que l'on doit prévoir, au niveau du référentiel commun, toutes les interactions possibles entre toutes les activités du système, ce qui revient à définir le comportement global du système. Or cette tâche peut rapidement devenir extrêmement complexe, surtout si les transactions n'utilisent pas toutes les mêmes règles de coopération pour leurs échanges de données. Notre second objectif est de tenir compte, directement au niveau du modèle de transactions, de l'hétérogénéïté des relations de coopération entre les différentes transactions. Outre la décentralisation du contrôle des interactions, nous proposons également de permettre aux transactions de négocier les règles (schémas de coopération) à respecter lors de leurs échanges.

Ainsi, les travaux présentés dans cette thèse s'articulent autour de la **coopération** (coordination des échanges de résultats intermédiaires), de la **distribution** (contrôle délocalisé, autonomie des activités) et de la **négociation** (règles de coopération négociées par les activités).

1.4 Organisation du Mémoire

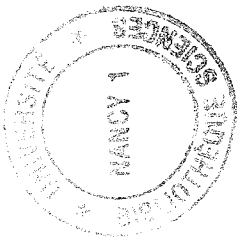
Dans le **chapitre 2** nous étudierons la **problématique** du contrôle de la coopération dans le cadre des applications distribuées. Nous commencerons tout d'abord par présenter les aspects de la coopération auxquels nous nous intéressons, à savoir la coopération indirecte via le partage de documents, ainsi que les problèmes à résoudre dans le contexte des applications visées: coopération par échanges de résultats intermédiaires, distribution du contrôle pour une plus grande autonomie des activités, hétérogénéïté des règles de coopération à respecter lors de ces échanges. La seconde partie de ce chapitre nous permettra ensuite d'**exposer notre approche (transactionnelle)** et de **situer nos travaux et nos objectifs** par rapport à ceux déjà développés dans le cadre du projet *COO*.

Le **chapitre 3** sera quant à lui consacré à l'**étude des solutions existantes** pouvant être utilisées pour coordonner et synchroniser les activités d'une application coopérative distribuée: **gestionnaires de configurations**, **environnements centrés procédés**, **systèmes transactionnels**, **outils d'aide au travail coopératif**. Nous y analyserons les **avantages** et les **inconvénients** de chacune de ces quatre approches par rapport aux contraintes inhérentes au domaine des applications visées: **coopération**, **distribution** et **hétérogénéïté**.

Les **résultats de nos travaux** seront présentés au **chapitre 4**. Nous y développerons **un nouveau modèle de transactions avancé** supportant les exécutions de transactions distribuées géographiquement, coopérant par échanges de résultats intermédiaires et négociant les règles de coopération à respecter lors de ces échanges. Outre la **coopération** au sens du projet *COO*, ce modèle de transactions prend également en compte les aspects liés à la **distribution des objets** (multiples copies d'un même objet logique) et à la **distribution du contrôle de la concurrence** (définition de critères de correction locaux) ainsi que les aspects liés à l'**hétérogénéïté des relations de coopération** entre les différentes transactions (tous les échanges ne sont pas forcément contrôlés par le même critère).

Une **mise en œuvre** de ce modèle sera proposée au **chapitre 5**. Etant donné que nous ne voulions pas définir un système propriétaire supplémentaire, nous avons plutôt choisi de développer un **ensemble de services de coopération de base** à partir desquels il sera possible de programmer de telles applications coopératives distribuées. Un **exemple** provenant du domaine de la construction d'un bâtiment sera également détaillé dans ce chapitre afin d'illustrer à la fois ces différents services de coopération et le modèle de transaction que nous aurons proposé.

Finalement, la **conclusion** présentera quelques **perspectives possibles** des travaux décrits dans ce mémoire.



Chapitre 2

Problématique

Table des matières

2.1 Problématique Générale

L'évolution des performances des réseaux d'ordinateurs combinée à la baisse de leur coût permet aux entreprises de coopérer électroniquement pour former des **entreprises virtuelles**. Cette démocratisation des moyens de communication tels qu'internet ouvre également la voie aux entreprises qui n'ont pas les moyens d'investir dans des réseaux dédiés, soit du fait de leur taille (il s'agit souvent de petites ou moyennes entreprises), soit du fait de la durée limitée de la coopération (durée d'un projet). Aussi, nous pensons que de telles applications coopératives distribuées à large échelle vont devenir de plus en plus fréquentes et notre objectif est de proposer un modèle d'activités permettant de les supporter.

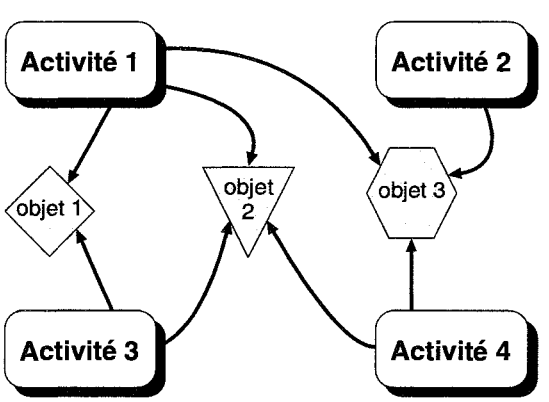
Cependant, comme nous le constaterons (cf. section 2.1.3 et chapitre 3), les environnements d'aide à la coopération existants (gestionnaires de configurations, outils de gestion des procédés, systèmes transactionnels, outils d'aide au travail coopératif) ne sont pas adaptés aux exigences de telles applications. Tout d'abord, la **coopération** ne se limite pas au contrôle de la cohérence des données partagées. Il est également nécessaire de coordonner les différentes activités de l'application (vérification de propriétés sur les

interactions entre ces activités), sans devoir toutefois prévoir explicitement toutes les interactions possibles. Un second problème est lié aux contraintes inhérentes à la **distribution** (panne d'un site, temps de réponse, ...). Les différentes activités d'une telle application peuvent en effet être réparties sur un réseau à large échelle avec des connexions à faible bande passante, voire même occasionnelles dans le cas de liaisons par modem. Il n'est donc pas réaliste qu'une activité soit obligée d'attendre la réponse ou l'autorisation d'un site central pour exécuter une opération et voir les résultats de son action. La moindre perturbation (défaillance ou surcharge) au niveau de ce site central empêcherait les activités de travailler. L'idée est donc de rendre les activités les plus autonomes possible. Un troisième problème lié à la classe d'applications visées est dû à l'**hétérogénéïté** des modes de coopération entre les différentes activités. Au sein d'une entreprise virtuelle par exemple, les règles de coopération entre les différents partenaires ne seront pas forcément les mêmes selon les partenaires.

2.1.1 Coopération Directe vs Coopération Indirecte

Le problème de la coopération entre les différentes activités d'un système distribué peut être abordé de deux manières: coopération directe ou coopération indirecte. La forme la plus "naturelle" d'interaction est la discussion où la coopération prend la forme d'échanges verbaux d'informations et a pour objectif la prise de décision collective. Nous appelerons **coopération directe** cette façon de coopérer. Une autre forme d'échange d'informations, que nous appelerons **coopération indirecte**, est celle qui passe par l'échange de documents, ou plus généralement par l'échange de données. Dans ce cas, il n'y a pas de dialogue explicite entre les deux partenaires; l'information est véhiculée par le document auquel ils accèdent tous les deux (figure 2.1). **C'est sur cet aspect coopération indirecte qu'est orienté notre travail.**

Figure 2.1 Coopération Indirecte



- L'activité 1 coopère avec l'activité 2 via l'objet 3, avec l'activité 3 via les objets 1 et 2, et avec l'activité 4 via les objets 2 et 3.
- L'activité 2 coopère avec les activités 1 et 4 via l'objet 3.
- L'activité 3 coopère avec l'activité 1 via les objets 1 et 2, et avec l'activité 4 via l'objet 2.
- L'activité 4 coopère avec l'activité 1 via les objets 2 et 3, avec l'activité 2 via l'objet 3, et avec l'activité 3 via l'objet 2.

Cette forme de coopération indirecte via les données partagées peut être vue comme un problème de concurrence d'accès puisque deux activités coopèrent en manipulant les mêmes données. Afin de préserver la cohérence de ces données partagées, il est bien évidemment nécessaire de contrôler et de synchroniser les accès effectués par les différentes

activités de manière à ne pas introduire d'inconsistance au niveau des résultats produits par ces activités.

Bien que de nombreux travaux aient été réalisés à ce sujet dans des domaines aussi différents que les systèmes transactionnels dans les bases de données, les gestionnaires de configurations, les techniques de gestion de procédés ou les outils d'aide au travail coopératif (CSCW[3]), ceux-ci ne sont pas toujours adaptés au type d'applications qui nous intéresse, à savoir les applications coopératives distribuées à large échelle. Comme nous le verrons en particulier dans le cas des entreprises virtuelles, ces applications ont en effet des propriétés et des besoins spécifiques vis-à-vis de la coopération.

2.1.2 Coopération dans les Applications Visées

Si les problèmes liés à la synchronisation des accès concurrents sur les données partagées sont bien connus, les solutions habituellement utilisées ne sont pas adaptées en ce qui concerne les applications auxquelles nous nous intéressons du fait de leur nature particulière:

- La forte **interactivité** entre les activités composant l'application: si chaque partenaire a son propre objectif, la réussite de chacun dépend également de la réussite des autres. Cela signifie que ces activités ne vont pas travailler de manière isolée, mais vont au contraire s'échanger, au cours de leur exécution, des résultats intermédiaires de leurs travaux. Ceci permet aux différents partenaires de réagir au plus tôt par rapport au travail effectué par l'un d'eux. Par contre, un tel résultat intermédiaire est potentiellement inconsistant puisque l'activité l'ayant produit n'a pas terminé sa tâche et est donc susceptible de le modifier. Les autres partenaires devront donc tenir compte de ces nouvelles modifications.
  Cette caractéristique est en contradiction avec la propriété d'isolation des activités assurée par les systèmes transactionnels classiques pour garantir l'atomicité à la concurrence.

- Les activités sont par nature **incertaines** puisque le déroulement de l'application dépend essentiellement de décisions humaines. Le "programme" de l'exécution peut ainsi évoluer au fil du temps et n'est donc pas figé au démarrage de l'application. Par conséquent, le contrôle des interactions entre les activités ne doit pas être réalisé à priori (en définissant à l'avance les procédés qui seront exécutés) mais de manière dynamique en fonction des opérations invoquées par les activités. Notre objectif est donc de définir de nouveaux critères de correction des exécutions, tels ceux des systèmes transactionnels, acceptant les exécutions coopératives.

- Les activités sont de **longue durée** par rapport aux unités d'exécution informatiques habituelles: une activité peut durer plusieurs heures, plusieurs jours, plusieurs semaines, voire même plusieurs mois. Si nous prenons l'exemple de la construction d'un bâtiment, la durée de l'activité "réalisation du plan" n'est pas de l'ordre de la milliseconde comme dans le cas d'une application bancaire de gestion de compte! Cet aspect a pour conséquence qu'il n'est pas envisageable de bloquer une activité à

3. Computer Supported Cooperative Work

cause d'un verrou quelconque pendant qu'une autre activité modifie un document. Il n'est pas non plus réaliste de devoir "annuler" une activité ayant réalisé plusieurs mois de travail puis de la redémarrer à cause d'un conflit d'accès sur un document partagé.
Mais la longue durée des activités peut aussi être un avantage. Par exemple en ce qui concerne les temps de transfert, si un décalage de plus de 20 ms n'est pas acceptable entre une image et le son associé, un décalage de plusieurs minutes entre deux plans d'un bâtiment sémantiquement reliés, mais qui nécessitent plusieurs jours d'élaboration, est tout à fait acceptable.

- Les entreprises peuvent éventuellement être **éphémères**: dans le cas d'une entreprise-projet, la durée de vie d'une entreprises peut être limitée à celle du projet. Par conséquent, il ne doit pas être nécessaire de disposer d'une infrastructure informatique "lourde" (lignes spécialisées pour des connexions permanentes à forte bande passante) pour pouvoir déployer de telles applications.
- L'**autonomie** des partenaires: pour les raisons de performance des connexions évoquées précédemment, il ne doit pas être nécessaire aux différents partenaires de devoir être connectés de manière permanente à un quelconque référentiel centralisé pour pouvoir accomplir leur tâche. Chaque partenaire doit donc posséder sa propre version des documents qu'il utilise. En outre, certains échanges de documents peuvent éventuellement être directs, i.e. de partenaire à partenaire, ce qui n'est pas autorisé dans le cas d'une architecture centralisée.
- L'**hétérogénéïté** des relations entre les différents partenaires: que ce soit pour des raisons d'organisation, de gestion des responsabilités ou simplement de confidentialité, les règles de coopération entre les différents partenaires ne sont pas forcément les mêmes.

Comme cela est expliqué dans la section suivante, ces différentes caractéristiques des applications coopératives distribuées (à large échelle) et hétérogènes ne nous permettent donc pas d'utiliser les environnements d'aide à la coopération traditionnels, tels que les gestionnaires de configurations, les outils de gestion des procédés, les systèmes transactionnels classiques ou encore les outils d'aide au travail coopératif. C'est donc guidés par ces exigences que nous avons travaillé au développement d'un nouvel environnement qui puisse supporter de telles applications.

2.1.3 Environnements d'Aide à la Coopération

Dans le cas d'une application relativement complexe, il n'existe généralement pas d'acteur qui possède à lui seul la maîtrise et la connaissance intégrale de l'activité globale exécutée. Il est donc extrêmement difficile pour un acteur quelconque d'appréhender, "manuellement", toutes les conséquences d'une modification apportée à un document de l'application. Ce sont les raisons pour lesquelles il est indispensable d'utiliser des environnements mettant en œuvre des mécanismes de coordination et de communication suffisament sophistiqués, permettant ainsi de notifier et de propager les changements aux

acteurs qui sont concernés et de s'assurer que les efforts de chacun des acteurs du projet sont coordonnés de manière à réduire l'impact d'une modification d'un document sur l'activité globale.

Ces environnements peuvent être classés en quatre catégories selon leur approche de la coopération. Nous avons tout d'abord les **gestionnaires de configurations** dont l'objectif est de gérer les versions et les configurations successives des différentes données partagées (cohérence des données). Viennent ensuite les **environnements centrés procédés** qui permettent de contrôler les états successifs des données partagées et/ou l'enchaînement des différentes activités. Dans le domaine des bases de données, les **systèmes transactionnels** garantissent que l'exécution en parallèle de plusieurs activités (encapsulées dans des transactions) n'introduit pas d'inconsistance au niveau des résultats produits par ces activités. La dernière catégorie, les **outils d'aide au travail coopératif**, est plus orientée vers les aspects communication et relations humaines de la coopération.

Gestionnaires de Configurations

Afin de contrôler les mises-à-jour concurrentes effectuées par les différentes activités d'un système distribué, il est possible d'utiliser un outil de gestion de configurations (RCS [Tic89], ClearCase [Atr94], Continuus, Adèle [Bel94]). Son rôle est d'assurer le stockage des données partagées, appelées ressources, tout en gardant une trace de leur évolution (généralement sous la forme de leurs versions successives) et en contrôlant les accès concurrents effectués par les activités. Par exemple, si deux activités modifient en parallèle une même ressource, elles vont chacune développer, à partir d'une version initiale de cette ressource, ce que l'on appelle une branche de versions. De cette façon, chacune des activités travaille sur sa propre copie de la ressource, sans être pertubée par les modifications effectuées par l'autre activité. Lorsqu'elles auront terminé leur travail, une activité (éventuellement différente) sera chargée de fusionner ces deux branches afin de ne produire qu'une seule nouvelle version, i.e. que les modifications d'une des activités n'écraseront pas les modifications de l'autre. Le rôle du gestionnaire de configurations sera alors de mémoriser le fait que cette nouvelle version est dérivée des deux précédentes.

Toutefois, la plupart des gestionnaires de configurations reposent sur une architecture client/serveur (référentiel centralisé, éventuellement répliqué et/ou partitionné sur plusieurs serveurs). Ils ne conviennent donc pas à nos exigences de distribution et d'autonomie des activités. En outre, les gestionnaires de configurations sont essentiellement concernés par les problèmes de concurrence d'accès à un référentiel: gestion des versions et des configurations de ressources partagées. Ils ne définissent aucun contrôle de la coopération sur les échanges entre activités.

Environnements Centrés Procédés

A la différence des gestionnaires de configurations, les outils de gestion de procédés sont principalement orientés vers la description des exécutions correctes en termes d'états successifs d'une ressource ou d'enchaînement des différentes activités: modèles à flots de tâches ou modèles de workflow [WFM97, Alo96] (modèle des contrats [Wac92]), règles événement/condition/action [Bar92a] (Adèle-Tempo [Bel94], MARVEL [Bar92a, Bar92b]).

Cette approche nécessite généralement de décrire l'application complète, i.e. en tenant compte de toutes les activités et de toutes les ressources. Si l'on intègre en plus les problèmes liés à la synchronisation des activités distribuées, le modèle obtenu devient alors rapidement complexe du fait de la complexité inhérente du contexte à modéliser.

Contrairement aux outils de gestion de procédés existants, notre objectif n'est donc pas de décrire comment les activités doivent travailler pour pouvoir coopérer, mais simplement de définir de quelle façon doivent se dérouler les échanges entre ces activités. Nous voulons donc imposer des contrôles sur les échanges de résultats entre activités, mais pas sur les activités elles-même ni sur la manière dont elles produisent ces résultats.

Systèmes Transactionnels

Dans la cas d'un système transactionnel [Agr90, Ber97], chaque activité est encapsulée dans une transaction dont l'exécution représente la séquence des opérations (lectures et écritures par exemple) invoquées par cette activité sur les objets partagés. Une transaction constitue l'unité d'exécution élementaire: soit la transaction se termine totalement et elle a les effets désirés sur les objets auxquels elle a accédé (la transaction est dite "validée"), soit elle est interrompue et elle n'a aucun effet (la transaction est dite "annulée"). Cette règle du "tout ou rien" (encore appelée propriété d'atomicité des transactions) permet d'assurer l'atomicité aux défaillances des transactions.

L'idée de base d'un système transactionnel est de garantir que si chaque transaction, prise individuellement, s'exécute correctement, alors leur exécution entremêlée (due à leurs accès concurrents aux objets partagés) devra être "correcte". Ceci est assuré par un critère de correction défini comme étant un ensemble de propriétés caractérisant l'histoire des exécutions considérées comme correctes. L'histoire de l'exécution entremêlée de plusieurs transactions est représentée par la séquence des opérations (lecture, écriture, ...) invoquées, concurremment, sur les objets partagés. Le critère de correction le plus répandu dans le domaine des applications traditionnelles (administration, banque, ...) est la "sérialisabilité". Celui-ci considère que l'exécution entremêlée de plusieurs transactions est correcte si elle produit un résultat équivalent à une exécution en série de ces transactions (l'exécution est dite sérialisable).

La sérialisabilité constitue cependant un critère trop strict puisqu'elle garantit l'isolation des transactions (atomicité à la concurrence): les états intermédiaires d'une transaction, en termes de valeurs des objets qu'elle manipule, ne sont pas visibles par d'autres transactions. La sérialisabilité ne supporte donc pas la coopération telle que nous l'avons définie, à savoir la possibilité offerte aux transactions de s'échanger des résultats intermédiaires au cours de leur exécution.

De nouveaux modèles de transactions et critères de correction ont toutefois été définis pour relâcher l'isolation entre les transactions (cf. chapitre 3): transactions emboîtées [Mos81], transactions multiniveaux [Bee88a], sagas [GM87], ...

Outils d'Aide au Travail Coopératif

Le travail de groupe assisté par ordinateur (*"Computer Supported Cooperative Work"*) a pour objectif de permettre à des groupes d'usagers de collaborer à des buts communs au

moyen d'un système informatique, appelé système collaboratif ou collecticiel. Toutefois, contrairement aux gestionnaires de configurations, aux outils de gestion de procédés ou aux systèmes transactionnels, un environnement CSCW n'est pas uniquement orienté vers le maintien de la cohérence des objets partagés (gestion des accès concurrents). Un tel environnement prend également en compte des aspects plus "humains" de la coopération tels que la gestion d'un groupe de personnes, les mécanismes de notification, les techniques de communication (messagerie électronique, vidéo-conférences, ...). BSCW [Ben97a, Ben97b] (partage d'informations au travers d'un référentiel centralisé), Wiki[4] (rédaction collective de documents via le Web) et Microsoft NetMeeting[5] (vidéo/audio conférence, partage d'applications, tableau blanc, forums de discussion synchrones) sont des exemples de tels collecticiels.

En ce qui concerne le contrôle de la cohérence, les collecticiels reposent sur les mécanismes développés dans les systèmes distribués ou les gestionnaires de configurations: verrouillage des objets, passage de jeton (ou "prise de tour"), détection des dépendances (conflits résolus par les utilisateurs). A la différence des systèmes transactionnels, ces techniques ont pour objectif d'assurer la cohérence des objets partagés et non de coordonner les activités qui coopèrent.

2.2 Coopération dans l'Environnement *COO*

Les travaux réalisés dans le cadre de cette thèse s'inscrivent dans le contexte du projet *COO* [Can98c, Mol96, Can96, God96] qui aborde le problème de la coopération selon une approche transactionnelle. Parmi les résultats de ce projet figurent notamment la définition d'un nouveau modèle de transactions, les *COO*-transactions, ainsi qu'un nouveau critère de correction, la *COO*-sérialisabilité [Mol96]. Ceux-ci permettent à un ensemble de transactions de s'exécuter de manière coopérative en relâchant la propriété d'isolation (les propriétés d'atomicité, de consistance et de durabilité étant bien entendu préservées). Cela signifie que les différentes transactions peuvent coopérer en s'échangeant, au cours de leur exécution, des données par l'intermédiaire d'un référentiel commun (architecture centralisée). Or, dans le domaine des applications qui nous intéressent cette centralisation est un obstacle à l'autonomie des activités. Il serait en effet bien plus adéquat que chaque activité du système possède sa propre base de données locale et coordonne elle-même ses échanges avec les autres activités

2.2.1 Approche Transactionnelle

Dans le cas des applications coopératives distribuées et hétérogènes, la programmation explicite des interactions est généralement difficile à maîtriser puisqu'elle nécessite de prévoir toutes les interactions possibles (problème combinatoire). En outre, cette approche "programmation concurrente" est également source d'erreurs (interactions non prévues ou incompatibles, verrous mortels,...). A l'inverse, une approche "contrôle de la concurrence

4. Wiki: `http://wiki.lri.fr:8080/scoop/scoop.wiki`
5. NetMeeting: `http://www.microsoft.com/netmeeting/features/`

d'accès" a justement pour objectif de masquer cette complexité et de décharger au maximum les programmeurs d'applications des problèmes liés aux interactions entre activités concurrentes: un protocole de contrôle de la concurrence assure que le fonctionnement global d'un ensemble d'activités concurrentes est correct, à supposer bien entendu que chaque activité soit individuellement correcte. Cette approche est d'autant plus intéressante dans le cas des applications coopératives visées que les différents partenaires d'une entreprise virtuelle n'ont généralement pas une grosse infrastructure informatique et sont souvent peu conscients des problèmes inhérents au parallélisme des exécutions induit par la coopération. Ce sont là les principales raisons qui nous ont conduit à choisir d'aborder le problème de la coopération dans les applications coopératives distribuées et hétérogènes sous l'angle des **systèmes transactionnels coopératifs**.

Une application coopérative est donc vue comme un ensemble de transactions qui accèdent "en même temps" à un ensemble d'objets. Chaque transaction encapsule une activité (supposée correcte) de l'application afin de lui cacher les problèmes liés à la concurrence d'accès. Une transaction peut ainsi être représentée par la séquence des opérations invoquées sur les objets manipulés. Mais le problème de la synchronisation de ces transactions est plus complexe que celui de la synchronisation des transactions dans des contextes applicatifs traditionnels (administration, banque, ...) [Gra93] où les activités sont essentiellement concurrentes, i.e. pouvant s'exécuter de manière isolée en s'ignorant complètement l'une de l'autre. Dans notre cas, il serait donc plus exact de parler d'une approche "contrôle de la coopération" que d'une approche "contrôle de la concurrence". En fait, l'extension du concept de transaction au domaine des applications coopératives distribuées nécessite donc de résoudre certains problèmes dus aux caractéristiques de ces applications dont nous avons parlé précédemment:

- La nature **coopérative** remet en cause le principe d'isolation généralement utilisé pour assurer l'atomicité à la concurrence. En effet, nous avons défini la coopération comme étant la possibilité offerte aux transactions de s'échanger des résultats intermédiaires **durant** leur exécution. Cela signifie également qu'une transaction, bien que devant être individuellement correcte (i.e. elle doit, au final, produire des résultats consistants), peut diffuser des résultats intermédiaires potentiellement inconsistants au cours de son exécution.

- La **longue durée** des transactions remet en cause la technique du "tout ou rien" généralement utilisée pour assurer l'atomicité aux défaillances. Il n'est en effet pas acceptable de perdre tout le travail réalisé pendant plusieurs heures, voire même plusieurs mois, pour des raisons liées au contrôle des interactions avec d'autres transactions.

  Supposons que deux transactions développent un cycle de dépendances suite à leurs interactions. Dans un système transactionnel classique, une solution est d'annuler une des deux transactions (voire les deux) puis de la relancer ultérieurement. Si cette solution est acceptable dans le cas de transactions de courte durée (opérations de débit ou de crédit d'un compte bancaire), ce n'est pas le cas quand il s'agit d'annuler une transaction représentant plusieurs mois de travail (développement d'un logiciel). Le contrôle des interactions doit donc être assez souple pour éviter de devoir annuler

une transaction.

- L'**hétérogénéïté** des relations entre les différents partenaires d'une entreprise virtuelle rend très difficile la définition d'un critère de correction unique permettant de caractériser toutes les exécutions correctes. Et la nature incertaine de ces transactions (i.e. le "programme" n'est pas connu à l'avance) complique encore le problème. Par conséquent, des parties différentes d'une entreprise, voire même d'une activité, pourront suivre des modèles de coopération différents. Il faudra alors être capable d'intégrer ces différents modèles au sein d'une même application.

Bien que les systèmes transactionnels classiques et les techniques qu'ils mettent en œuvre (isolation, "tout ou rien", ...) ne soient pas adaptés pour supporter la coopération telle que nous l'avons définie (échanges de résultats intermédiaires en cours d'exécution), fonder notre approche de "contrôle de la coopération" sur le concept de transaction n'est toutefois pas contradictoire. Nous pensons en effet que le principe de base des systèmes transactionnels (à savoir simplifier la programmation des applications en cachant au maximum la complexité introduite par l'exécution concurrente des différentes activités) est encore plus important dans le contexte des applications coopératives distribuées (à large échelle) et hétérogènes. Notre travail consiste donc à **définir un nouveau modèle de transactions avancé** permettant de supporter de telles applications, à la manière des modèles de transactions avancés présentés dans [Elm92a, Jaj97].

2.2.2 Point de Départ: Le Protocole *COO* Centralisé

Dans *COO* nous distinguons en fait deux types de résultats: les résultats **intermédiaires** produits par une transaction en cours d'exécution (ces résultats préliminaires sont potentiellement inconsistants et sujets à de nouvelle modifications), et les résultats **finaux** produits par une transaction à la fin de son exécution. Le fait que les transactions ne soient plus isolées les unes des autres durant leur exécution nous procure deux avantages incontestables dans le cas des applications coopératives qui nous intéressent:

- Si une transaction t_1 nécessite l'accès aux résultats produits par une autre transaction t_2 pour accomplir sa tâche, elle peut alors démarrer "au plus tôt" dès que la transaction t_2 aura publié ses premiers résultats intermédiaires. Dans le cadre des modèles de transactions classiques où les transactions sont isolées, la transaction t_1 aurait été obligée d'attendre que la transaction t_2 soit terminée pour pouvoir accéder à ses résultats, ce qui n'est pas acceptable dans le cas de transactions pouvant durer plusieurs mois.
- Deux (ou plusieurs) transactions peuvent s'échanger des résultats intermédiaires de différents objets au cours de leur exécution afin de "s'entraider" pour l'accomplissement de leurs tâches respectives. Ce type d'interactions est bien à la base de la coopération.

Permettre aux transactions de communiquer au cours de leur exécution correspond à la manière dont nous coopérons "naturellement": travailler ensemble. D'autres solutions

consistent à décomposer, parfois "artificiellement" [6], une transaction en une série de sous-transactions représentant ses différentes étapes successives, les résultats étant dans ce cas publiés à la fin de chaque étape (cf. modèle des sagas [GM87], techniques de fractionnement). Une telle approche serait toutefois difficile à mettre en œuvre dans notre contexte du fait de la nature incertaine des transactions considérées: si le programme exécuté par une transaction n'est pas connu à l'avance, comment décomposer cette transaction?

Correction des Interactions

Bien entendu, "ouvrir" ainsi les transactions n'est pas sans poser de problèmes. En effet, que se passe-t-il si une transaction t_1 accède à des résultats intermédiaires produits par une transaction t_2, puis si cette transaction t_2 est annulée? Si deux transactions accèdent simultanément à un même objet, laquelle des deux aura le mot de la fin? Ceci constitue autant de cas pouvant remettre en cause la consistance des transactions si aucune règle de synchronisation n'est définie. Le modèle des *COO*-transactions repose donc sur un critère de correction syntaxique, la *COO*-sérialisabilité [Mol96], qui peut être vu comme une extension de la sérialisabilité classique pour supporter la notion de résultat intermédiaire. Intuitivement, une exécution coopérative sera considérée comme étant correcte si elle respecte les règles de synchronisation suivantes:

1. Si une transaction produit un résultat intermédiaire, alors elle doit produire le résultat final correspondant lorsqu'elle est validée.
2. Si une transaction t_2 accède à un résultat intemédiaire d'une transaction t_1, alors t_2 doit lire le résultat final correspondant (produit quand t_1 est validée) avant de pouvoir produire ses propres résultats finaux [7]. Techniquement, lorsqu'une transaction accède à un résultat intemédiaire d'une autre transaction, elle devient dépendante de cette transaction. Lorsqu'elle lit le résultat final correspondant, cette dépendance est levée.
3. En cas de cycle dans le graphe des dépendances entre transactions (échanges bidirectionnels entre deux transactions par exemple), les transactions impliquées dans le cycle sont groupées. Les transactions d'un même groupe ont alors l'obligation de terminer leur exécution en même temps de manière indivisible.

Comme cela est expliqué dans [Can98a], la *COO*-sérialisabilité nous permet de supporter trois modes de coopération différents entre les transactions (activités) d'une application:

- mode **"client/serveur"**: dans ce cas de figure, une transaction appelée "serveur" produit différents résultats intermédiaires successifs d'un objet. Une autre transac-

6. Nous entendons par "artificiellement" le fait que certaines sous-transactions obtenues ne correspondront par forcément à des (sous-)tâches réelles de l'application.

7. A noter que si la transaction t_1 dure plusieurs mois, la validation de t_2 sera effectivement retardée de plusieurs mois. Cela ne signifie toutefois pas que la transaction t_2 est "bloquée" puisqu'elle pourra éventuellement continuer à travailler pendant ce laps de temps. En outre, si une transaction t_3 a besoin des résultats de t_2, elle pourra démarrer dès que t_2 aura produit ses premiers résultats intermédiaires. Lorsque t_1 sera validée, t_2 pourra à son tour produire ses résultats finaux qui seront ensuite pris en compte par t_3. Il n'y a donc pas de blocage des transactions mais seulement un report de leur validation.

tion, appelée "client", lit (certains de) ces résultats intermédiaires, ce qui lui permet de commencer à travailler sans avoir à attendre que le serveur ait produit le résultat final. Bien entendu, du fait de la règle de synchronisation $n°2$, le client devra tenir compte de ce résultat final (que le serveur est obligé de produire, cf. règle $n°1$) avant de produire ses propres résultats finaux. Ce type d'interactions est représenté sur la figure 2.2.

- mode **"écriture coopérative"**: il s'agit de deux (ou plusieurs) transactions s'échangeant les différents résultats intermédiaires successifs d'un même objet (figure 2.3). Comme cela est expliqué par la règle de synchronisation $n°2$, ces transactions vont donc devenir inter-dépendantes. La règle $n°3$ va donc obliger ces différentes transactions à atteindre un consensus sur le résultat final de cet objet.
- mode **"rédacteur/relecteur"**: ce mode de coopération est similaire au mode "écriture coopérative", excepté le fait que les interactions ont lieu sur des objets différents. Les transactions sont également inter-dépendantes et doivent donc se terminer en même temps.

Si le modèle des *COO*-transactions ainsi que la *COO*-sérialisabilité permettent effectivement de briser l'isolation entre les différentes transactions, il ne s'agit toutefois que d'une première étape en ce qui concerne les applications coopératives distribuées et hétérogènes. En effet, le projet *COO* a été focalisé sur l'aspect **coopération** et, à l'image des systèmes transactionnels classiques, suppose que les transactions interagissent via l'utilisation d'un référentiel commun. Ainsi l'aspect **distribution** (exécution d'activités réparties sur un réseau à large échelle) n'a pas été résolu puisque toutes les données et tous les contrôles sont centralisés sur un serveur unique. De plus, ces interactions sont toutes contrôlées par un même critère de correction: la *COO*-sérialisabilité (mode "écriture coopérative"). Il n'est pas possible par exemple de décider, à priori, que deux activités vont coopérer selon un mode "rédacteur/relecteur" ou "client/serveur" (aspect **hétérogénéïté** non résolu).

Correction des Résultats

Le contrôle syntaxique décrit ci-dessus (voir [Mol96] pour plus de détails) garantit donc la correction des interactions coopératives, à supposer que les transactions, prises individuellement, s'exécutent correctement (propriété de consistance des transactions). Or, dans notre contexte, le "programme" des transactions n'est pas connu à l'avance puisque ce sont les utilisateurs qui décident, interactivement, des opérations à exécuter. Par conséquent, la correction syntaxique des interactions coopératives proposée dans *COO* n'est pas suffisante pour assurer la correction des résultats produits par les transactions. Par exemple, dans le cas du mode de coopération "rédacteur/relecteur", la transaction "relecteur" sera bien obligée de relire la version finale du document à relire produite par la transaction "rédacteur" avant de pouvoir être validée (correction syntaxique), mais nous n'avons aucune garantie que la dernière version de la revue produite par la transaction "relecteur" aura bien pris en compte cette dernière version du document à relire.

Pour garantir la correction individuelle des transactions au sein de l'environnement *COO*, [Ska97] propose de définir un ensemble de contraintes sur le référentiel commun et de vérifier que les résultats de chaque *COO*-transaction repectent bien ces contraintes.

Figure 2.2 Coopération "Client/Serveur"

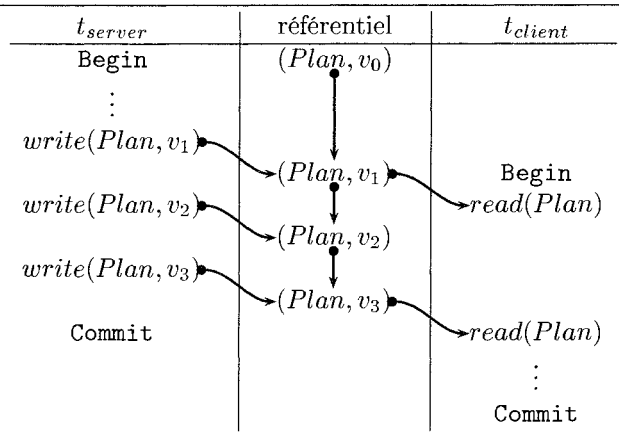


Figure 2.3 Coopération "Ecriture Coopérative"

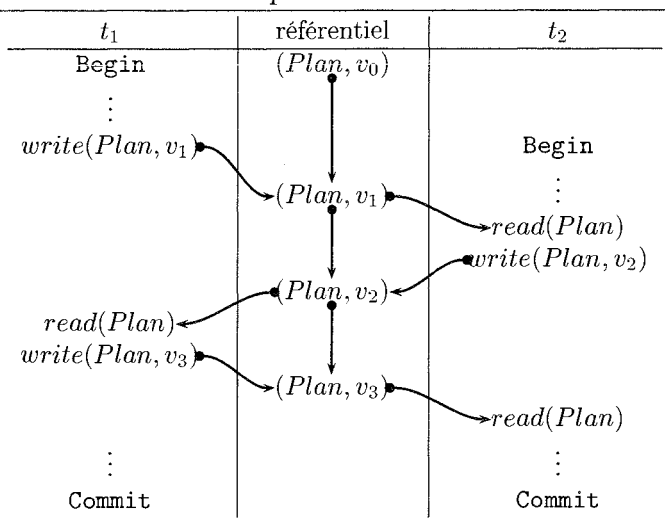
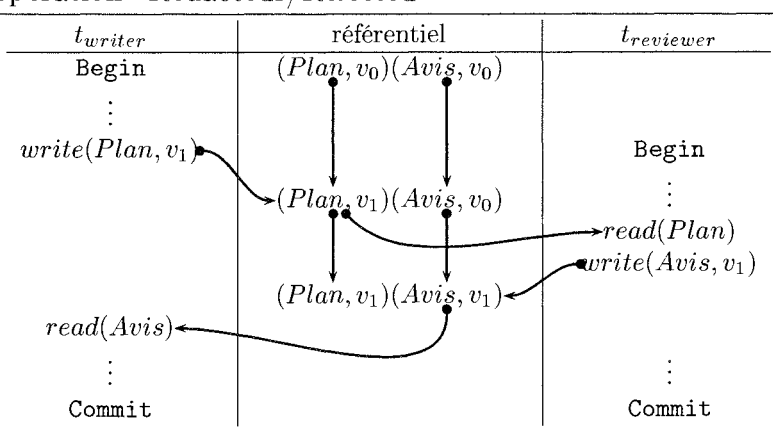


Figure 2.4 Coopération "Rédacteur/Relecteur"



Celles-ci permettent d'exprimer la cohérence des objets contenus dans le référentiel et d'empêcher l'occurence d'un objet qui ne respecterait pas l'une de ces contraintes.

2.2.3 Objectif de la Thèse: Protocoles de Coopération Distribués

Partant des travaux réalisés dans *COO* en ce qui concerne la correction syntaxique des interactions coopératives, l'objectif de cette thèse est donc de proposer un nouveau modèle de transactions avancé permettant une distribution physique des transactions (utilisation de référentiels locaux) ainsi que la définition de nouveaux critères de correction pour coordonner les échanges de données entre ces transactions. Puisque nous envisageons l'utilisation simultanée de plusieurs critères différents pour le partage des différents objets, il nous faudra également être capable d'intégrer ces différents critères au sein d'une même application.

Motivations

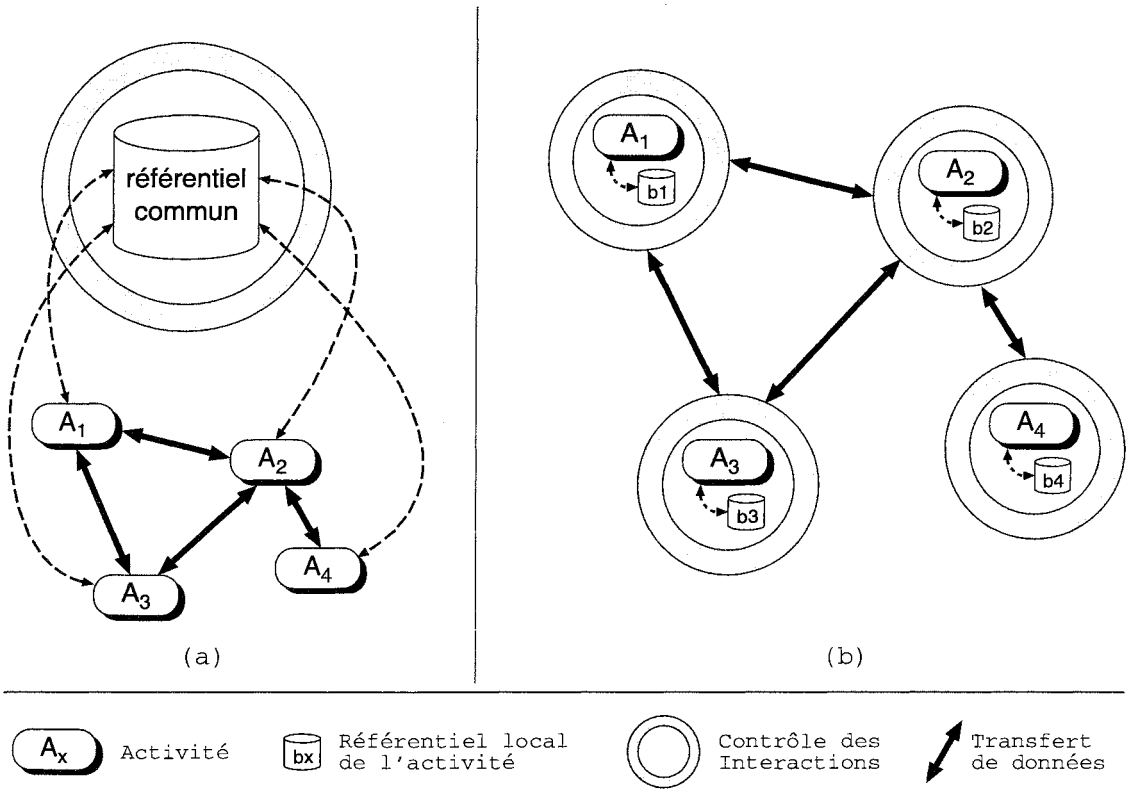
Notre principal objectif concerne donc l'aspect **distribution**. En effet, la plupart des systèmes transactionnels, y compris *COO*, sont fondés sur une base de données commune à toutes les transactions et dans laquelle sont stockées toutes les données partagées. Bien que cette base puisse ensuite être répartie, partitionnée ou répliquée, l'architecture logique du système reste centralisée, c'est-à-dire qu'il s'agit d'une architecture de type client/serveur. Une couche "contrôle des interactions" enveloppe cette base de données et permet de contrôler les accès concurrents effectués par les différentes transactions du système (figure 2.5-a). Or, dans le domaine des applications qui nous intéressent cette centralisation est un obstacle à l'autonomie des activités. Il serait en effet bien plus adéquat que chaque activité du système possède sa propre base de données locale et coordonne elle-même ses échanges avec les autres activités (figure 2.5-b).

Toutefois, en procédant ainsi, nous passons d'un critère de correction centralisé et basé sur l'histoire globale du système à un critère de correction distribué et basé uniquement sur les histoires locales des différentes transactions du système. En effet, la centralisation nous permettait d'avoir une vue globale du système. En revanche, dans une architecture d'égal-à-égal (*"peer-to-peer architecture"*)[8] les transactions n'ont qu'une vue locale et partielle du système (généralement limitée aux transactions avec lesquelles elles coopèrent et à leurs interactions). Il nous faut donc formaliser, au niveau du modèle de transactions lui-même, ces différentes notions (bases de données locales, opérations de transfert entre transactions) puis étudier l'impact de la distribution sur des critères de correction connus, tels que la sérialisabilité ou la *COO*-sérialisabilité.

Un second inconvénient de *COO* et des systèmes transactionnels classiques, également lié à la centralisation du contrôle des interactions, est la nécessité de définir le comportement global du système, i.e. de définir un seul et unique critère de correction que l'on appliquera à toutes les transactions. Cela signifie que l'on doit prévoir, au niveau du "serveur", toutes les interactions possibles entre toutes les transactions du système, et

8. Contrairement à une architecture Client/Serveur, dans une architecture d'égal-à-égal (*peer-to-peer*) chaque site joue à la fois le rôle de serveur et de client, i.e. il n'y a pas de site central (serveur) qui contrôle les autres (clients).

Figure 2.5 Contrôle Centralisé vs Contrôle Distribué



ce pour toutes les données gérées par ce serveur. Cette tâche peut rapidement devenir extrêmement complexe, surtout si les transactions n'utilisent pas toutes les mêmes règles de coopération pour leurs échanges de données. Dans un environnement hétérogène en termes de critères de correction, le fait d'avoir un contrôle local sur chaque transaction plutôt qu'un contrôle centralisé mono-bloc facilite donc l'étude des interactions entre ces différents critères de correction.

Notre travail s'articule donc autour de trois mots-clés: **"coopération"**, **"distribution"** et **"hétérogénéïté"**. La première étape consiste à distribuer l'exécution d'un ensemble de transactions coopératives. Il s'agit tout d'abord de remplacer la base de données centrale par de petites bases locales "attachées" aux transactions, puis de reformuler certains critères de correction (la sérialisabilité et la *COO*-sérialisabilité) afin d'obtenir des critères eux-mêmes distribués, i.e. des critères pouvant être vérifiés localement par chaque transaction mais assurant les mêmes propriétés au niveau global que les critères de correction "centralisés". La seconde étape de notre travail est plus orientée vers l'aspect hétérogénéïté des modes de coopération au sein d'une même application avec pour objectif d'offrir aux transactions la possibilité d'utiliser des règles de coopération différentes selon leurs partenaires et/ou les données partagées. Ceci nous apparaît essentiel, notamment dans les cas des entreprises virtuelles. Lorsque deux partenaires veulent s'associer pour partager certaines données, ils désirent pouvoir fixer eux-mêmes les règles d'échange à respecter et leurs rôles respectifs dans cette association. Toutes ces associations ne seront donc pas forcément basées sur les mêmes règles de coopération.

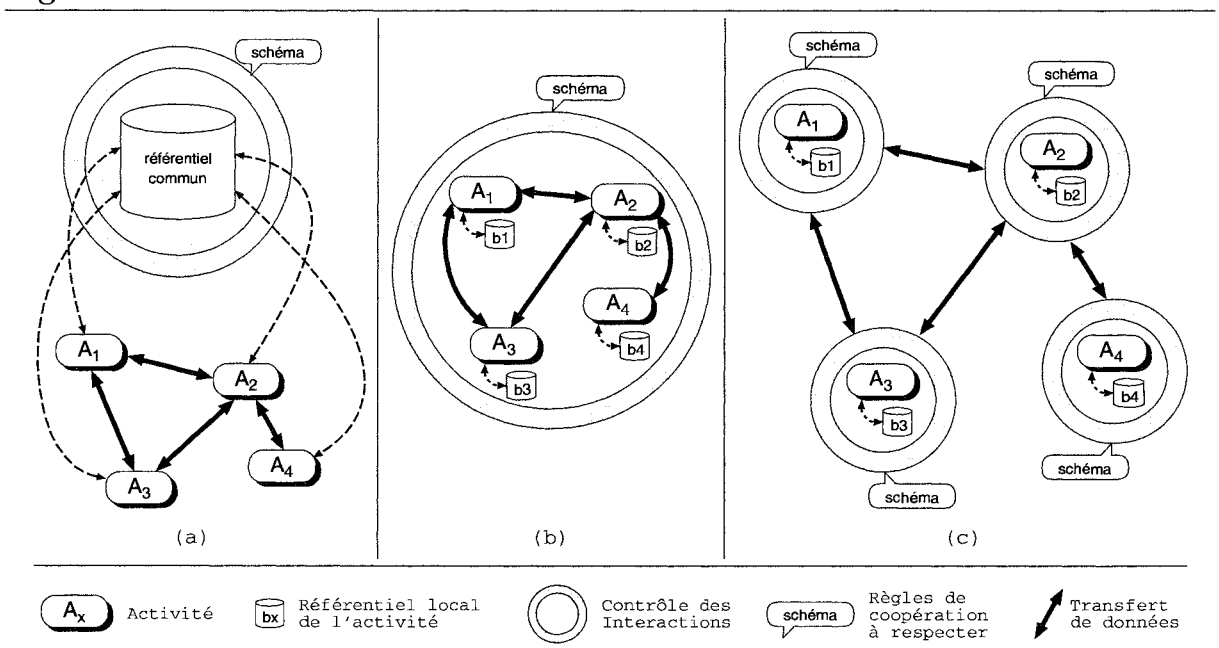
Distribution de l'Exécution

Comme nous l'avons déjà expliqué, nous abordons la coopération comme étant un problème de partage de données soumis au contrôle d'un critère de correction. La distribution de l'exécution des transactions signifie donc deux choses:

- distribuer l'accès aux données
- distribuer le contrôle des interactions entre les transactions

Les modèles de transactions classiques sont destinés à coordonner les accès concurrents effectués par les différentes transactions sur les objets d'un référentiel commun. Les échanges de données entre les transactions se font ainsi de manière implicite au niveau de ce référentiel (figure 2.6-a). Le fait de remplacer ce référentiel commun par des bases locales pour que chaque transaction puisse avoir sa propre copie des objets auxquels elle accède (figure 2.6-b) nous confronte à un premier problème: celui de la gestion de copies multiples pour un même objet logique. Outre l'identification de ces différentes copies (on ne parle plus de l'objet *"plan"* mais de l'objet *"plan de l'activité architecte"*), il est également nécessaire d'assurer la synchronisation de ces différentes copies. Cela signifie donc que les interactions entre les transactions ne sont plus implicites via des accès concurrents à une même copie d'un objet (cas des modèles de transactions classiques avec un référentiel commun), mais **explicites via des échanges entre ces transactions de valeurs de leurs copies, i.e. des transferts de données entre les bases locales.**

Figure 2.6 Distribution de l'Exécution des Transactions



Comme nous l'avons déjà expliqué précédemment, il ne suffit pas de distribuer l'accès aux données pour que les transactions coopératives soient autonomes. La coopération n'est pas en effet synonyme de "liberté totale". Les modèles de transactions reposent donc sur un critère de correction pour contrôler les interactions entre les différentes transactions du

modèle. Toutefois, s'il n'est pas surprenant que dans un modèle classique (du point de vue de la distribution) ce contrôle soit centralisé sur le référentiel qui est lui-même centralisé (figure 2.6-a), il serait absurde d'aboutir à la situation décrite sur la figure 2.6-b où les transactions, bien qu'ayant chacune leur propre copie des objets partagés, soient obligées de demander l'autorisation d'y accéder à un quelconque serveur chargé du contrôle centralisé des interactions. Puisque les interactions sont explicites, cela signifie intuitivement que notre objectif est de rendre chaque transaction responsable de ses propres échanges de données avec les autres transactions (figure 2.6-c). **Nous voulons donc également distribuer le contrôle des interactions.**

Partant de critères de correction existants tels que la sérialisabilité ou la *COO*-sérialisabilité, notre objectif est donc de définir pour chacun de ces critères un ensemble de règles de coopération pouvant être vérifiées localement par chaque transaction (i.e. en utilisant uniquement des informations connues de cette transaction) mais assurant implicitement au niveau global les mêmes propriétés que celles garanties par le critère "centralisé". Intuitivement, que le contrôle des interactions entre les différentes transactions soit effectué de façon centralisée ou qu'il soit délocalisé sur chaque transaction, nous voulons que le comportement du système soit identique.

La première phase de notre travail est donc constituée de deux étapes. Il s'agit tout d'abord de définir un modèle de transactions fondé non plus sur un référentiel commun à toutes la transactions mais sur la notion de base de données locale attachée à chaque transaction. La seconde étape consiste ensuite à reformuler deux critère de corrections connus (la sérialisabilité et la *COO*-sérialisabilité) de manière à obtenir une version distribuée de ces critères (respectivement la *D*-sérialisabilité et la *DisCOO*-sérialisabilité).

Diversité des Schémas de Coopération

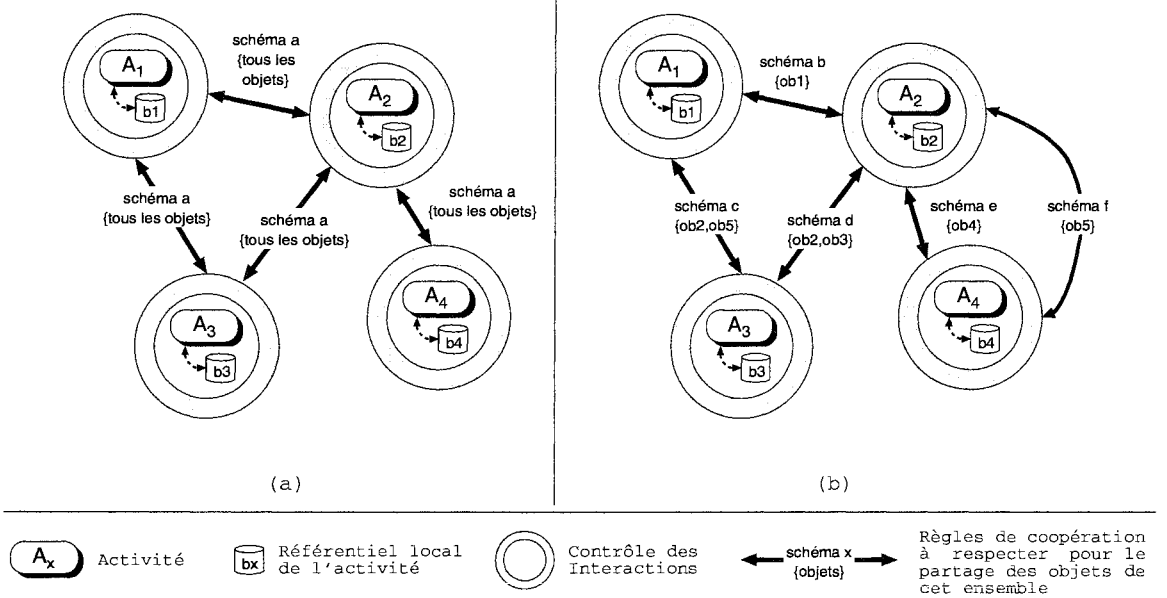
La seconde phase de notre travail est, quant à elle, consacrée à l'aspect **"hétérogénéïté"** des relations entre les différentes activités d'une application coopérative distribuée. En effet, les systèmes transactionnels classiques ne reposent que sur un seul et unique critère de correction pour contrôler toutes les interactions entre toutes les transactions (figure 2.7-a). Par conséquent, cela impose que:

- soit toutes les transactions coopèrent effectivement en suivant strictement les mêmes règles de contrôle,
- soit elles peuvent utiliser des règles différentes, mais le critère de correction est capable de gérer cette diversité (ce qui le rend plus complexe).

Le premier cas de figure est trop restrictif, surtout en ce qui concerne les entreprises virtuelles dans lesquelles les partenaires ont souvent des rôles et des responsabilités différents les uns par rapport aux autres. Le second nécessite de prévoir toutes les interactions possibles, ce qui peut rapidement devenir très difficile et source d'erreurs.

Nous préférons donc développer une approche dans laquelle les interactions entre les transactions sont contrôlées par des règles de coopération locales et pouvant être différentes selon le contexte (figure 2.7-b). Bien entendu, deux partenaires doivent s'accorder sur les règles à respecter lors de leurs échanges. Nous parlons alors de **négociation d'un schéma de coopération entre deux transactions pour le partage d'un ensemble d'objets**

Figure 2.7 Diversité des Schémas de Coopération



donné. Chaque transaction est ainsi responsable de ses propres échanges de données avec les autres transactions. Par conséquent, ceci nous permet de prendre en compte, au niveau du contrôle des échanges de données entre les transactions, l'aspect "hétérogénéïté" des relations de coopération (rôles, règles de coordination, ...) entre les différentes activités d'une application coopérative distribuée.

Toutefois, si cette approche nous permet effectivement de contrôler plus facilement les interactions entre deux transactions du fait de l'utilisation de critères de correction plus simples, nous devons également être capable d'intégrer ces différents schémas de coopération. En effet, comme cela est représenté sur la figure 2.7-b, la couche "contrôle des interactions" d'une transaction donnée devra tenir compte des différents schémas de coopération négociés par cette transaction, ce qui peut provoquer des **interactions** entre ces schémas de coopération.

Intuitivement, de telles interactions peuvent avoir lieu lorsqu'une transaction partage, avec différentes transactions, un même objet en utilisant des schémas de coopération différents. C'est le cas par exemple de la transaction A_3 et de l'objet ob_2 sur la figure 2.7-b. Il est tout à fait possible que certaines opérations d'échange concernant cet objet soient correctes vis-à-vis d'un des schémas de coopération négociés mais incorrectes vis-à-vis d'un autre. Le problème est donc de savoir de quelle manière cette transaction va coordonner ses échanges avec les autres transactions de sorte que les schémas de coopération qu'elle aura négociés pour le partage de cet objet ne soient pas violés.

Il existe également d'autres formes plus "sournoises" d'interactions entre les schémas de coopération. En effet, certaines opérations peuvent ne pas violer explicitement l'un ou l'autre des schémas lorsqu'elles sont invoquées mais conduire, indirectement, à des situations d'interblocage. C'est le cas par exemple lorsque deux opérations, invoquées sur des objets éventuellement distincts, induisent au niveau de schémas de coopération

différents des dépendances contradictoires entre les transactions. Un tel conflit est non seulement difficile à prévoir, mais peut également n'apparaître que tardivement, c'est-à-dire bien après que les opérations conflictuelles n'aient été invoquées.

Outre la définition, au sein de notre modèle de transactions avancé, de la notion de négociation d'un schéma de coopération entre deux transactions pour le partage d'un ensemble d'objets donné, nous devons donc également détailler dans cette partie de notre travail de quelle manière sont prises en compte les interactions entre ces différents schémas de coopération.

2.3 Synthèse

Nous avons donc choisi d'aborder le problème de la coopération dans le contexte des applications distribuées du point de vue **coopération indirecte**, c'est-à-dire via des échanges de données effectués par les différentes activités d'une telle application au cours de leur exécution. En outre, ces échanges peuvent éventuellement porter sur des valeurs dites "intermédiaires" des objets partagés.

Afin de simplifier la programmation de telles applications en cachant au maximum la complexité introduite par l'exécution concurrente des différentes activités, nous avons choisi une **approche transactionnelle** plutôt qu'une approche basée sur les gestionnaires de configurations, les environnements centrés procédés ou les outils de CSCW. Le contrôle de la concurrence est ainsi réalisé par un critère de correction qui définit les exécutions (concurrentes) considérées comme étant correctes.

Toutefois, la notion de coopération est en totale contradiction avec la propriété d'isolation (atomicité à la concurrence) imposée par la plupart des systèmes transactionnels. Il nous faut donc développer un nouveau modèle de transactions et de nouveaux critères de correction, i.e. définir ce qu'est une exécution coopérative correcte et quelles sont les propriétés attendues d'une telle exécution. De ce point de vue, les travaux présentés dans cette thèse sont basés sur le modèle des *COO*-transactions et la *COO*-sérialisabilité définis dans [Mol96], lesquels permettent de briser l'isolation entre les différentes transactions.

En ce qui concerne les applications coopératives distribuées (à large échelle) et hétérogènes il ne s'agit toutefois que d'une première étape. En effet, le système *COO* était focalisé sur l'aspect **coopération** et, à l'image des systèmes transactionnels classiques, suppose que les transactions interagissent via l'utilisation d'un référentiel commun (aspect **distribution** non résolu) et que ces interactions sont toutes contrôlées par un unique critère de correction (aspect **hétérogénéïté** non résolu). L'objectif de cette thèse est donc de définir un nouveau modèle de transactions avancé et de nouveaux critères de correction permettant de prendre en compte les aspects liés à:

- la **distribution des objets** partagés: Chaque transaction doit avoir sa propre copie des objets qu'elle manipule. Cela nécessite donc de pouvoir, au niveau du modèle de transactions, identifier et synchroniser les différentes copies d'un même objet logique. Les transactions coopèrent donc par des échanges de données explicites entre leurs référentiels locaux.
- la **distribution du contrôle de la concurrence**: Afin de rendre les transactions

les plus autonomes possible, la coordination de leurs échanges doit pouvoir être réalisée localement. Cela signifie que chaque transaction est responsable de la consistance des données contenues dans son référentiel local. Notre objectif est donc de définir de nouveaux critères de correction locaux qui, lorsqu'ils sont assurés au niveau de chaque transaction, garantissent les mêmes propriétés au niveau du système complet que leurs homologues "globaux" classiques.

- l'**hétérogénéïté des relations de coopération** entre les différentes transactions: Dans le cadre des applications coopératives distribuées, toutes les activités ne sont pas forcément guidées par les mêmes règles de coopération. Notre objectif est donc de définir la notion de négociation d'un schéma de coopération entre deux transactions pour le partage d'un ensemble d'objets donné. Comme nous l'avons mentionné, cette diversité des schémas de coopération au sein d'un même système peut produire des interactions entre ces schémas.

Chapitre 3

Etat de l'Art

Table des matières

Les problèmes liés à la coordination et à la synchronisation des activités d'une application coopérative distribuée ont déjà fait l'objet de nombreux travaux et résultats dans différents domaines de recherche, chacun ayant ses propres objectifs. Nous pouvons distinguer quatre grandes approches:

- Les **gestionnaires de configurations** dont l'objectif est de gérér les versions et les configurations successives des différentes données partagées.
- Les **environnements centrés procédés** qui permettent de décrire les états successifs des données partagées et/ou l'enchaînement des différentes activités.

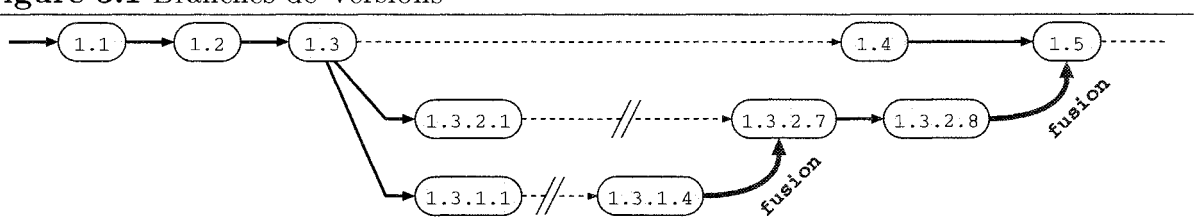
- Les **systèmes transactionnels** dont l'objectif est de garantir que l'exécution en parallèle de plusieurs activités (encapsulées dans des transactions) n'introduit pas d'inconsistance (par rapport à un critère de correction) au niveau des résultats produits par ces activités.
- Les **outils d'aide au travail coopératif**, plus orientés vers les aspects communication et relations humaines de la coopération.

Nous allons donc analyser ces différentes approches et présenter de quelle manière elles répondent à nos trois principaux objectifs: **coopération** (interactions entre les activités), **distribution** (autonomie des activités, contrôle décentralisé) et **hétérogénéïté** (utilisation de règles de contrôle différentes adaptées aux relations entre les activités). Toutefois, comme nous pourrons le constater, aucune de ces approches n'aborde simultanément ces trois aspects dans leur intégralité.

3.1 Gestionnaires de Configurations

Afin de contrôler les mises-à-jour concurrentes effectuées par les différentes activités d'un système distribué, il est possible d'utiliser un outil de gestion de configurations. Son rôle est d'assurer le stockage des données partagées, appelées **ressources**, tout en gardant une trace de leur évolution (généralement sous la forme de leurs **versions successives**) et en assurant leur cohérence. Par exemple, si deux activités modifient en parallèle une même ressource, elles vont chacune développer, à partir d'une version initiale de cette ressource, ce que l'on appelle une **branche de versions**. De cette façon, chacune des activités travaille sur sa propre copie de la ressource, sans être pertubée par les modifications effectuées par l'autre activité. Lorsqu'elles auront terminé leur travail, le gestionnaire de configurations s'assurera toutefois que les deux branches seront fusionnées afin de ne produire qu'une seule nouvelle version, i.e. que les modifications d'une des activités n'écraseront pas les modifications de l'autre par exemple.

Figure 3.1 Branches de Versions



3.1.1 Modèle "Checkin/Checkout"

De nombreux gestionnaires de configurations reposent sur le modèle "checkin/checkout" décrit dans [Tic89]. Celui-ci est construit autour d'un référentiel commun à toutes les activités avec des espaces de travail associés à chaque activité. Le référentiel stocke un ensemble de ressources multi-versionnées. Les versions sont organisées selon un graphe direct acyclique qui comporte toujours une branche principale et des

branches annexes. Chaque activité possède un espace de travail qui est un sous-ensemble mono-versionné du référentiel. Cet espace de travail représente la vue, au sens base de données, qu'a l'activité du référentiel. L'activité dispose alors d'opérateurs de transfert pour synchroniser son espace de travail et le référentiel :

- **checkout** : à partir de l'identificateur d'une ressource et d'un identificateur de version, l'opération `Check_Out` crée une copie de cette ressource dans l'espace de travail de l'activité.
- **ckeckin** : à partir de l'identificateur d'une ressource dans l'espace de travail de l'activité, l'opération `Check_In` crée une nouvelle version de la ressource dans le référentiel.

Les accès concurrents sont gérés en utilisant des verrous en lecture/écriture. Si une ressource est extraite du référentiel en écriture, alors l'activité est assurée de pouvoir créer la prochaine version de cette ressource dans la même branche. Ce mode de fonctionnement permet :

- **le développement en parallèle** : En effet, il est possible de modifier simultanément une même ressource de deux manières différentes dans deux branches séparées. Une branche peut être ultérieurement fusionnée avec une autre branche ou être abandonnée.
- **la gestion des mises-à-jour conflictuelles** : Supposons qu'une ressource soit en cours de modification. La durée de cette modification peut être longue et la ressource est verrouillée au niveau du référentiel. Une autre activité, désirant effectuer rapidement un changement mineur, peut accéder à cette ressource, la modifier et créer une nouvelle version dans une autre branche. Cette nouvelle version pourra être fusionnée à la version principale ultérieurement.

Il faut toutefois être conscient que la fusion de deux branches peut devenir une tâche extrêmement complexe. En effet, s'il existe des outils pour fusionner différentes versions d'un fichier texte (source d'un programme par exemple), fusionner deux versions d'un plan issu d'Autocad ou d'un document Word est autrement plus difficile. En fait, cette fusion peut être vue comme une activité d'intégration des différentes versions d'une ressource dont l'issue est incertaine. Elle peut être anodine ou impossible, voire même remettre en cause le travail effectué par les différentes activités.

Les ressources du référentiel peuvent ensuite être regroupées en **configurations**. Une telle configuration est un sous-ensemble nommé, cohérent et mono-versionné du référentiel. Les opérateurs de transfert entre le référentiel et les espaces de travail peuvent alors également s'appliquer à une configuration. Dans ce cas, les utilisateurs transfèrent dans leurs espaces de travail respectifs non plus une seule ressource mais une configuration.

La plupart des gestionnaires de configurations restent cependant dédiés à la gestion des ressources partagées sur un réseau local. Bien que le référentiel (base de données) puisse être réparti, partitionné ou répliqué, l'architecture logique du système reste centralisée (architecture de type client/serveur). En outre, les espaces de travail sont généralement construits comme étant des vues de cette base de données et sont donc stockés sur ce même serveur. Bien qu'une vue puisse être exportée sous la forme de fichiers (via NFS

par exemple), les différentes activités du système restent donc généralement dépendantes du serveur.

3.1.2 Exemples d'Environnements

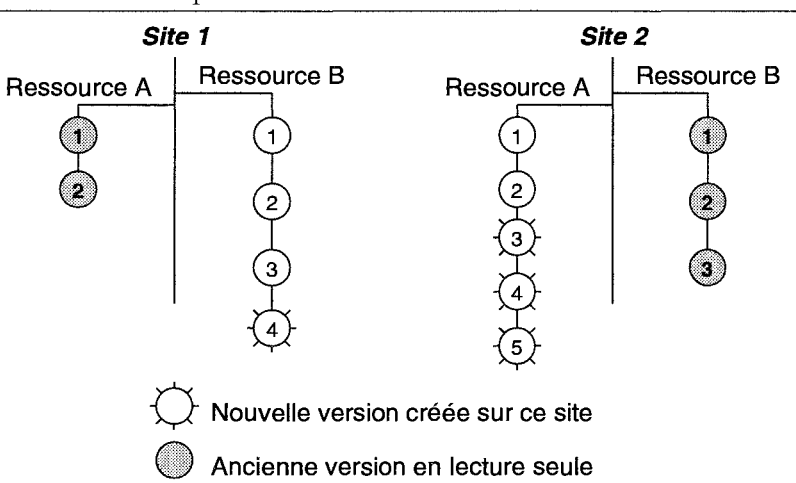
ClearCase

ClearCase [Atr94] est à la base un gestionnaire de configurations centralisé. Via l'extension MultiSite [All95], il permet dorénavant la réplication des ressources sur différents sites. Il utilise une architecture d'égal-à-égal entre les sites qu'il gère afin de garantir leur autonomie. Il y a ensuite des mécanismes de synchronisation des différents sites. Toutefois, pour un site donné, les activités demeurent dépendantes du serveur. En outre, l'espace de travail d'une activité n'existe pas à proprement parler, i.e. il s'agit d'une vue du référentiel construite dynamiquement et présentée sous la forme d'un système de fichiers "virtuel".

En ce qui concerne la distribution, ClearCase MultiSite utilise la notion de ressource maître/ressource répliquée (*Object Mastership*) qui permet d'éviter les conflits de mises-à-jour concourantes entre les différents sites. Cela signifie que pour une ressource donnée, un seul de ses réplicas (ou copies) peut être modifié: c'est la ressource maître. Ces modifications sont ensuite propagées vers ses réplicas. Ce droit de modification peut ensuite être transféré d'un site à un autre.

ClearCase MultiSite se base donc sur la consistance faible des réplicas. Cette approche nécessite le partitionnement des ressources en plusieurs ensembles disjoints (un par site). Ce partitionnement n'est cependant pas sans conséquence sur les possibilités de coopération entre les activités réparties sur différents sites, au risque de transferts de droits de modification intempestifs. L'utilisation des branches dans ClearCase peut être vue comme un moyen naturel pour partitionner les ressources. En effet, chaque branche n'est modifiée qu'à un seul site, les autres branches évoluant lors d'opérations de synchronisation des différents sites.

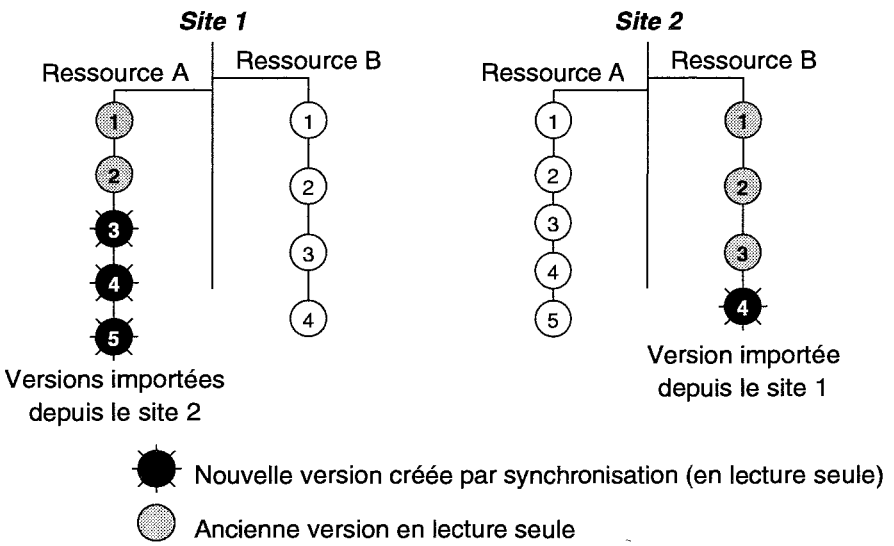
Figure 3.2 Evolution des Réplicas dans ClearCase



La séparation des modifications sur différentes branches rend la synchronisation complètement automatique. Puisque seulement un site peut étendre une branche particulière

(Figure 3.2), toutes les modifications sur cette branche peuvent ensuite être greffées dans les branches correspondantes sur les autres sites (Figure 3.3). Une fois cette synchronisation effectuée, les modifications indépendantes faites sur une même ressource peuvent être fusionnées en utilisant des outils de fusion standard de ClearCase.

Figure 3.3 Synchronisation des Réplicas dans ClearCase



En tant que tel, ClearCase répond donc strictement aux objectifs d'un gestionnaire de versions et de configurations. Il n'est pas possible, par exemple, de définir des règles d'échange entre les activités, si ce n'est via la notion de partitionnement des ressources. Du point de vue de la coopération, pour aller au-delà de la simple concurrence d'accès, il sera donc nécessaire de faire appel à des outils de gestion des procédés tels que ClearGuide (cf. section 3.2).

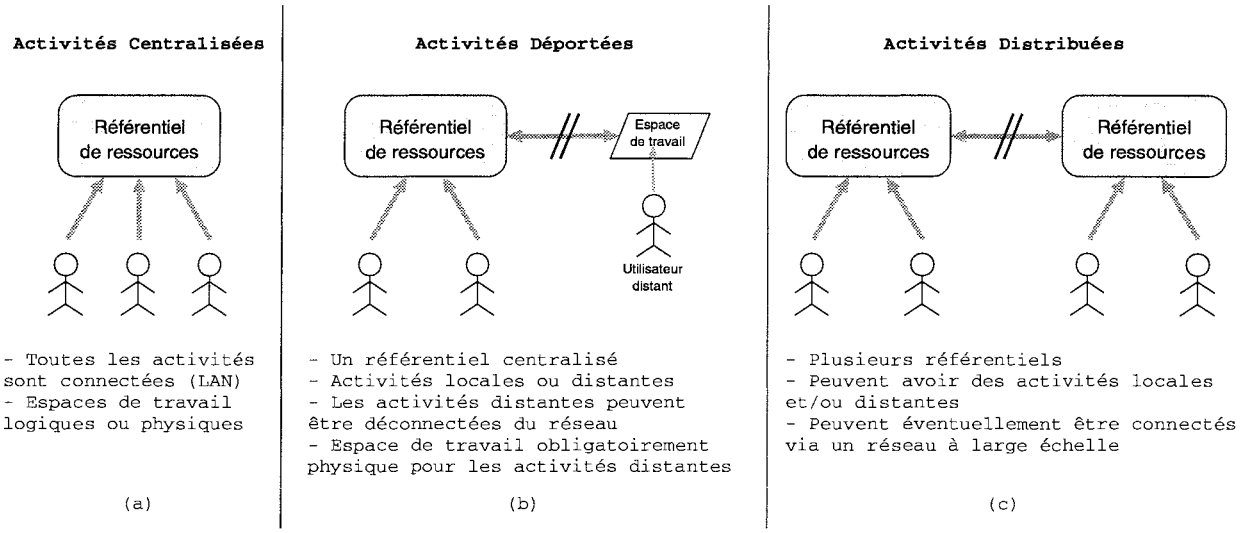
Continuus

Le référentiel de ce gestionnaire de configurations peut être structuré de trois façons différentes selon les besoins des activités et/ou des acteurs:

- **centralisé** (Figure 3.4-a) : Toutes les activités partagent le même référentiel. Les espaces de travail (ou projections), s'ils sont logiques, sont stockés sur le serveur.
- **déporté** (Figure 3.4-b) : A partir d'un référentiel (centralisé), une activité crée (physiquement) un espace de travail "déporté" sur la machine où elle s'exécute. Elle peut ainsi travailler même si elle est déconnectée du réseau. Elle doit ensuite passer par une phase de "réconciliation" pour intégrer ses modifications au référentiel.
  Il faut noter que seul l'espace de travail est déporté vers l'activité, i.e. uniquement les versions en cours de modification des ressources. Si l'activité désire consulter l'historique des versions de ses ressources, elle doit se reconnecter au référentiel.
- **distribué** (Figure 3.4-c) : Il s'agit en fait de plusieurs référentiels qui doivent être synchronisés. Il faut préciser que si une ressource est présente (répliquée) sur plu-

sieurs sites, ses différentes copies peuvent éventuellement être modifiées simultanément (par opposition au "droit de modification" de ClearCase MultiSite).

Figure 3.4 Organisation des Référentiels dans Continuus



Avec le gestionnaire de configurations Continuus il est donc possible de faire en sorte que chaque activité puisse disposer d'un espace local pour stocker les ressources qu'elle utilise (espace de travail dans le cas d'une activité déportée, référentiel "local" dans le cas d'une activité distribuée). Les activités sont ainsi autonomes du point de vue de l'accès aux données. Le contrôle des interactions entre les différentes activités, quant à lui, reste toutefois centralisé sur le(s) référentiel(s). Il n'est pas possible, par exemple, de contrôler les échanges de données que deux activités déportées pourraient avoir directement entre leurs espaces de travail respectifs.

Tout comme ClearCase, Continuus n'aborde le problème de la coopération entre les différentes activités que du point de vue de la concurrence d'accès aux données partagées. Pour enrichir la sémantique des échanges de données entre les activités il sera donc nécessaire d'utiliser également un environnement de développement centré procédés.

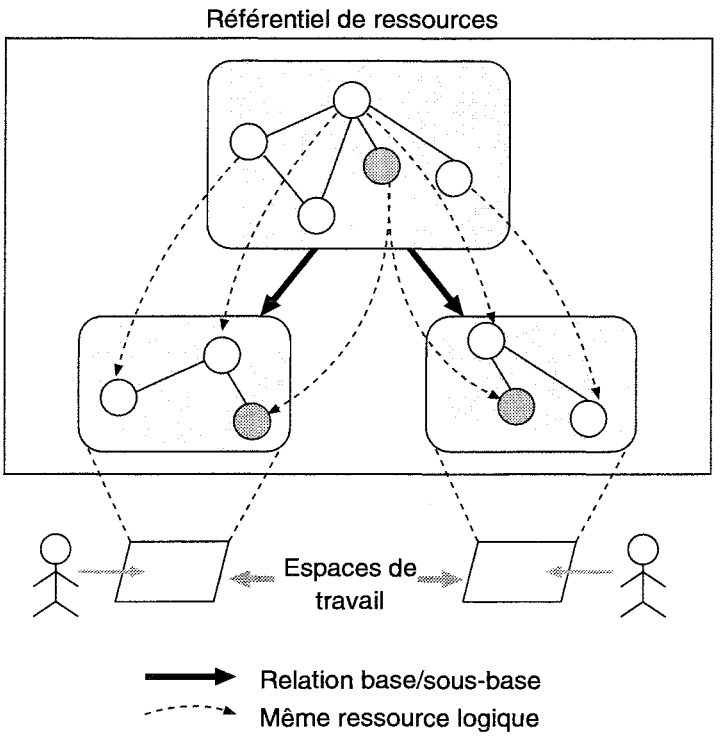
Adèle

Dans Adèle [Bel94], les activités existent explicitement et s'organisent de manière hiérarchique. A chaque activité est associée une sous-base du référentiel (modèle base/sous-base). Adèle fait en effet la distinction entre espace de travail et sous-base (Figure 3.5). Un espace de travail est la représentation de la sous-base sur un système de gestion de fichiers. L'objectif est de permettre à un utilisateur d'appeler ses outils UNIX habituels sur les fichiers présents dans son espace de travail.

Lorsqu'une ressource sera partagée entre deux activités, cette ressource sera donc présente dans les deux sous-bases de ces activités. Pour synchroniser ces deux copies de la même ressource, Adèle supporte deux politiques de partage:

- Les deux sous-bases se partagent **physiquement** la même ressource. Dans ce cas,

Figure 3.5 Référentiel & Espaces de Travail dans Adèle



si cette ressource est modifiée dans une base, la modification est immédiatement visible dans l'autre.

- Chaque sous-base possède **sa propre copie** de la ressource. Dans ce cas il est possible de modifier la même ressource en même temps dans deux bases différentes. Cette ressource a donc une valeur différente par base. Il faut remarquer que la ressource originale n'est pas à priori verrouillée. Sa valeur peut être modifiée d'une façon alors que sa copie l'est d'une autre. Pour prévenir toute perte de mise-à-jour, la copie doit être resynchronisée avec l'originale avant d'être transférée de l'activité fille vers l'activité mère.

  Pour décrire les règles à respecter pour le partage des ressources, Adèle a été étendu de manière à former un véritable environnement de développement centré procédés. Adèle utilise pour cela un langage à base de règles de synchronisation: Tempo.

Il faut cependant noter que le référentiel (organisé selon un modèle bases/sous-bases) est géré de façon centralisée (i.e. sur un seul serveur). Il en est de même des espaces de travail qui ne sont que des vues des sous-bases sous la forme d'un système de gestion de fichiers. Toutes les activités sont donc dépendantes du serveur.

3.1.3 Conclusion

Concernant les applications coopératives distribuées (à large échelle) et hétérogènes, les gestionnaires de configurations ne prennent en compte que les aspects liés à la concurrence

d'accès aux données partagées. En particulier, ils ne répondent pas entièrement à nos besoins de:

- **distribution**: La plupart des gestionnaires de configurations reposent sur une architecture centralisée. Bien que certains, tels que ClearCase ou Continuus, offrent la possibilité d'avoir plusieurs référentiels, la gestion d'un site donné (contrôle des interactions) demeure centralisée.
- **autonomie**: Les espaces de travail des activités sont le plus souvent stockés sur le serveur, ce qui oblige les activités à y être connectées en permanence. Continuus permet toutefois de créer des espaces de travail déportés. Une activité pourra ainsi travailler tout en étant déconnectée du réseau puisqu'elle aura une copie de ses ressources stockée localement. Pour créer une nouvelle version de ces ressources, ou bien pour consulter l'historique des versions précédentes, elle devra cependant se reconnecter au référentiel. Elle n'est donc pas entièrement autonome par rapport à ce référentiel.
- **coopération**: Les gestionnaires de configurations sont essentiellement concernés par les problèmes de concurrence d'accès à un référentiel, i.e. par la gestion des versions et des configurations des ressources partagées. Adèle (couplé avec Tempo et Apel) mis à part, ils ne définissent aucun contrôle de la coopération sur les échanges entre activités. Cela signifie que chaque activité peut faire ce qu'elle désire, du moment que le gestionnaire de configurations réussit à construire son graphe de versions! Cet aspect de la coopération est généralement confié à une couche "gestion des procédés" ajoutée au-dessus du gestionnaire de configurations.

Que ce soit ClearCase (avec ClearGuide, cf. section 3.2), Continuus (système de workflow) ou Adèle (avec Tempo puis Apel), de nombreux gestionnaires de configurations intègrent dorénavant des outils de gestion de procédés. Cela leur permet ainsi de coordonner les différentes activités d'une application en se basant cette fois sur la connaissance des procédés exécutés par ces activités.

3.2 Environnements Centrés Procédés

Nous pouvons distinguer deux grandes approches pour décrire les procédes: l'approche basée sur les flots de tâches (*"workflow"*) et celle utilisant des règles événement/condition/action (règles ECA ou *"triggers"*). La première approche est orientée vers la définition de l'enchaînement des différentes activités et la conduite de projet (délais à respecter, responsabilités, ...). Elle nécessite toutefois de connaître à priori les différentes actions qui seront exécutées par les activités. L'objectif de la seconde approche est plutôt de définir de quelle manière doivent réagir les activités aux différents événements. Il s'agit donc plus d'une approche "propagation de modifications". Les approches à base de réseaux de Pétri (cas de Process Weaver par exemple) sont quant à elles à mi-chemin entre ces deux approches.

3.2.1 Modèles à Flots de Tâches

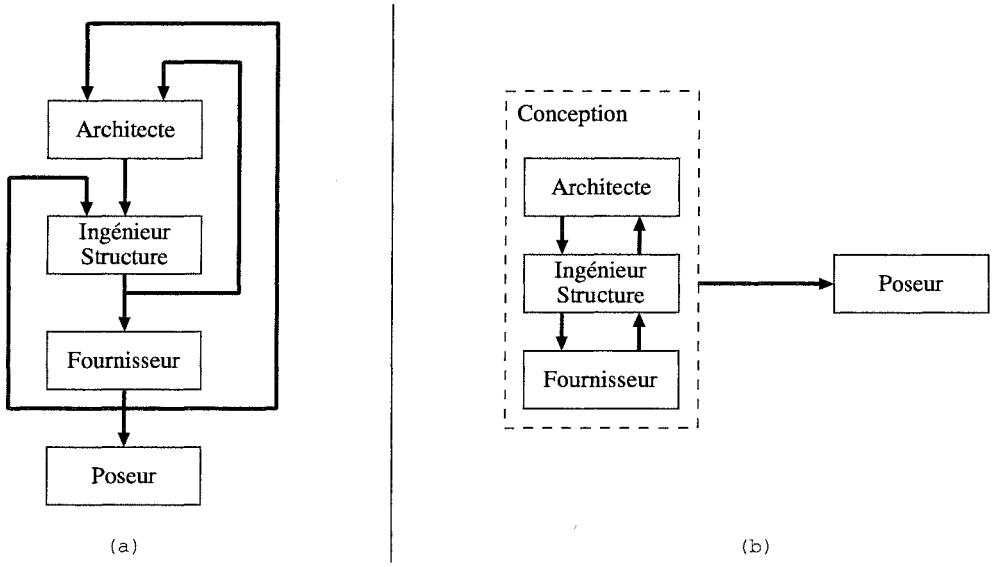
Les modèles à flots de tâches (ou modèles de workflow) ont été conçus pour faciliter le développement d'activités de longue durée, coordonnant des tâches exécutées sur des systèmes existants et potentiellement hétérogènes. Le principe d'un modèle de workflow consiste à construire et à contrôler un scénario d'exécution mettant en œuvre des traitements existants, en préservant l'autonomie des sources de données sur lesquelles ils ont été développés. Ces traitements sont vus comme des briques de base (*"step"*) et sont intégrés dans un graphe d'enchaînement de tâches (*"script"*). Un modèle de workflow est avant tout un modèle de programmation incluant la persistance, la cohérence, la reprise, la synchronisation et la coopération. Plus précisément, un modèle de workflow vise les objectifs suivants:

- Définir un modèle de programmation permettant de fédérer des traitements existants et distribués. Ce point revient à contrôler et à synchroniser un flot d'exécution entre des activités élémentaires indépendantes.
- Définir un modèle de reprise permettant de redémarrer un flot d'exécution après une défaillance, sans annulation des traitements déjà effectués.
- Définir un modèle d'exécution autorisant l'arrêt volontaire d'un flot d'exécution et sa reprise après un délai quelconque, potentiellement sur une machine différente. Le flot d'exécution doit donc avoir une forme partielle de durabilité.
- Spécifier, au sein même du flot d'exécution, les actions à exécuter en cas de conflit (alternative à l'attente) ou en cas d'échec (alternative à l'abandon). En particulier, afin d'éviter les abandons en cascade, il est préférable de baser les procédures d'abandon sur des actions de compensation.

L'objectif d'un modèle de workflow est donc d'assurer la cohérence sémantique des applications. Toutefois, dans le cadre des applications visées, cette façon de procéder n'est pas suffisante pour décrire la nature des interactions entre les activités. Prenons l'exemple d'une application visant à construire un bâtiment. Cette application peut être divisée en quatre activités. La première représente l'architecte qui a pour tâche de fournir les plans du bâtiment correspondant aux désirs du client. A partir de ces plans et de son expérience personnelle, l'ingénieur structure va, dans une seconde activité, effectuer les principaux choix techniques pour la réalisation du bâtiment (panneaux en bois par exemple). Une troisième activité, celle du fournisseur, est responsable de l'acheminement sur le chantier des matériaux choisis par l'ingénieur structure. Finalement, la quatrième activité, celle du poseur, représente la construction effective du bâtiment (assemblage des matériaux).

Cette application pourrait être représentée par le workflow de la figure 3.6-a. Bien que procédant par itérations, ce n'est cependant pas de cette manière que se déroulent réellement les différentes activités. Les trois activités correspondant à l'architecte, à l'ingénieur structure et au fournisseur sont plus compliquées. Elles s'exécutent plutôt en parallèle (figure 3.6-b) au sein d'une pseudo-activité de conception et non de façon isolée tel que cela est représenté par le workflow. En effet, elles s'échangent différents documents au cours de leur exécution afin de réagir le plus rapidement possible aux décisions prises par les autres partenaire. Supposons par exemple que le fournisseur, du fait de contraintes sur

Figure 3.6 Coordination des Activités d'un Projet de Construction de Bâtiment



le transport des panneaux en bois (longueur des panneaux limitée), décide de découper un panneau en plusieurs morceaux. Si l'architecte a placé une ouverture au niveau du raccord, le fournisseur devra demander à l'ingénieur structure de déplacer cette ouverture, ce qui peut nécessiter que ce dernier contacte l'architecte pour vérifier que les désirs du client sont toujours satisfaits. L'utilisation d'un modèle tel que celui de la figure 3.6-a risque donc de retarder la mise en évidence de ce problème jusqu'à la prochaine itération.

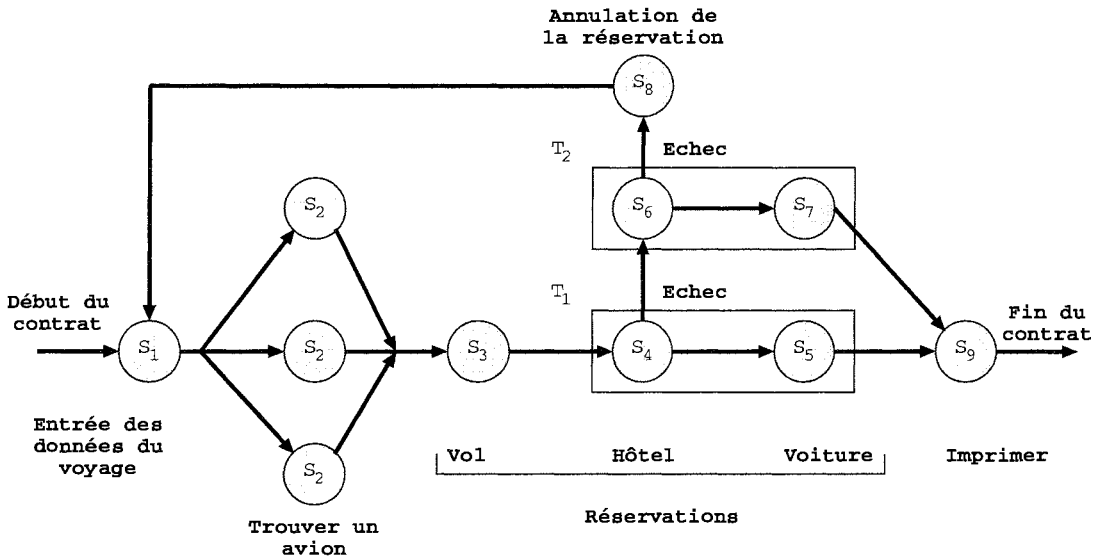
D'un autre côté, si nous essayons de représenter toutes ces interactions sur le diagramme de workflow, le risque est de devoir atteindre un niveau de granularité trop fin pour les activités et de surcharger le diagramme en tentant de décrire tous les enchaînements possibles des différentes activités. Le diagramme qui en résulterait serait bien trop complexe et pourrait rapidement devenir "inutilisable". En outre, cela signifie que l'on doit être capable de prévoir toutes les interactions entre activités.

Modèle des Contrats

L'idée de base du modèle des contrats (*"ConTract model"*) introduit dans [Wac92] est de construire les applications à partir de transactions courtes ACID (cf. paragraphe 3.3.1) et de fournir des mécanismes système indépendants des applications permettant de contrôler ces transactions. Un contrat est défini comme étant une exécution consistante et tolérante aux pannes d'une séquence d'actions prédéfinies (appelées étapes ou *"steps"*) selon un diagramme prédéfini (appelé *"script"*).

La figure 3.7 présente le "script" d'une réservation pour une voyage d'affaire. Intuitivement, à partir des données du voyage, il faut consulter les horaires d'avions, faire la réservation du vol, de l'hôtel et d'une voiture pour finir le contrat. $S1, S2, ..., S9$ représentent les actions prédéfinies du contrat. Le concepteur de ces actions n'a pas à prendre en compte les aspects concurrence d'accès, communication, synchronisation, pannes systèmes, ou encore recouvrement. C'est au gestionnaire de contrats de prendre en compte ces aspects

Figure 3.7 Script d'une Réservation pour un Voyage d'Affaire



Le concepteur de contrat définit quelles actions doivent être regroupées au sein de transaction ACID. Dans notre figure les actions "réserver l'hôtel et la voiture" doivent être groupées au sein d'une seule transaction ACID. Le concepteur peut décrire des relations entre transaction. Par exemple: si T_1 échoue alors lancer T_2. Un contrat lui-même n'est pas une transaction ACID, c'est-à-dire que les mises à jour peuvent être rendues publiques avant la fin du contrat. Un contrat est donc en mesure de lire des résultats partiels en provenance d'un autre contrat. Par exemple, si le vol est réservé, ce résultat est effectif sitôt l'action prédéfinie terminée.

Un contrat est une activité de longue durée: en cas d'interruption il faut être en mesure de le redémarrer dans un état consistant qui ne soit pas l'état initial et où les pertes de travail sont minimales. Un des points clefs du modèle réside dans la notion de recouvrement en avant: en cas d'incident, le gestionnaire de contrats réinstancie les contrats interrompus dans l'état consistant le plus récent et continue l'exécution du script. Dans ce but, chaque contrat dispose d'un contexte privé sauvegardé par le gestionnaire de contrats. Toutes les informations d'état du contrat y sont présentes. En cas d'incident, le contrat est réinstancié à partir de son contexte privé et le script continue.

Par ailleurs, étant donné qu'un contrat est en mesure de publier ses résultats avant qu'il ne soit terminé, si par malheur il échoue, il faut pouvoir se ramener à un état consistant qui n'est pas l'état initial du contrat. Le modèle introduit dans ce sens la notion de transactions de compensation pour défaire sémantiquement les effets des actions prédéfinies [GM87]. A chaque étape est associée une transaction de compensation. La compensation d'un contrat ne peut être déclenchée qu'en cas de demande explicite par l'utilisateur. Pour synchroniser les différent contrats, le modèle fait appel à l'idée d'invariance d'environnement. Chaque programme définit explicitement quelles sont les parties de son environnement qui doivent rester invariantes durant son exécution.

Le modèle des contrats autorise des exécutions non-isolées à l'image des sagas (cf. paragraphe 3.3.2). Comme pour les sagas, les résultats échangés entre deux transactions

doivent appartenir à l'ensemble des états consistants de la base puisqu'ils sont produits au sein de transactions ACID. Le modèle des contrats n'autorise donc pas les exécutions coopératives comme nous les avons définies, c'est-à-dire la possibilité de s'échanger des résultats intermédiaires "non-consistants" durant l'exécution. Cependant, l'idée d'un mécanisme de recouvrement en avant est intéressante pour éviter la perte de travail en cas de défaillance du système, ou plus simplement en cas d'interblocage entre les transactions dû aux règles de coopération qu'elles doivent respecter.

ClearGuide

Comme nous l'avons indiqué à la section 3.1, l'environnement de gestion de configurations ClearCase peut être associé à l'outils ClearGuide[9] pour assurer le suivi de production de logiciels. En tant qu'outils de gestion des procédés, ClearGuide combine à la fois des fonctionnalités de gestion de projet (planification et allocation des tâches, gestion des priorités, suivi des changements, ...) et des fonctionnalités de type workflow (description de l'enchaînement des tâches).

Avec ClearGuide, le travail de développement d'un logiciel est organisé en activités. Il n'existe, en général, qu'une seule activité racine représentant le projet global. Les sous-projets et les tâches individuelles sont représentés par des sous-activités. Le chef de projet et les développeurs peuvent à tout moment décomposer une activité existante en plusieurs sous-activités. Un projet de grande envergure est ainsi matérialisé dans ClearGuide par une hiérarchie d'activités.

Cette hiérarchie définit déjà implicitement un certain nombre de relations entre les différentes tâches du projet. Par exemple, le modèle hiérarchique implique naturellement que le travail d'une activité ne peut pas être considéré comme terminé tant que toutes les sous-activités ne sont pas terminées. Au delà de ces relations hiérarchiques, ClearGuide permet de représenter explicitement d'autres relations entre les activités au moment de la planification:

- **relations de suivi de projet :** Lorsqu'une activité est planifiée, elle est souvent synchronisée avec d'autres activités qui ne sont ni des ancêtres ni des descendants directs. ClearGuide fournit pour cela les dépendances de synchronisation habituelles en gestion de projet: *"end-to-start"* (X doit terminer avant que Y ne démarre), *"start-to-start"* (X doit démarrer avant que Y ne démarre), *"end-to-end"* (X doit terminer avant que Y ne termine), *"start-to-end"* (X doit démarrer avant que Y ne termine).
- **relations de gestion de configurations :** La plupart des activités d'un projet seront destinées à réaliser des modifications sur les données gérées par ClearCase: prototypage, nouvelle partie de code, correction d'une erreur, ... En conséquence, certaines relations de ClearGuide établissent une correspondance entre la hiérarchie des activités du projet et le système de contrôle des versions de ClearCase sous-jacent. En particulier, il est possible d'indiquer le point de départ d'une activité (configuration du code source en entrée) ainsi que le point d'intégration d'une acti-

9. ClearGuide: `http://www.rational.com/products/clearcase/prodinfo/03.jtmpl`

vité dans une autre (fusion des changements apportés). A l'intérieur d'une activité, les différentes étapes du développement sont représentées par des *"checkpoints"*.

Concernant les procédés exécutés par les différentes activités d'un projet, chaque activité de ClearGuide dispose, entre autres:

- d'un ensemble de **règles de procédé**: Ces règles sont associées à certaines opérations de ClearCase et de ClearGuide ainsi qu'à différents objets de l'environnement de développement (éléments, types de branches, ...). ClearGuide invoque ces règles de procédé automatiquement quand les opérations sont appelées (cf. règles ECA décrites ci-dessous). Ces règles permettent d'imposer automatiquement le respect de certaines exigences et restrictions sur les tâches.
- d'un **diagramme états/transitions**: Les différents états par lesquels passe une activité permettent de suivre et de mesurer la progression de cette activité. Ces diagrammes peuvent être très simples ou très détaillés selon la nature de l'activité considérée. Il s'agit d'un élément essentiel pour le suivi de projet. Les états d'une activité peuvent également permettre de déclencher certaines règles de procédé ou certaines opérations, voire même modifier les restrictions imposées à l'activité selon son état.

Par conséquent, pour coordonner les différentes activités, ClearGuide nécessite de connaître et de définir (diagramme états/transitions, règles ECA), à priori, les procédés qui seront exécutés par chacune des activités. De ce fait, une telle approche n'est pas adaptée à la nature incertaine des applications visées.

3.2.2 Règles Evénement/Condition/Action

Adèle-Tempo

Un exemple d'intégration de règles événement/condition/action au sein d'un gestionnaire de configurations est le couple Adèle-Tempo [Bel94]. Comme nous l'avons expliqué, Adèle est à la base un gestionnaire de configurations. Il a ensuite été étendu de manière à former un véritable environnement de développement centré procédés. Cette extension d'Adèle, nommée Tempo, permet de décrire les règles à respecter pour le partage des ressources. Tempo défini pour cela un langage à base de **règles de synchronisation** à l'aide duquel il est possible d'exprimer des diagrammes états/transitions caractérisant les exécutions correctes. Adèle propose un ensemble de stratégies prédéfinies dont le comportement a été expérimenté. Ces stratégies sont stockées dans des librairies à la disposition de l'administrateur.

Dans Adèle-Tempo, la coopération est donc plutôt vue comme un problème de "propagation" des modifications entre deux sous-bases se partageant un ensemble de ressources. Les activités se déroulent dans des espaces de travail et non dans des transactions. Il n'y a de ce fait pas véritablement de notion de correction des exécutions ni de propriétés à garantir sur les activités. En outre, cela signifie qu'il est également nécessaire de prévoir toutes les interactions possibles entre toutes les activités, ce qui peut rapidement devenir extrêmement complexe.

Il existe également une autre façon (complémentaire) de gérer les procédés dans Adèle: APEL[10]. Il s'agit d'un langage de plus haut niveau permettant de décrire graphiquement les procédés. Les spécifications de procédés réalisées dans ce langage peuvent ensuite être compilées puis utilisées avec deux produits commerciaux: Adèle et Process Weaver. Avec APEL, un procédé est réprésenté selon plusieurs aspects tels que le flot de contrôle, le flot de données, des diagrammes états/transitions ou les rôles des différents utilisateurs. APEL est donc très proche des systèmes de workflow.

Adèle propose néanmoins une approche pragmatique pour résoudre les problèmes de développement en parallèle sur les modèles utilisés en gestion de configurations.

MARVEL

Un autre exemple d'outils basé sur des règles événement/condition/action est MARVEL [Bar92a, Bar92b], un environnement de développement centré procédés développé par l'université de Columbia entre 87 et 92. Il s'agit d'un environnement basé sur la description des procédés à travers un ensemble de règles. Chaque fois qu'un utilisateur MARVEL émet une commande (compiler, editer, tester), MARVEL déclenche une règle correspondante. Cette règle contrôle la validité de la requête, implémente la commande et valide la post-condition de la règle. La validation de la post-condition peut à son tour déclencher une nouvelle règle en chaînage avant. L'échec de la pré-condition peut déclencher une autre règle par chaînage arrière. MARVEL n'utilise pas de mécanisme de base/sous-base.

L'évaluation de la partie condition d'une règle est atomique, de même que les évaluations des parties action et post-condition. L'atomicité est garantie par le référentiel de MARVEL et assure la correction des exécutions concurrentes de ces parties. Cependant, il peut y avoir entrelacement des exécutions entre les différentes parties des différentes règles. Pour garantir la correction de l'exécution d'une règle, MARVEL encapsule l'exécution d'une règle (et des règles déclenchées par chaînage avant et arrière) au sein d'une transaction.

MARVEL propose un contrôle de la concurrence d'accès basé sur le maintien de règles de cohérence. A la base, MARVEL détecte tous les conflits d'accès en implantant un verrou à deux phases. Si un conflit ne remet pas en cause une règle de cohérence, alors l'exécution continue, sinon une transaction doit échouer. De cette manière, MARVEL autorise les exécutions coopératives du moment qu'elles ne remettent pas en cause les règles de cohérence.

MARVEL suppose ainsi que l'on connaisse, à priori, tous les procédés qui seront exécutés par les différentes activités de manière à pouvoir définir les règles ECA, ce que ne permet pas la nature incertaine des applications visées.

3.2.3 Conclusion

Pour contrôler la coopération et la cohérence des différentes activités d'une application, les environnements centrés procédés sont basés sur la connaissance explicite des procédés

10. Abstract Process Engine Language

exécutés par ces activités. Dans le cadre d'applications coopératives distribuées (à large échelle) et hétérogènes, l'utilisation de tels environnements pour coordonner les différentes activités ne résoud toutefois pas tous les problèmes:

- **distribution** : La plupart du temps, la gestion des procédés est centralisée, ceci afin de faciliter la synchronisation et l'administration de l'activité globale.
- **complexité** : Il s'agit généralement de décrire le système complet, c'est-à-dire en tenant compte de toutes les activités et de toutes les ressources. Si l'on intègre les problèmes liés à la synchronisation des activités distribuées, les diagrammes états/transitions ou de flots de tâches deviennent rapidement complexes, voire même "inutilisables". Cette façon de procéder peut devenir source d'erreurs.
- **hétérogénéïté** : En ce qui concerne le domaine des entreprises-projet, chaque participant doit pouvoir travailler comme il le désire. Il n'est pas réaliste de vouloir donner une description du procédé général: d'une part nous ne voulons pas imposer aux partenaires une façon de travailler particulière, et d'autre part une entreprise peut, pour des raisons de confidentialité par exemple, refuser de fournir la description de ses procédés de conception.

Contrairement aux outils de gestion de procédés existants, notre objectif n'est donc pas de décrire comment les activités doivent travailler pour pouvoir coopérer, mais simplement de définir de quelle manière doivent se dérouler les échanges entre ces activités. Nous voulons imposer des contrôles sur les échanges, mais pas sur les activités elles-mêmes.

En outre, à la différence des systèmes basés sur des règles ECA par exemple, nous ne voulons pas obliger les activités à exécuter certaines opérations. Etant donnée la nature particulière des activités dans les applications visées (forte interactivité avec les utilisateurs, longue durée, ...), nous préférons une approche dans laquelle les opérations sont invoquées à l'initiative des activités elles-mêmes, mais dont l'exécution peut être refusée par le système si certaines règles ne sont pas respectées. L'idée sous-jacente est qu'une activité (représentant éventuellement le travail d'un utilisateur) doit rester maître de ses actions.

3.3 Systèmes Transactionnels

Garantir la cohérence des informations en cas d'accès concurrents (et éventuellement de défaillances) nécessite la mise en œuvre de protocoles permettant de synchroniser les traitements concurrents d'une part, et d'annuler ou de compléter les effets partiels des activités interrompues par des défaillances d'autre part. Ces protocoles sont complexes et le fait de les programmer explicitement est souvent source d'erreurs. De plus en plus de systèmes, dits transactionnels [Gra93, Bes97], offrent maintenant de tels protocoles, ce qui évite aux utilisateurs de ces systèmes de les programmer eux-mêmes. Ces protocoles, appelés **protocoles transactionnels**, sont encapsulés dans le concept de **transaction**.

3.3.1 Notion de Transaction

De manière schématique, une transaction peut être vue comme un script décrivant les différentes opérations (lectures et écritures par exemple) invoquées par les différentes activités sur les objets partagés. Dans le cas d'une application bancaire par exemple, une transaction peut représenter une séquence d'opérations de consultation de certains comptes, ou un simple retrait d'argent à partir d'un compte donné, ou encore un virement d'une somme d'argent d'un compte sur un autre. De manière plus précise, une transaction est une séquence d'opérations sur un ensemble d'objets qui vérifie les propriétés suivantes:

(A) **Atomicité**: Soit une transaction se termine correctement et elle a les effets désirés sur les objets manipulés (la transaction est "validée"), soit la transaction est interrompue et elle n'a aucun effet sur ces objets (elle est "abandonnée"). On parle également d'"atomicité aux défaillances" ou de règle du "tout ou rien".

(C) **Cohérence**: Le "script" d'une transaction est supposé correct. Cela signifie qu'une transaction, en dehors des problèmes liés à la concurrence d'accès, doit préserver la cohérence des objets qu'elle manipule.

(I) **Isolation**: Les effets d'une transaction sont invisibles aux transactions concurrentes. Cela signifie que les états intermédiaires d'une transaction, en termes de valeurs des objets qu'elle manipule, ne sont pas visibles par les autres transactions du système tant qu'elle n'est pas validée. Cette propriété est également appelée "atomicité à la concurrence".

(D) **Durabilité**: Les effets d'une transaction validée sont permanents et ne peuvent plus être remis en cause, ni par une défaillance, ni par une autre transaction. Remettre en cause cette propriété de durabilité signifie que l'on autorise les exécutions partielles (i.e. incomplètes).

Ces quatre propriétés sont appelées les propriétés **ACID** des transactions. Lorsque ces propriétés sont satisfaites, la cohérence des objets du système est assurée malgré les défaillances et l'exécution simultanée de plusieurs traitements manipulant les mêmes objets.

Pour illustrer ces propriétés, reprenons l'exemple des comptes bancaires. Supposons que l'opération de virement entre deux comptes A et B (débit sur le compte A puis crédit sur le compte B) soit mise en œuvre au sein d'une transaction. La propriété de cohérence indique simplement que l'opération de virement en elle-même est correcte, i.e. que le montant crédité doit être identique au montant débité. La propriété d'isolation permet d'éviter les interférences entre les transactions, et ainsi d'éviter le cas où la consultation des deux comptes se produirait au milieu du virement tel que cela est représenté sur la figure 3.8. Sur cet exemple, la balance calculée accuse un déficit de $500FF$. Ce résultat est donc inconsistant par rapport aux comptes A et B. La propriété d'atomicité garantit qu'en cas de défaillance du système au cours du virement la transaction chargée d'effectuer ce virement est abandonnée et que toutes ses modifications sont annulées. Si au contraire cette transaction se termine correctement, i.e. si elle est validée, alors la propriété de durabilité nous assure que ses modifications ne seront pas remises en cause, ni par des défaillances ultérieures, ni par une autre transaction.

Figure 3.8 Propriété d'Isolation Non Respectée

virement($compte_A \rightsquigarrow compte_B$, $500FF$)	balance($compte_A + compte_B$)
$montant_A \leftarrow read(compte_A)$	
$montant_B \leftarrow read(compte_B)$	
$write(compte_A, montant_A - 500FF)$	
	$solde_A \leftarrow read(compte_A)$
	$solde_B \leftarrow read(compte_B)$
	$write(balance, solde_A + solde_B)$
$write(compte_B, montant_B + 500FF)$	

La propriété de cohérence est en général du ressort de l'utilisateur de système (i.e. de celui qui programme les transactions) car elle ne porte que sur le contenu de chaque transaction, considérée indépendamment des autres transactions et sans prendre en compte les défaillances. Ainsi, en assurant de son côté les propriétés d'atomicité, d'isolation et de durabilité, un système transactionnel permet de réduire le problème du maintien de la cohérence d'un ensemble de transactions concurrentes en milieu non fiables (défaillances possibles), au problème beaucoup plus simple du maintien de la cohérence individuelle de chacune des transactions.

3.3.2 Contrôle de la Concurrence

Le contrôle de la concurrence dans les systèmes transactionnels a pour objectif de permettre l'exécution concurrente de transactions, tout en garantissant à chaque transaction que les effets indésirables dus à la concurrence, tels que l'introduction d'une incohérence entre les objets ou la production par les transactions de résultats incohérents, sont soit évités (modèles de transactions classiques) soit compensés (modèles de transactions étendus). Les systèmes transactionnels se basent pour cela sur un **critère de correction**.

Un tel critère est un ensemble de propriétés qui caractérisent l'histoire (liste ordonnée des opérations invoquées par les transactions) des exécutions considérées comme correctes. L'histoire de l'exécution entremêlée de plusieurs transactions est généralement représentée par la séquence des opérations (lecture, écriture, ...) invoquées sur les objets. Le critère de correction le plus répandu dans le domaine des applications traditionnelles est la **sérialisabilité**. Celui-ci considère que l'exécution entremêlée de plusieurs transactions est correcte (l'exécution est dite sérialisable) si elle produit un résultat équivalent à une exécution en série de ces transactions.

Modèles de Transactions Classiques

Un modèle de transactions dit "classique" a pour but de permettre à de multiples utilisateurs d'accéder simultanément à un ensemble d'objets partagés, tout en leur garantissant les propriétés d'atomicité, d'isolation et de durabilité (AID). Pour cela, le contrôle de la concurrence associé au modèle repose sur le principe de l'équivalence des exécutions concurrentes à une exécution en série des transactions, encore appelé principe de la sérialisabilité des transactions. Le respect de ce principe se traduit, pour les transac-

tions, par la propriété d'isolation. Il garantit, à la condition que chaque transaction soit elle-même cohérente, que les exécutions concurrentes sont correctes, c'est-à-dire qu'elles n'introduisent pas d'incohérence entre les objets et que les transactions produisent des résultats cohérents.

Etant donné qu'il est de plus en plus fréquent qu'une application accède à des objets situés sur plusieurs sites, différentes techniques ont été développées pour permettre la mise en œuvre de ces concepts de base des transactions au sein d'architecture non plus centralisée mais répartie:

- **transactions réparties**: Les différentes transactions de l'application s'exécutent sur des sites géographiquement distribués. Afin de préserver les propriétés AID il est donc nécessaire d'utiliser des mécanismes de vérouillage réparti, de détection des interblocages répartis, ou encore de validation atomique répartie[11].
- **duplication**: La duplication des objets sur plusieurs sites permet d'assurer la disponibilité de ces objets afin d'améliorer la tolérance aux défaillances (plusieurs copies d'un même objet) et de supporter le travail en mode déconnecté (copie des objets sur un ordinateur portable par exemple). Bien que le principe de duplication soit très intuitif, sa mise en œuvre s'avère relativement complexe si on veut assurer que, malgré les accès concurrents et les défaillances, les différentes copies d'un même objet gardent le même état et retournent les mêmes valeurs.
- **systèmes multibases**: Le principe d'un système multibase consiste à fédérer différentes sources de données hétérogènes et autonomes pour donner l'illusion à l'utilisateur d'une base de données homogène et centralisée. Chaque base de données intègre son propre gestionnaire de transactions locales dont le rôle est de garantir les propriétés AID des transactions soumises à son contrôle. Du fait de l'autonomie des sites, chaque gestionnaire de transactions locales est inconscient de son intégration dans un système multibase. Lorsqu'une transaction dite "globale" accèdera à plusieurs bases de données pour manipuler différents objets elle sera donc soumise au contrôle de plusieurs gestionnaires locaux. Un contrôle global complémentaire est alors nécessaire pour préserver la sérialisabilité et l'atomicité des transactions globales.

Quels que soient les mécanismes utilisés pour contrôler la concurrence entre des transactions distribuées, nous constatons qu'à la manière d'un système centralisé ils ont tous pour objectif de garantir les propriétés AID des transactions et sont tous basés sur le principe de la sérialisabilité des transactions.

Toutefois, si ces modèles de transactions (transactions plates[12], ACID et sérialisables) sont adaptés aux applications transactionnelles classiques (telles que les applications bancaires, les systèmes de réservation de place, ...) dont la durée des transactions est de l'ordre de quelques secondes, il n'en est pas de même en ce qui concerne les applications

11. La validation atomique peut se dérouler en deux phases (2PC ou *Two Phases Commit* [Gra78], D2PC ou *Distributed Two Phases Commit* [Ber87]) ou en trois phases (3PC) tel le protocole de validation atomique non-bloquant développé par Skeen [Ske81].

12. Toutes les transactions se trouvent au même niveau, i.e. ne sont pas organisées, à l'inverse d'une structure hiérarchique, réseau, ...

qui nous intéressent (entreprises virtuelles, informatique mobile, ...) dont les transactions interactives peuvent durer plusieurs jours, voire même plusieurs semaines. En effet, dans le cadre des applications coopératives distribuées, l'organisation à plat des transactions, les propriétés ACID et la sérialisabilité présentent des limitations importantes, à la fois du point de vue de la coopération, de la distribution et de l'hétérogénéïté. Dans le cas de transactions de longue durée, la propriété d'atomicité peut par exemple avoir pour conséquence de faire perdre le travail de plusieurs jours si, à la suite d'une défaillance, une transaction doit être abandonnée. Quant à la propriété d'isolation (obtenue via la sérialisation des transactions), outre le fait qu'elle est en contradiction avec la notion de coopération telle que nous l'avons définie (les transactions peuvent s'échanger des résultats intermédiaires au cours de leur exécution), elle risque également de conduire, en cas de conflit, soit à des interblocages soit à des abandons en cascade, ce qui n'est pas acceptable dans le cas de transactions longues.

Pour limiter l'impact de ces conséquences, de nouveaux modèles de transactions dits "étendus" ont donc été développés afin de relâcher ou d'assouplir une ou plusieurs des propriétés ACID, d'organiser les transactions selon les spécificités de l'application (parallèlisme local, décomposition d'une transaction en sous-transactions, ...) ou de permettre l'utilisation de critères de corrections moins contraignants que la sérialisabilité.

Modèle de Transactions Emboîtées

Dans le modèle de transactions emboîtées (*"nested transactions"*) introduit dans [Mos81], une transaction peut être décomposée en plusieurs transactions, appelées sous-transactions[13], et ainsi de suite. Les transactions sont donc organisées de manière hiérarchique, la transaction se trouvant à la racine de la hiérarchie représentant la transaction globale. Ces relations structurelles entre les transactions nous permettent alors de définir des règles de contrôle de la concurrence moins strictes que dans les modèles de transactions classiques.

- **règle d'abandon** : L'abandon d'une transaction entraîne l'abandon de toutes ses sous-transactions mais pas celui de sa transaction mère. Cette dernière est libre de poursuivre son exécution, soit en démarrant une nouvelle sous-transaction soit en reprenant la sous-transaction abandonnée, ou d'abandonner à son tour. Par conséquent, les transactions de ce modèle sont plus résistantes aux défaillances que celles des modèles de transactions classiques puisque les conséquences de l'abandon d'une transaction sont limitées à l'abandon des sous-transactions appartenant à sa descendance.
- **règle de validation** : La validation d'une sous-transaction emboîtée est conditionnée par celle de sa (sous-)transaction mère. Cela signifie d'une part que l'abandon d'une transaction entraîne l'abandon de toutes les sous-transactions de sa descendance même si elles sont déjà validées, et d'autre part que les mises à jour effectuées par une sous-transaction validée ne deviennent durables qu'une fois toutes ses transactions ancêtres validées, c'est-à-dire à la validation de la transaction globale.

13. Une sous-transaction doit démarrer après sa transaction mère et se terminer avant elle.

- **règle de visibilité**: Tous les objets détenus par une transaction peuvent être rendu accessibles à ses sous-transactions[14]. Les mises à jour effectuées par une sous-transaction deviennent visibles à sa transaction mère quand la sous-transaction est validée. Ces modifications ne sont toutefois pas visibles par ses sous-transactions sœurs.

De cette manière, si le modèle des transactions emboîtées garantit effectivement les propriétés ACID à la transaction racine, la seule contrainte sur une sous-transaction (supposée cohérente) est qu'elle doit être atomique et isolée par rapport à toute sous-transaction qui n'appartient pas à sa descendance (propriétés ACI). Outre l'aspect résistance aux défaillances (particulièrement intéressant dans le cas de transactions longues), le modèle des transactions emboîtées introduit également un certain degré de parallèlisme en permettant la décomposition d'une transaction en plusieurs sous-transactions pouvant s'exécuter en parallèle. Ce modèle est donc tout à fait adapté aux programmeurs qui peuvent ainsi décomposer un module en sous-modules concurrents sans avoir à coder explicitement le contrôle de la concurrence.

Toutefois, le critère de correction utilisé pour synchroniser les transactions concurrentes reste la sérialisabilité, l'objectif étant d'assurer l'isolation entre les différentes transactions globales d'une part, et l'isolation de chaque sous-transaction par rapport à toutes les autres sous-transactions ne faisant pas partie de sa descendance d'autre part. Or, dans le cadre des applications coopératives distribuées, nous voulons justement relâcher la propriété d'isolation entre les différentes transactions.

En outre, la structure hiérarchique imposée par le modèle des transactions emboîtées, si elle est adaptée à la décomposition d'un programme (conception modulaire), est moins naturelle dans le cas d'une entreprise virtuelle par exemple. En effet, étant donné que seules les transactions feuilles ont la possibilité de modifier les objets, toutes les transactions encapsulant les tâches des différents partenaires devront être des transactions feuilles. Ceci nous oblige ensuite à définir, à priori, de nouvelles transactions nœuds, ne correspondant à aucune tâche réelle de l'application, dans le seul but de coordonner celles des partenaires. Intuitivement, une organisation d'égal-à-égal des transactions (structure en réseau) serait donc plus adéquate.

Modèle de Transactions Multiniveaux

Le modèle des transactions multiniveaux (*"multi-level transactions"*) introduit dans [Bee88a] est dérivé du modèle des transactions emboîtées pour permettre de prendre en compte la sémantique des objets manipulés (en exploitant les propriétés de commutativité de leurs opérations) ainsi que la manière dont ils sont construits[15] (en se basant

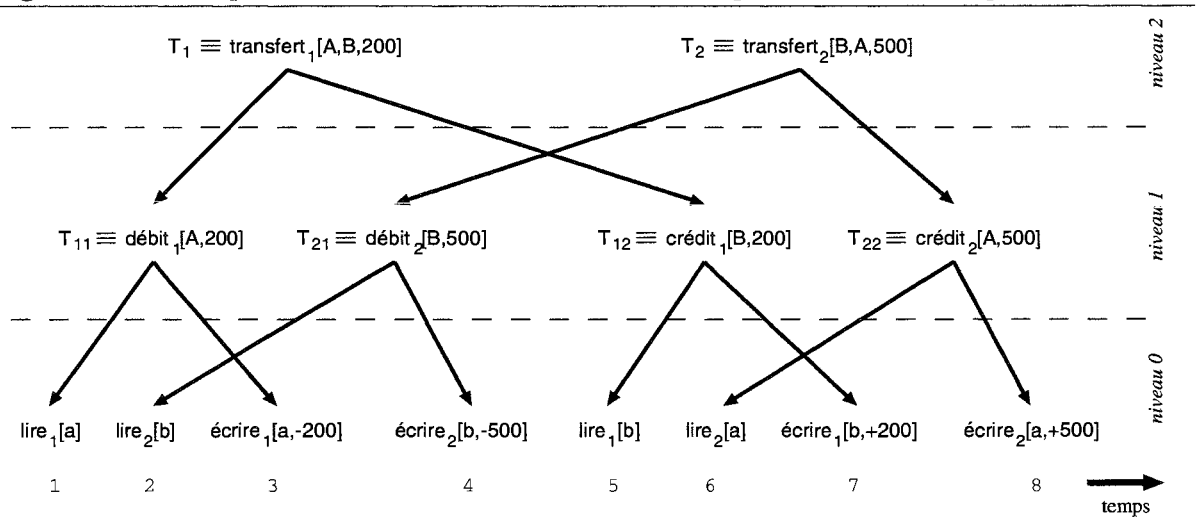
14. Le modèle des transactions emboîtées le plus simple suppose que seules les transactions feuilles de la hiérarchie peuvent invoquer des opérations sur les objets. Une transaction nœud se limite à coordonner ses sous-transactions, ce qui évite les problèmes d'isolation entre une transaction mère et ses sous-transactions filles.

15. Un objet X manipulé par une application est construit par niveaux d'abstraction successifs à partir d'objets existants. Ainsi, à partir d'objets primitifs (ex: pages, segments, ...) manipulables par des opérations élémentaires (lire, écrire), sont obtenus de proche en proche des objets typés plus élaborés, manipulables par des opérations appelées méthodes.

sur la propriété d'indépendance entre les objets[16] [Car90a]).

Prenons l'exemple d'une opération de transfert entre deux comptes bancaires. Une telle opération $transfert[A, B, value]$ sera en fait décomposée en deux sous-opérations $debit[A, value]$ et $credit[B, value]$ qui seront exécutées respectivement sur les objets comptes bancaires A et B, eux-mêmes construits à partir des objets a et b du niveau inférieur intervenant dans les représentations de A et B. L'opération $debit[X, v]$ portant sur l'objet compte bancaire X peut alors à son tour être décomposée en deux opérations élémentaires $lire[x]$ et $ecrire[x, -v]$ portant sur la représentation x de l'objet X. De la même manière, l'opération $credit[X, v]$ sera composée des deux opérations élémentaires $lire[x]$ et $ecrire[x, +v]$.

Figure 3.9 Exemple de Transactions Multiniveaux: Opérations sur Comptes Bancaires



La figure 3.9 représente un exemple d'exécution concurrente de deux opérations de transfert T_1 et T_2, la première du compte A vers le compte B pour un montant de $200FF$ et la seconde du compte B vers le compte A pour un montant de $500FF$. Si l'on effectue le contrôle de la concurrence au niveau 0, i.e. sur les opérations élémentaires *lire* et *ecrire* (comme dans le cas des transactions plates), nous constatons que cette exécution n'est pas sérialisable. En effet, à l'étape 5 la transaction T_1 devient dépendante de la transaction T_2 (noté $T_2 \rightarrow T_1$)[17], et à l'étape 6 la transaction T_2 devient elle-même dépendante de la transaction T_1 (noté $T_1 \rightarrow T_2$)[18].

Toutefois, si l'on considère maintenant les sous-transactions T_{ij} (les dépendances sont $T_{11} \rightarrow T_{22}$ et $T_{21} \rightarrow T_{12}$), nous constatons que ces sous-transactions sont sérialisables. Puisqu'elles sont isolées l'une de l'autre, nous pouvons alors exploiter les propriétés de commutativité des opérations *debit* et *credit* qui leur correspondent. Nous pouvons ainsi permuter les sous-transactions T_{ij} de telle manière que les transactions T_1 et T_2 soient sérialisables avec un contrôle de la concurrence réalisé sur les opérations *debit* et *credit* (opérations du

16. Deux objets sont dits indépendants quand une modification de l'un n'a pas de conséquence sur la représentation de l'autre.

17. T_1 lit la valeur de b écrite par T_2 à l'étape 4

18. T_2 lit la valeur de a écrite par T_1 à l'étape 3

niveau 1). Seule l'isolation des opérations *debit* et *credit* est donc requise, i.e. il n'est plus nécessaire que ces opérations soient indivisibles (opérations élémentaires).

En considérant les opérations comme divisibles et en exploitant la commutativité des opérations et l'indépendance des objets, le modèle des transactions multiniveaux permet donc d'augmenter le degré de concurrence entre les transactions (davantage d'entrelacements) tout en préservant leurs propriétés ACID. Les transactions de ce modèle restent toutefois isolées les unes des autres (sérialisabilité multiniveaux) et ne peuvent donc pas coopérer au sens où nous l'avons défini dans le cadre des applications coopératives distribuées (échanges de résultats intermédiaires au cours de leur exécution).

Modèle des Sagas

Le modèle des sagas introduit dans [GM87] permet de résoudre le problème de l'isolation dans le cas de transactions longues. Une telle transaction, appelée **saga**, sera décomposée en plusieurs sous-transactions ACID, à la manière d'un modèle de transactions emboîtées à deux niveaux. Les règles de contrôle de la concurrence ne sont toutefois pas les mêmes. En premier lieu, une saga (transaction racine) ne possède pas la propriété d'isolation alors que ses sous-transactions l'ont. Par conséquent, une sous-transaction d'une saga peut voir les résultats partiels d'une autre saga. Deuxièmement, contrairement au modèle des transactions emboîtées, la validation d'une sous-transaction n'est pas conditionnée par la validation de la saga dont elle fait partie. Finalement, l'abandon d'une sous-transaction entraîne l'abandon de la saga correspondante.

Comme nous pouvons constater, ces deux dernières règles peuvent poser quelques problèmes dans le cas où une saga doit abandonner alors que les mises à jour de certaines de ses sous-transactions ont déjà été rendues durables dans la base (et éventuellement utilisées par les sous-transactions d'autres sagas). A cet effet, le modèle des sagas associe à chaque sous-transaction t_i une sous-transaction de compensation ct_i dont l'objectif est de défaire sémantiquement les opérations effectuées par la sous-transaction t_i. Prenons par exemple le cas d'une saga composée des sous-transactions ACID $t_0, t_1, \ldots, t_n$. Si la sous-transaction t_j de cette saga échoue, alors la saga elle-même doit abandonner. Elle exécute pour cela les sous-transactions de compensation $ct_0, \ldots, ct_j$ associées aux sous-transactions $t_0, \ldots, t_j$ pour préserver la cohérence de la base. Ces sous-transactions de compensation sont exécutées dans l'ordre inverse de celui dans lequel les sous-transactions ont été validées. Ainsi, l'exécution de cette saga (abandonnée à la sous-transaction t_j) est la suivante: $t_0, \ldots, t_j, ct_j, ct_{j-1}, \ldots, ct_0$

Le modèle des sagas permet donc aux sous-transactions de voir les résultats intermédiaires des autres sagas. Ces résultats intermédiaires sont toutefois des résultats cohérents (du point de vue de la base de données) puisqu'ils sont produits par d'autres sous-transactions ACID. Le modèle des sagas n'autorise donc pas les exécutions coopératives au sens où nous avons défini la coopération entre les transactions (échanges de résultats intermédiaires potentiellement incohérents au cours de leur exécution). De plus, ce modèle suppose que chaque sous-transaction ct_i doit avoir été initialement définie et prévue par le concepteur de la saga pour annuler les effets de la sous-transaction validée t_i correspondante. Outre la difficulté que peut rencontrer le concepteur de la saga pour la découper en sous-transactions ACID, il se peut également que certaines de ces sous-

transactions soient difficilement compensables. En effet, exécuter une sous-transaction de compensation ct_i revient en quelque sorte à remettre en cause les résultats (rendus durables dans la base) de la sous-transaction t_i. S'il s'agit d'un document contractuel signé par deux partenaires d'une entreprise virtuelle par exemple, cette remise en cause peut devenir problématique. Il serait donc préférable de permettre aux (sous-)transactions de coopérer **avant** d'être validées, c'est-à-dire au cours de leur exécution (transactions non isolées).

Modèles de Transactions Coopérantes

Les modèles de transactions que nous avons étudiés jusqu'à présent sont tous indépendants des applications. En effet, qu'il s'agisse des transactions plates, emboîtées, multiniveaux ou encore des sagas, les notions définies par ces modèles ne sont pas liées à la sémantique des applications visées. Afin de relâcher les propriétés ACID des transactions (notamment en ce qui concerne la contrainte d'isolation), une autre solution consiste à définir des modèles de transactions spécifiques à des domaines particuliers.

C'est le cas par exemple des transactions coopératives introduites dans [Nod92]. Le contrôle de la concurrence est réalisé sur la base d'une grammaire (dont les terminaux sont les opérations) qui caractérise les exécutions correctes. Une part importante du contrôle de la cohérence est ainsi reportée sur les utilisateurs qui devront écrire une grammaire correcte. Il est en outre difficile dans ce cas de garantir des propriétés générales sur les exécutions.

D'autres solutions pour assurer la coopération entre les différentes transactions consistent à étendre les modes de verrouillage permettant de contrôler les conflits d'accès aux objets et à définir les actions à exécuter en cas de conflit. Là encore il s'agit de solutions pragmatiques ressemblant plus à une combinaison de mécanismes de contrôle des accès concurrents (versions, notifications, ...) mis en œuvre dans les gestionnaires de configurations qu'à de véritables modèles de transactions définis de manière formelle.

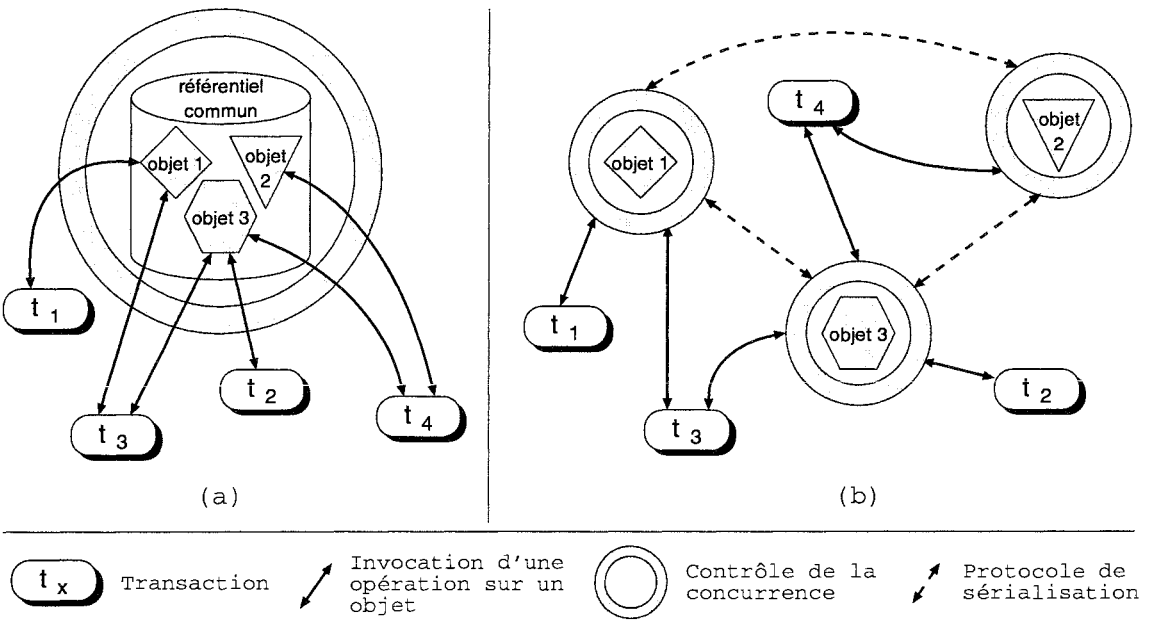
Objets Atomiques

Comme nous pouvons le constater, les modèles de transactions sont généralement définis au-dessus d'une base de données centralisée, i.e. ils supposent que toutes les transactions accèdent à un unique référentiel commun (figure 3.10-a). Cela permet en particulier de centraliser le contrôle de la concurrence. Celui-ci dispose ainsi d'une vue globale du système puisqu'il a connaissance de toutes les opérations invoquées par toutes les transactions sur les différents objets.

Une autre solution (travaux réalisés dans le cadre des langages concurrents à objets) consiste à intégrer le concept de transaction directement au niveau des objets en concevant des mécanismes permettant d'assurer la sérialisabilité. Ces mécanismes ont pour rôle de synchroniser les invocations des objets, appelés **objets atomiques** [Wei89, Wei84], concernés par la sérialisabilité (objets comptes bancaires par exemple). Chaque objet devient ainsi responsable de la gestion de ses accès concurrents effectués par les différentes transactions qui le manipulent (figure 3.10-b). Un protocole de sérialisation (identique pour tous les objets atomiques) permet ensuite de garantir que tous les objets atomiques

manipulés par une transaction définissent le même ordre de sérialisation pour cette transaction par rapport aux transactions concurrentes. En ce qui concerne la détection et le traitement des conflits, chaque objet atomique est libre de définir ses propres mécanismes. La prise en compte de la sémantique des objets dans ces mécanismes peut d'ailleurs permettre d'augmenter la concurrence entre les transactions [Car89].

Figure 3.10 Objets Atomiques et Contrôle Local



L'utilisation d'objets atomiques permet donc de délocaliser le contrôle de la concurrence vers les objets eux-mêmes. Ceci est particulièrement intéressant dans le cas où ces objets sont distribués (cas des systèmes multibases par exemple). Cette façon de procéder n'est par contre pas adaptée au cas où il existe plusieurs copies d'un même objets logique (chaque transaction possède sa propre copie des objets qu'elle manipule par exemple).

3.3.3 Conclusion

Le principe de base des systèmes transactionnels (à savoir simplifier la programmation des applications en cachant au maximum la complexité introduite par l'exécution concurrente des différentes activités) correspond donc exactement à notre approche du "contrôle de la coopération" dans le cadre des applications coopératives distribuées et hétérogènes. Toutefois, les modèles de transactions et les critères de correction existants ne sont pas adaptés à nos besoins de:

- **coopération**: Le principal écueil est bien évidemment la propriété d'isolation des transactions. Elle est en contradiction avec nos objectifs et notre définition de la coopération, à savoir la possibilité offerte aux transactions de s'échanger des résultats intermédiaires (potentiellement incohérents) au cours de leur exécution. Un nouveau modèle de transactions (les *COO*-transactions) ainsi qu'un nouveau critère

de correction (la *COO*-sérialisabilité) ont déjà été définis dans [Mol96] pour relâcher cette propriété d'isolation.

- **distribution**: Les modèles de transactions sont généralement définis au-dessus d'une base de données commune (éventuellement répliquée et/ou répartie sur quelques serveurs), i.e. ils supposent que toutes les transactions accèdent à un unique référentiel commun. Cette dépendance des transactions vis-à-vis de ce référentiel centralisé (i.e. vis-à-vis des serveurs) peut devenir extrêmement pénalisante dans le cas de transactions géographiquement distribuées et connectées de manière occasionnelle.

  L'idée est de permettre à chaque transaction de disposer de sa propre copie des objets qu'elle manipule et de définir des mécanismes pour synchroniser ces différentes copies d'un même objet logique, tout en préservant les mêmes propriétés que dans le cas d'un référentiel commun. Un travail similaire a déjà été réalisé dans [Ber87] en ce qui concerne la sérialisabilité avec la définition de la "sérialisabilité à une copie" (*"one-copy serializability"*)[19]. Notre démarche se rapproche également des critères de cohérence (linéarisabilité, cohérence causale, ...) définis dans le domaine des systèmes distribués pour résoudre les problèmes de synchronisation entre processus.

- **autonomie**: L'utilisation d'un référentiel commun à toutes les transactions permet également de centraliser le contrôle de la concurrence. Etant donné que nous nous orientons vers une solution où chaque transaction disposera de sa propre copie des objets qu'elle manipule de manière à être autonome du point de vue des données, il serait également souhaitable que le contrôle de la concurrence soit lui aussi distribué, i.e. que chaque transaction soit elle-même responsable de la cohérence de ses propres objets.

  De ce point de vue, notre objectif est de définir des propriétés locales, c'est-à-dire des propriétés qui, lorsqu'elle sont assurées par chaque transaction de façon individuelle, sont également garanties "implicitement" au niveau du système global. Notre démarche se rapproche ainsi de ce qui a été réalisé dans les systèmes transactionnels avec les objets atomiques (décrits précédemment) du point de vue des critères de correction distribués (mais en délocalisant le contrôle sur les transactions et non sur les objets).

- **hétérogénéïté**: Dans le cas d'une entreprise virtuelle par exemple, l'hétérogénéïté des relations entre les différents partenaires rend très difficile la définition d'un critère de correction unique permettant de caractériser toutes les exécutions correctes. Et la nature incertaine de ces transactions (i.e. le "programme" n'est pas connu à l'avance) complique encore le problème. Par conséquent, des parties différentes du système, voire même d'une transaction, pourront être soumises à des règles de contrôle différentes. Il faudra alors être capable d'intégrer ces différentes règles au sein d'un même modèle de transactions.

19. Intuitivement, la sérialisabilité à une copie exige que l'exécution des transactions sur les copies d'objets soit équivalente à une exécution en série de ces mêmes transactions sur des objets logiques: tout se passe comme si les transactions s'exécutent en série sur des objets non-dupliqués.

Ainsi, pour assurer le contrôle de la coopération dans le cadre des applications coopératives distribuées et hétérogènes, notre objectif est de définir un nouveau modèle de transactions avancé. Comme nous l'avons expliqué, celui-ci devra intégrer à la fois les notions de coopération (cf. *COO*-sérialisabilité), de multiplicité des copies d'un même objet logique, de distribution des critères de correction, et d'hétérogénéïté des règles de contrôle.

3.4 Outils d'Aide au Travail Coopératif

Qu'il s'agisse des gestionnaires de configurations, des outils de gestion des procédés ou des systèmes transactionnels, l'objectif est de coordonner un ensemble d'activités du point de vue des changements qu'elles effectuent sur les ressources du système. La coopération ne se limite cependant pas à la gestion des ressources partagées. En effet, il ne faut pas oublier que dans notre cas (entreprises-projet), il s'agit d'activités généralement interactives, i.e. dirigées par des utilisateurs (ou acteurs). Des aspects "humains" interviennent alors, tels que la gestion d'un groupe de personnes, les mécanismes de notification, les techniques de communication (messagerie électronique, vidéo-conférences, ...). Ce domaine, appelé CSCW[20], a pour objectif d'utiliser les technologies informatiques (les interactions hommes/machines, les réseaux, le multimédia, des concepts orientés objet, de réalité virtuelle et d'intelligence artificielle, ...) pour faciliter la collaboration entre les individus. Les logiciels de ce domaine sont appelés "collecticiels" ou "synergiciels".

3.4.1 Mécanismes pour la Collaboration

Le principal objectif des collecticiels est donc de définir/découvrir de nouvelles utilisations des technologies informatiques existantes afin d'améliorer le travail en groupe selon les deux dimensions: l'espace et le temps.

- **l'espace** :
  - centralisé : les acteurs se trouvent au même endroit,
  - distribué : les participants sont géographiquement répartis sur plusieurs sites.
- **le temps** :
  - temps réel : les actions et réactions ont lieu en même temps,
  - asynchrone : les actions et réactions ont lieu à des instants différents.

Ces deux critères nous permettent alors de classer les collecticiels en quatre catégories (figure 3.11). Les systèmes offrant des interactions synchrones entre les différents participants sont appelés "systèmes collaboratifs en temps réel" (catégories 1 et 3). Ils permettent à chaque participant de voir instantanément les modifications faites par les autres, même s'ils sont éloignés géographiquement (éditeurs partagés par exemple). Dans le cas des "systèmes collaboratifs asynchrones" (messagerie électronique par exemple), la propagation des modifications effectuées par un des participants à ses collaborateurs peut au contraire prendre un temps relativement important.

Cette classification n'est cependant pas stricte. Un collecticiel peut en effet appartenir à plusieurs catégories. C'est le cas par exemple d'un éditeur partagé qui permettrait

20. Computer Supported Cooperative Work

Figure 3.11 Classification Espace/Temps des Collecticiels

Espace \ Temps	*temps réel*	*asynchrone*
centralisé	**1** interactions synchrones en face à face ex: - réunion électronique - tableau blanc partagé	**2** interactions asynchrones centralisées ex: - forums électroniques (bulletin board)
distribué	**3** interactions synchrones distribuées ex: - conférence assistée par ordinateur - éditeur partagé	**4** interactions asynchrones distribuées ex: - agenda électronique partagé - messagerie électronique (email)

une collaboration synchrone entre plusieurs participants, qu'ils soient localisés au même endroit ou répartis sur des sites différents. Un tel collecticiel appartiendrait à la fois aux catégories 1 et 3.

Comme nous l'avons indiqué, le domaine de CSCW utilise les techniques développées dans d'autres domaines tels que les systèmes distribués, la communication, la conception d'interfaces homme-machine, les sciences humaines, ... Un collecticiel combine ainsi ces différentes techniques pour permettre à un ensemble de participants de collaborer et de coopérer malgré les contraintes imposées par les critères espace et temps. Voici quelques-unes des techniques utilisées:

- **WYSIWIS**: *"What You See Is What I See"*. De la même façon que deux personnes regardant, chez elles, la télévision peuvent assister à la même émission, l'informatique doit permettre à différentes personnes d'interagir et de communiquer dans un environnement WYSIWIS.
- **gestion d'agenda**: Les outils de gestion de groupe permettent de gérer un agenda commun et de planifier un projet. Cela facilite la gestion des emplois du temps, aussi bien pour les dirigeants que pour les dirigés.
- **multimédia**: Les collecticiels tirent avantage des capacités graphiques et sonores des ordinateurs pour améliorer les interfaces utilisateur dédiées à la coopération (vidéo conférence, icônes, ...).

Les outils de CSCW sont donc essentiellement orientés vers les aspects humains de la coopération (notification des changements, planification, gestion d'un groupe de personnes, techniques de communication, ...). Le contrôle de la concurrence garde cependant un rôle important au sein des collecticiels. Il est en effet indispensable de gérer les problèmes liés aux accès concurrents effectués sur un même objet par différents participants.

3.4.2 Contrôle de la Concurrence

Le contrôle de la concurrence réalisé dans les collecticiels se rapproche plus du contrôle de la cohérence des systèmes distribués (sérialisation des opérations concurrentes, respect de la causalité, synchronisation des copies) que du contrôle des interactions des systèmes transactionnels (critères de correction des exécutions). Nous pouvons cependant constater que les contraintes des collecticiels asynchrones sont les mêmes que les nôtres, à savoir: répartition des sites (distribution à large échelle), réplication des données (multiplicité des copies d'un même objet) et robustesse (tolérance aux défaillances, autonomie des participants). Les systèmes collaboratifs en temps réel sont en outre soumis à des contraintes de temps de réponse (délai entre l'invocation d'une opération et son exécution) et de temps de notification (délai entre l'exécution d'une opération sur un site et la propagation de ses effets sur les autres sites).

Les mécanismes mis en œuvre par les collecticiels pour contrôler la concurrence (architecture centralisée, répliquée ou distribuée) sont donc similaires à ceux développés dans les systèmes distribués ou les gestionnaires de configurations: verrouillage des objets, passage de jeton (ou "prise de tour"), détection des dépendances (conflits résolus par les utilisateurs). Par conséquent, contrairement aux systèmes transactionnels, il est difficile de garantir des propriétés sur les différentes tâches exécutées (consensus sur la valeur finale d'un objet pour tous les participants l'ayant modifié, aucun résultat final produit à partir de résultats intermédiaires, ...).

3.4.3 Exemples de Collecticiels

BSCW

L'objectif du projet BSCW (*"Basic Support for Cooperative Work"*) est d'utiliser les facilités offertes par le Web pour développer des services de collaboration multi-plateformes. Bien qu'il existe maintenant d'autres technologies dédiées au travail de groupe (Lotus Notes par exemple), celles-ci sont moins adaptées dès que l'on sort du cadre d'une entreprise et qu'il est nécessaire d'interopérer avec d'autres systèmes informatiques, bases de données ou applications. Dans un tel contexte le Web dispose de plusieurs avantages en tant que support pour le partage d'informations:

- Il existe des navigateurs Web pour toutes les plates-formes et systèmes informatiques courants (Unix, Windows, Mac, ...). L'accès aux informations est ainsi indépedant du système utilisé.
- L'interface utilisateur des navigateurs Web est simple et la présentation des informations est uniforme, quelle que soit le système utilisé.
- Les structures de données qui ne peuvent pas être affichées directement par le navigateur peuvent être confiées à des modules externes.
- Les navigateurs Web sont de plus en plus souvent inclus dans l'environnment standard des utilisateurs. La coopération au travers du Web ne nécessite donc aucune installation ou maintenance supplémentaire.

- Les entreprises disposant d'un serveur Web (présence sur Internet ou simplement un Intranet) sont de plus en plus fréquentes. Il s'agit d'une technologie familière.

Le projet BSCW propose de transformer le Web, actuellement utilisé en tant que source d'informations passive, en un outil de coopération actif. La base de ce projet est le système d'espace de travail partagé (*"BSCW Shared Workspace"*) [Ben95], une application qui, aux fonctions de navigation et de téléchargement d'informations offertes par le Web, ajoute des fonctions plus évoluées telles que la mise à jour de documents, la gestion de versions, la notification des changements, l'administration de groupes et de membres, ... Ces mécanismes dédiés à la collaboration et au partage d'informations sont accessibles au travers des navigateurs Web standards et sont ainsi disponibles quelles que soient les plates-formes système et réseau.

Wiki

Dans le cadre du groupe de travail Collecticiels du GDR-PRC I3, les interactions entre les différents participants sont réalisées (en dehors des réunions de travail) via le collecticiel Wiki[21], un outil d'écriture collaborative se positionnant comme une alternative aux forums de discussion, newsgroups et mailing lists.

La principale fonctionnalité d'un site Wiki est de permettre à tout utilisateur de créer des pages et d'éditer des pages. Chaque page possède un titre (son nom), une description et une catégorie qui permet de faciliter la navigation. Le serveur gère également la carte du site qui répertorie toutes les pages existantes, ainsi qu'un historique des changements récents. Enfin, pour chaque page, le serveur conserve un historique des versions successives de la page. Le serveur gère de façon aussi transparente que possible les conflits d'édition (lorsque deux utilisateurs modifient la même version d'une page en parallèle).

De même que pour BSCW, les utilisateurs accèdent à Wiki au travers d'un navigateur Web standard. Ce collecticiel peut ainsi être utilisé sur toutes les plates-formes système et réseau pour lesquelles il existe un navigateur Web.

Microsoft NetMeeting

La vidéo-conférence ne constitue que l'une des fonctionnalités offertes par Microsoft NetMeeting[22]. Ce collecticiel supporte également le partage de données (*"multipoint data conferencing"*) au sein de l'environnement Microsoft Windows pour communiquer et collaborer en temps réel avec d'autres utilisateurs (sur Internet ou sur un réseau local). NetMeeting permet de partager des applications Windows, d'échanger des informations au travers d'un presse-papiers partagé, de transférer des fichiers, de collaborer sur un tableau blanc partagé, ou encore de communiquer via un forum de discussion (*"chat"*).

- **partage d'applications** : Il est possible de partager une application s'exécutant sur une machine avec les autres participants de la conférence. Ceux-ci peuvent lire les mêmes données ou informations et voir les mêmes actions que la personne partageant l'application (par exemple: éditer un texte, se déplacer dans le texte). Cette dernière

21. Wiki: `http://wiki.lri.fr:8080/scoop/scoop.wiki`
22. NetMeeting: `http://www.microsoft.com/netmeeting/features/`

peut également choisir de collaborer avec les autres participants. Dans ce cas, chacun peut contrôler l'application à tour de rôle. A noter que seule la personne partageant l'application doit avoir installé cette application sur sa machine.

- **presse-papiers partagé** : Le presse-papiers partagé permet d'échanger son contenu avec d'autres personnes en utilisant les opérations couper/copier/coller habituelles des applications.
- **transfert de fichiers** : NetMeeting offre la possibilité d'envoyer un fichier à un participant particulier ou à tous les participants de la conférence. Ceux-ci peuvent accepter ou refuser la réception du fichier.
- **tableau blanc** : Il s'agit d'une application de dessin vectoriel multi-pages, multi-utilisateurs. Elle permet de tracer toute sorte de diagramme et de visualiser des informations graphiques avec les autres participants de la conférence. Cette fonctionnalité complète celle de partage d'applications en supportant la collaboration "ad hoc" au travers d'une zone de dessin commune.
- **forum de discussion** : Cette fonction permet de discuter en temps réel avec les autres participants, même en l'absence de support audio/vidéo. NetMeeting supporte également les apartés (communications privées entre deux participants).

3.4.4 Conclusion

Ainsi, le domaine CSCW est plus orienté vers les aspects humains de la coopération (notification des changements, gestion d'un groupe de personnes, techniques de communication, ...) que vers les aspects contrôle de la concurrence et correction des tâches exécutées. Les collecticiels ne sont donc pas adaptés à notre définition de la coopération dans le cadre des applications coopératives distribuées (à large échelle) et hétérogènes.

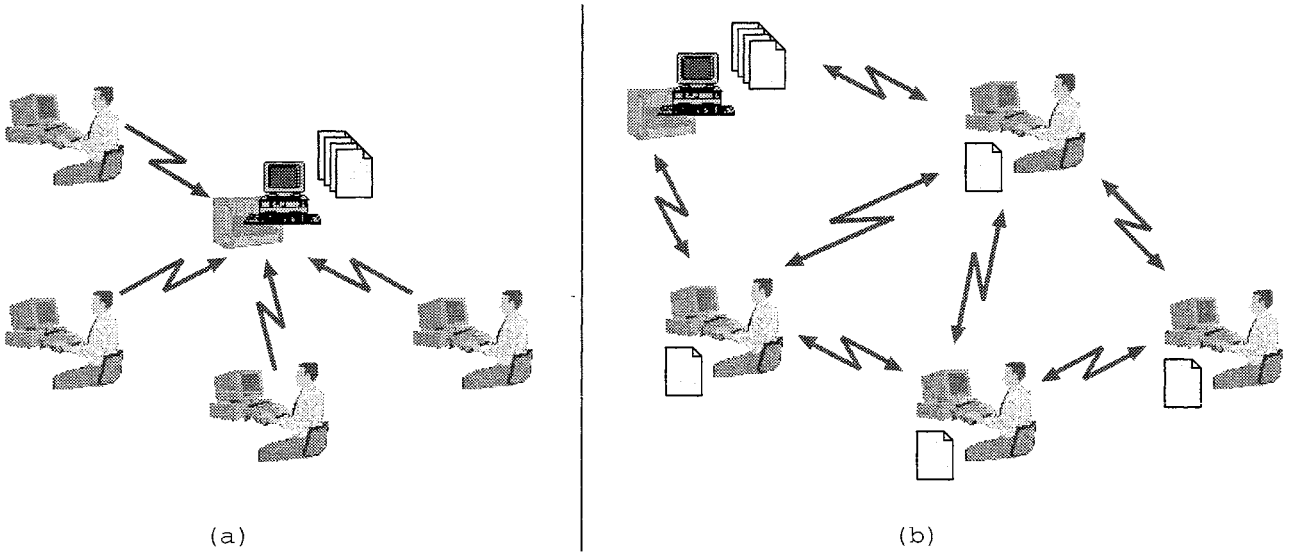
Certains mécanismes utilisés en CSCW peuvent cependant être considérés comme complémentaires de notre approche. C'est le cas par exemple des mécanismes de notification des changements au sein du système (création d'une nouvelle version de telle ressource par telle activité, ...). L'idée est de permettre aux activités réparties sur plusieurs sites de travailler de manière autonome tout en étant conscientes de ce qui se passe dans le groupe (*"group awareness"*).

3.5 Synthèse

Comme nous pouvons le constater, la plupart des systèmes distribués actuels sont basés sur une architecture client-serveur dans laquelle, bien que les différentes activités du système puissent s'exécuter sur des sites géographiquement distribués, les mécanismes de gestion de ces activités restent généralement centralisés sur le serveur (figure 3.12-a). La centralisation facilite la synchronisation et l'administration de l'activité globale puisque toutes les décisions sont prises au niveau de ce serveur qui possède une vue globale de tout le système. Toutefois, son principal inconvénient est la nécessité, pour les clients (i.e. les acteurs), d'être connectés en permanence au serveur, ce qui est bien évidemment

inacceptable dans le cas d'une entreprise-projet. A l'heure d'internet et de l'informatique mobile, il nous semble indispensable de pouvoir déployer un système distribué non plus sur un réseau local ou dédié, mais sur un réseau à large échelle, généralement peu fiable, dont certaines connexions pourraient être à faible bande passante (cas des liaisons longue distance) voire même occasionnelles (liaisons par modem).

Figure 3.12 Architecture Centralisée vs Architecture Distribuée



Il est alors nécessaire de rendre les différentes activités du système les plus autonomes possible, aussi bien vis-à-vis du réseau (qualité des connexions, coupures) que des autres activités (indisponibilité d'un serveur, ...). Pour une entreprise-projet, cela signifie que chaque partenaire, représenté par une activité du système, peut prendre **localement** et librement les décisions qu'il désire, alors que dans un système distribué classique il devrait attendre la réponse ou l'autorisation d'un site central pour exécuter une opération et voir les résultats de son action. Notre objectif est donc de parvenir à une architecture d'égal-à-égal, c'est-à-dire un réseau d'activités autonomes coopérant par échanges d'informations, ce réseau étant dépourvu de site central (figure 3.12-b).

Afin de contrôler la coopération entre les différentes activités, nous avons choisi de fonder notre approche sur la notion de transaction de manière à simplifier la programmation des applications en cachant au maximum la complexité introduite par l'exécution concurrente des différentes activités. Le contrôle de la concurrence devra en outre permettre l'utilisation de règles de coopération différentes selon les activités, ceci afin de prendre en compte l'hétérogénéïté des relations entre les activités sans avoir à construire un critère de correction unique.

Chapitre 4

Transactions Coopératives Distribuées

Table des matières

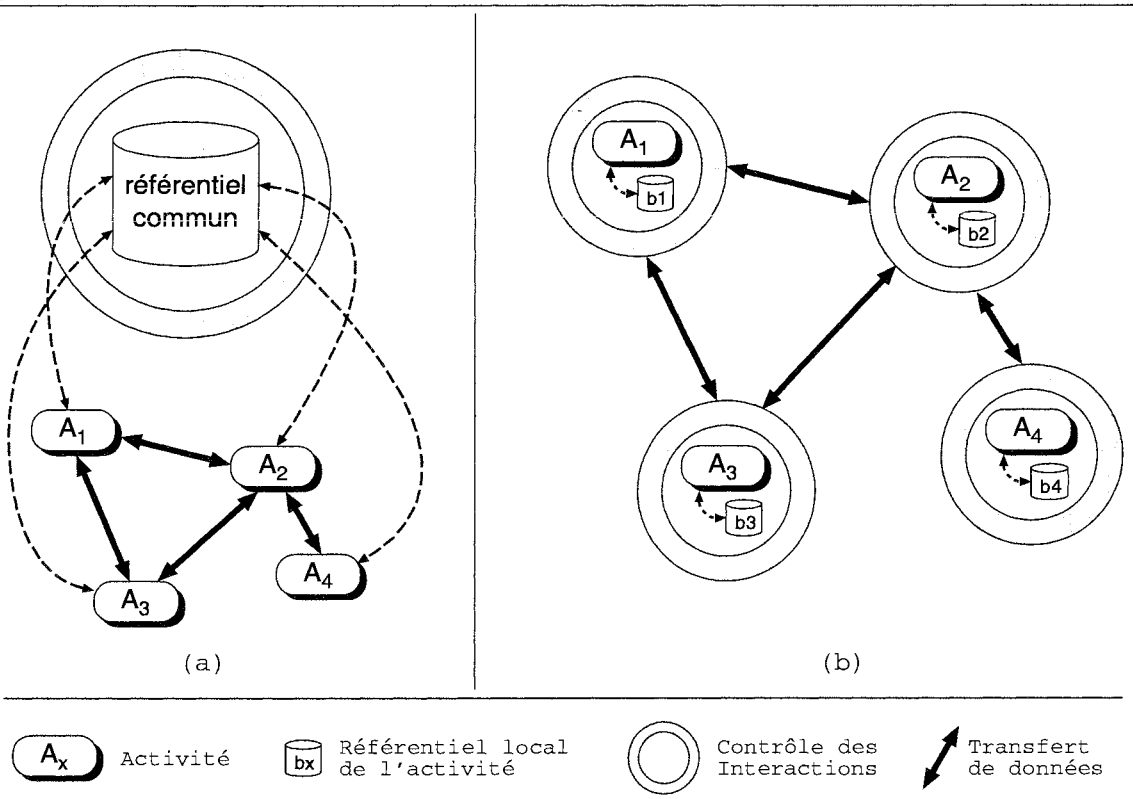
Les travaux présentés dans cette thèse ont donc pour objectif de définir un nouveau modèle de transactions avancé supportant à la fois la coopération entre les transactions, leur distribution et l'hétérogénéïté de leurs relations de coopération. Un certain nombre de travaux concernant le problème de la coopération ont déjà été réalisés dans le cadre du projet *COO* et ont donné lieu à la définition du modèle des *COO*-transactions et de la *COO*-sérialisabilité, un critère de correction supportant la coopération indirecte entre les transactions (échanges de résultats intermédiaires au cours de leur exécution). Toutefois, à la manière des systèmes transactionnels classiques, *COO* repose sur une base de données commune à toutes les transactions et dans laquelle sont stockées toutes les données partagées. Bien que cette base puisse ensuite être répartie, partitionnée ou répliquée, l'architecture logique du système reste centralisée (architecture de type client/serveur). Une couche "contrôle des interactions" enveloppe cette base de données et permet de contrôler les accès concurrents effectués par les différentes transactions du système (figure 4.1-a).

Partant des travaux réalisés dans *COO* en ce qui concerne la correction syntaxique des interactions coopératives, notre premier objectif est donc de proposer un nouveau modèle

de transactions avancé permettant une plus grande autonomie des transactions. Celui-ci est basé sur la définition des notions de référentiel local d'une transaction, d'histoire locale d'une transaction, et d'opération de transfert entre transactions.

Notre second objectif concerne la définition de nouveaux critères de correction qui soient eux-mêmes distribués. L'idée est en effet de permettre à chaque transaction du système de coordonner elle-même ses propres échanges de données avec les autres transactions (architecture d'égal-à-égal). Cela signifie que nous voulons assurer un contrôle global équivalent aux critères de correction classiques (sérialisabilité, *COO*-sérialisabilité) mais en ne nous basant que sur des contrôles locaux effectués par chacune des transactions (figure 4.1-b).

Figure 4.1 Contrôle Centralisé vs Contrôle Distribué



Dans les systèmes transactionnels classiques, y compris *COO*, il est nécessaire de définir le comportement global du système, i.e. de définir un seul et unique critère de correction que l'on appliquera à toutes les transactions. Cela signifie que l'on doit prévoir, au niveau du référentiel commun, toutes les interactions possibles entre toutes les activités du système. Or notre objectif est de rendre les transactions les plus autonomes possible, ce qui signifie en particulier qu'elles sont libres de choisir les règles de coopération qui contrôleront leurs échanges de données. La troisième partie de ce chapitre est donc consacrée à la prise en compte, au niveau du modèle de transactions, de l'hétérogénéïté des relations de coopération entre les différentes transactions. Outre la décentralisation du contrôle des interactions, nous proposons donc également de permettre aux transactions de négocier les règles (schémas de coopération) à respecter lors de leurs échanges.

Ainsi, nous avons organisé ce chapitre en trois parties correspondant à la construction, étape par étape, de notre modèle de transactions coopératives distribuées: présentation des concepts de base de notre modèle, définition de critères de correction "distribués", puis utilisation de plusieurs critères au sein d'un même système.

1. Nous commençons par formaliser notre modèle de transactions distribuées en termes de **bases locales** attachées aux transactions, d'**opérations de transfert** entre ces bases locales, et d'**histoire locale** d'une transaction. Il s'agit de permettre à chaque transaction de posséder sa propre copie des objets auxquels elle accède. Les transactions coopèreront ainsi par échanges de valeurs de leurs copies respectives. Ce sera justement grâce à ces opérations de transfert que nous pourrons synchroniser les histoires locales des différentes transactions.
2. La seconde partie de ce chapitre est consacrée à la définition de **critères de correction distribués**. Le rôle d'un critère de correction "classique" est de caractériser les exécutions dans lesquelles l'entrelacement des différentes opérations invoquées par les transactions est considéré comme correct. Un tel critère est défini sur l'histoire globale du système, c'est-à-dire en ayant une vue de toutes les opérations invoquées par toutes les transactions. Il s'agit donc d'un contrôle centralisé. A l'inverse, un critère de correction dit "distribué" est destiné à être vérifié par chaque nœud du système (un nœud représentant l'exécution d'une transaction) en n'ayant accès qu'aux informations journalisées localement par ce nœud (l'histoire locale de la transaction).
3. Toutefois, l'objectif de ce chapitre ne se limite pas uniquement à reformuler, certes de manière distribuée, des critères de correction existants tels que la sérialisabilité ou la *COO*-sérialisabilité. En définissant des critères de correction distribués nous sommes en fait passés d'un **contrôle des interactions** (accès concurrents à un référentiel commun) à un **contrôle des échanges** entre une transaction donnée et ses transactions partenaires (échanges de données explicites entre les transactions). La troisième partie de ce chapitre développe cette idée en définissant les notions de **schéma de coopération** et de **négociation** d'un tel schéma entre deux transactions, l'objectif final étant de permettre à chaque couple de transactions désirant partager certains objets de fixer elles-même les règles de coopération à respecter lors de leurs échanges. Les transactions de notre modèle sont ainsi organisées en réseau, les nœuds réprésentant les transactions et les arcs réprésentant les schémas de coopération négociés entre les transactions. Par conséquent, il n'existe plus **un seul** critère de correction contrôlant tous les échanges mais **plusieurs** critères de correction distribués devant cohabiter au sein du même système.

Le fait que notre modèle de transactions coopératives distribuées soit résolument orienté vers le contrôle des échanges de données explicites entre transactions plutôt que vers le contrôle des interactions (accès concurrents) au niveau d'un référentiel commun constitue une différence essentielle par rapport aux modèles de transactions classiques. Ceci nous apporte à la fois une **plus grande autonomie des transactions** (critères distribués ou "décentralisés") et une **meilleure adéquation du contrôle de la coopération** aux relations existant entre les différents partenaires (hétérogénéïté des critères).

4.1 Modèle de Transactions Distribuées

Pour définir notre modèle de transactions, la première étape consiste à formaliser l'aspect distribution de ces transactions. Il s'agit en fait d'attribuer un référentiel local à chaque activité du système, cette activité s'exécutant dans une transaction longue. Les objets manipulés par les transactions ne seront donc plus stockés dans un seul et unique référentiel, mais dans les référentiels locaux associés aux différentes activités. Celles-ci possèderont donc chacune leur propre copie des objets qu'elles manipulent. Par conséquent, contrairement aux objets des modèles de transactions classiques, un même "objet logique" de notre modèle aura donc plusieurs "instances" qu'il nous faudra distinguer. Un second point est la définition de l'histoire locale d'une transaction, i.e. la liste des événements du système qui seront journalisés pour une transaction.

Puisque chaque transaction possèdera sa propre copie des objets qu'elle manipule, les échanges de données entre transactions ne seront donc plus **implicites** (via des accès concurrents à un référentiel commun) mais **explicites** via l'utilisation d'**opérations de transfert** entre les référentiels respectifs de ces transactions. Le rôle de ces opérations sera de synchroniser, entre plusieurs transactions, les valeurs des différentes instances d'un même objet logique. Ces notions de bases/histoires locales et d'opérations de transfert nous permettront par la suite de définir des **règles de coopération locales** pour le partage des objets entre deux transactions, i.e. des règles qui ne sont définies que sur des **informations locales** à ces deux transactions.

Nous étudierons ensuite l'impact de ces changements sur deux critères de correction particuliers: la sérialisabilité et la *COO*-sérialisabilité. De tels critères seront dits "globaux" puisque basés sur l'histoire globale du système (ex: détection de cycles de dépendances entre activités). Pour chacune d'eux, nous reformulerons leur définition axiomatique en utilisant les notations nouvellement introduites de manière à obtenir un critère de correction "local", c'est-à-dire se basant uniquement sur les histoires locales des transactions. L'idée sous-jacente est que le fait d'assurer la correction de l'exécution de chaque transaction par rapport aux transactions avec lesquelles elle interagit directement doit permettre d'assurer, implicitement, la correction de l'exécution globale du système.

4.1.1 Bases Locales et Histoires Locales

Base Locale d'une Transaction

Notre premier objectif est donc de représenter explicitement, au niveau du modèle de transactions, le fait qu'il n'existe plus un seul et unique référentiel commun à toutes les transactions du système, mais que celles-ci possèdent chacune leur propre base de données locale. Comme nous l'avons précédemment expliqué, cela signifie que si un objet est partagé (utilisations simultanées) entre plusieurs transactions, chacune d'entre elles possèdera sa propre copie de cet objet dans sa base locale. Par conséquent, un même **objet logique** aura donc plusieurs **instances** (une instance par transaction qui le manipule) qu'il nous faudra distinguer au niveau de notre modèle. Il ne s'agit pas simplement de réplicas (synchronisés automatiquement par le système) mais bien de copies indépendantes dont les transactions s'échangeront explicitement les valeurs. En outre, ces différentes instances

pourront éventuellement avoir des valeurs différentes à un instant donné lors de l'exécution.

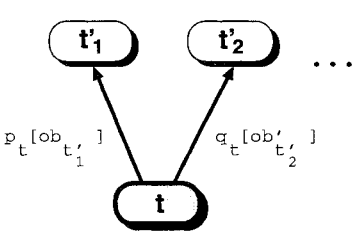
Lorsqu'une transaction t exécutera une opération op sur un objet ob (ce qui est noté $op_t[ob]$ dans le formalisme ACTA [Chr94]), il sera donc nécessaire de préciser sur quelle instance de cet objet ob sera effectuée cette opération op. Nous utiliserons la notation $op_t[ob_{t'}]$ pour représenter le fait que l'opération $op_t[ob]$ est réalisée sur l'objet ob de la transaction t'. Ce que nous appelons la **base locale** d'une transaction t sera donc l'ensemble des objets ob_t.

Histoire Locale d'une Transaction

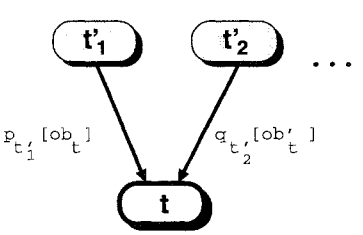
Comme nous pouvons le constater, cette notation nous permet de représenter le fait qu'une transaction puisse effectuer des opérations sur des objets se trouvant non seulement dans sa propre base locale mais également dans les bases locales d'autres transactions (exemple: $op_{t_i}[ob_{t_j}]$). Ce sont précisément ces opérations qui nous permettront par la suite de représenter les **interactions** entre transactions, et en particulier les transferts de données entre transactions.

En utilisant cette nouvelle notation nous pouvons maintenant définir la notion d'histoire locale d'une transaction t. Il s'agit en fait d'identifier les événements du système qui concernent la transaction t. En ACTA [Chr94], ces événements sont de deux types: les événements dits significatifs (`Begin`, `Commit`, `Abort`, ...) et les événements relatifs aux objets (invocations d'opérations sur les objets). Ce sont principalement ces derniers qui nous intéressent puisqu'ils vont nous permettre de contrôler la visibilité des objets entre les différentes transactions via la notion de "vue d'une transaction". En ACTA, la vue d'une transaction t, notée $View_t$, indique les objets et les états de ces objets visibles par la transaction t à un instant donné. En d'autres termes, la vue d'une transaction détermine quelles sont les opérations dont les effets (sur les objets) sont visibles par cette transaction. En outre, une vue étant une projection de l'histoire globale courante (notée H_{ct}), l'ordre partiel entre les opérations est conservé. En ce qui concerne notre modèle de transactions, les opérations dont les effets sont visibles par une transaction t sont:

- les opérations invoquées par la transaction t elle-même, quels que soient les objets sur lesquels ces opérations ont été invoquées: $\{p_t[ob_{t'}] \in H_{ct}\}$



- les opérations invoquées par des transactions t' (différentes de t) sur des objets se trouvant dans la base locale de la transaction t: $\{p_{t'}[ob_t] \in H_{ct}\}$



Par conséquent, la vue d'une transaction t de notre modèle est définie de la manière suivante: $View_t = \{p_t[ob_{t'}] \in H_{ct}\} \cup \{p_{t'}[ob_t] \in H_{ct}\}$. Ceci determine quelles sont, parmi

toutes les opérations de l'histoire globale, celles dont la transaction t a connaissance, et donc ainsi son **histoire locale** que nous noterons $H_{ct/t}$.

Conflits entre Opérations

Une histoire H (locale ou globale) est donc une succession d'invocations d'opération (*"object events"*). Nous noterons $H^{(ob)}$ la projection de l'histoire H par rapport à un objet *ob* particulier. Cette histoire $H^{(ob)} = p_1 \circ p_2 \circ \ldots \circ p_n$ indique à la fois l'ordre des opérations invoquées sur l'objet *ob* (i.e. l'opération p_i précède l'opération p_{i+1}, ce que nous noterons également $p_i \rightarrow p_{i+1}$) ainsi que la composition fonctionnelle des opérations. Ceci signifie que l'état s de l'objet *ob* dans l'histoire $H^{(ob)}$ est l'état produit par l'invocation successive (cf. relation d'ordre $\rightarrow$) des différentes opérations de $H^{(ob)}$ à partir d'un état initial s_0 (i.e. $s = state(s_0, H^{(ob)})$). Pour simplifier, nous noterons cet état $state(H^{(ob)})$.

En ACTA, une transaction accède et manipule des objets de la base en invoquant des opérations spécifiques aux différents objets. Chaque opération retourne une valeur et produit un état. Si s est l'état d'un objet, $return(s, p)$ renvoie le résultat produit par l'opération p. L'état de cet objet produit par l'exécution de l'opération p est noté $state(s, p)$. Deux opérations (également vues comme des appels de fonctions) sont alors dites conflictuelles si leurs effets sur l'état d'un objet ou si leurs valeurs de retour ne sont pas indépendantes de leur ordre d'exécution (définition 4.1).

DÉFINITION 4.1
Deux opérations p et q sont dites *conflictuelles* pour un état $H^{(ob)}$ (ce qui sera noté $conflict(H^{(ob)}, p, q)$ ou plus simplement $conflict(p[ob], q[ob])$) ssi

$$\begin{aligned}(state(H^{(ob)} \circ p,\ q) &\neq state(H^{(ob)} \circ q,\ p)) \vee \\ (return(H^{(ob)},\ q) &\neq return(H^{(ob)} \circ p,\ q)) \vee \\ (return(H^{(ob)},\ p) &\neq return(H^{(ob)} \circ q,\ p))\end{aligned}$$

Deux opérations qui ne sont pas en conflit sont dites *compatibles*.

Puisque les changements d'état des objets sont observés via les valeurs de retour des opérations, nous pouvons définir une relation de dépendance entre deux opérations conflictuelles (définition 4.2).

DÉFINITION 4.2
En cas de conflit entre deux opérations (i.e. le prédicat $conflict(H^{(ob)}, p, q)$ vaut vrai), $return\_value\_independent(H^{(ob)}, p, q)$ est vrai si la valeur de retour de q est independante du fait que p précède q ou non, i.e., $return(H^{(ob)} \circ p,\ q) = return(H^{(ob)},\ q)$; sinon q est *"return-value dependent"* de p (noté $return\_value\_dependent(H^{(ob)}, p, q)$).

Comme nous pouvons le constater, ces deux définitions (introduites par le formalisme ACTA) ne sont pas, du point de vue de notre modèle de transactions, exprimées par

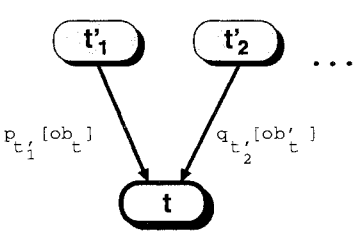
rapport aux instances "physiques" des objets (ex ob_t), mais par rapport aux objets dits "logiques" (ex: ob).

En fait, de la même façon que $conflict(p[ob], q[ob])$ est une abbréviation de $conflict(p_{t_i}[ob], q_{t_j}[ob])$[23], nous pouvons considérer, dans un premier temps, que la notation $conflict(p_{t_i}[ob], q_{t_j}[ob])$ est une abbréviation de $conflict(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}])$[24]. Cela signifie que pour le moment, deux opérations ne peuvent être en conflit que si elles sont invoquées sur la même instance d'un objet logique. Par la suite, lorsque nous définirons la notion d'opérations de transfert, nous préciserons la signification de $conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$, i.e. d'un conflit entre deux opérations p_{t_i} et q_{t_j} invoquées sur des instances différentes d'un même objet logique ob. Idem pour le prédicat $return\_value\_independent(p_{t_i}[ob], q_{t_j}[ob])$.

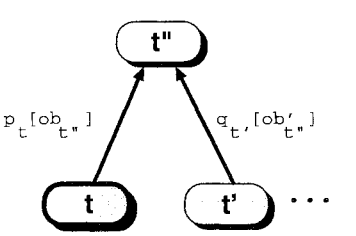
Dans un premier temps, notre objectif est donc de pouvoir déterminer, lorsqu'une transaction t invoque une opération p sur un objet $ob_{t'}$, quelles sont les opérations, invoquées par d'autres transactions, avec lesquelles l'opération $p_t[ob_{t'}]$ peut être en conflit (ensemble noté $ConflictSet_t$ dans le formalisme ACTA). Les ensembles $View_t$ et $ConflictSet_t$ définissent ainsi les événements pouvant être invoqués. Plus précisément, les préconditions de ces événements[25] sont évaluées par rapport à ces deux ensembles. Si ses préconditions sont vérifiées, le nouvel événement est effectivement invoqué puis journalisé dans les histoires locales des différentes transactions concernées par l'occurence de cet événement (cf. définition de la vue d'une transaction). Dans le cas contraire (i.e. au moins une des préconditions n'est pas vérifiée), l'invocation de cet événement est refusée.

En ce qui concerne notre modèle de transactions, les opérations pouvant être en conflit avec des opérations invoquées par une transaction t sont définies ci-dessous. Le prédicat $Inprogress(p)$ représente le fait que l'opération p est en cours d'exécution, i.e. qu'elle n'a pas encore été ni validée ni annulée.

- les opérations effectuées par des transactions t' $(t' \neq t)$ sur des objets ob_t de la transaction t: $\{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\}$



- les opérations exécutées par des transactions t' $(t' \neq t)$ sur des objets d'une tierce transaction t'' $(t'' \neq t)$ sur lesquels la transaction t a précédemment invoqué des opérations:



$\{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\ q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\}$

23. Les deux opérations p et q (invoquées respectivement par deux transactions t_i et t_j quelconques) sont conflictuelles au niveau de l'objet ob.

24. Les deux opérations p_{t_i} et q_{t_j} sont conflictuelles quelle que soit l'instance ob_{t_k} de l'objet logique ob.

25. Pour chaque événement, ses préconditions sont dérivées de la définition axiomatique de la transaction qui l'invoque.

Par conséquent, pour une transaction t de notre modèle, l'ensemble des opérations pouvant être en conflit avec une des opérations de t est défini de la manière suivante:

$$ConflictSet_t = \{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\} \cup \{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\ q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\}$$

Remarque

Si l'on impose que tous les objets du système se trouvent dans une seule base de données commune à toutes les transactions, alors les définitions des ensembles $View_t$ et $ConflictSet_t$ deviennent équivalentes à celles fournies dans la définition axiomatique des transactions atomiques, à savoir:

- $View_t = H_{ct}$ où H_{ct} représente l'histoire globale courante du système
- $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t,\ Inprogress(p_{t'}[ob])\}$

Par conséquent, les notions de base locale (différentes instances ob_t d'un même objet logique ob) et d'histoire locale (chaque transaction n'a qu'une vision locale du système) ne sont qu'une extension apportée au formalisme ACTA. Elles vont nous permettre de modéliser les échanges de données **explicites** entre les transactions.

4.1.2 Opérations de Transfert

Chaque transaction de notre modèle possède dorénavant sa propre base de données locale dans laquelle elle stocke une copie physique (une instance) de tous les objets qu'elle partage. Par conséquent, si deux transactions t_i et t_j partagent un même objet logique ob, chacune d'elles aura sa propre instance (respectivement ob_{t_i} et ob_{t_j}) dans sa base locale. De quelle manière les transactions t_i et t_j vont-elles s'échanger les valeurs de cet objet ob?

En fait, nous avons vu, lors de la présentation de notre nouvelle notation pour désigner les objets, qu'une transaction n'était pas limitée à l'invocation d'opérations uniquement sur "ses" objets. Une transaction t peut en effet exécuter une opération p sur un objet ob se trouvant dans la base locale d'une autre transaction t': cette opération est noté $p_t[ob_{t'}]$. Ce sont donc de telles opérations dites "de transfert" qui vont permettre aux activités de coopérer par échanges de données.

Le principal écueil consiste bien évidemment à faire le lien entre ces opérations. En effet, dans un modèle de transactions classique, si deux transactions t_i et t_j coopèrent via un objet ob, elles vont invoquer des opérations $p_{t_i}[ob]$ et $q_{t_j}[ob]$ (une seule instance de l'objet ob). Les interactions entre les transactions t_i et t_j sont donc clairement identifiées. Par contre, dans le cas de notre modèle de transactions, deux transactions t_i et t_j peuvent interagir en invoquant des opérations $p_{t_i}[ob_{t_{k_1}}]$ et $q_{t_j}[ob_{t_{k_n}}]$ sur des instances éventuellement différentes d'un objet ob. Notre objectif est donc maintenant d'expliquer de quelle manière nous allons "reconstruire" ce lien entre les opérations $p_{t_i}[ob_{t_{k_1}}]$ et $q_{t_j}[ob_{t_{k_n}}]$.

Conflits entre Opérations

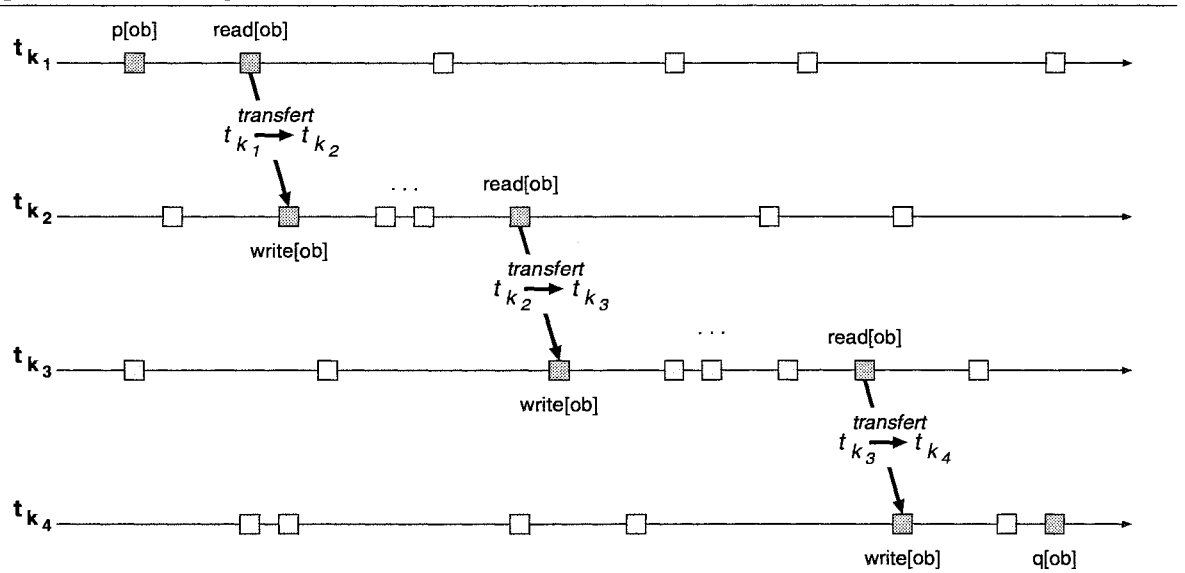
Habituellement, une interaction entre des transactions t_i et t_j au niveau d'un objet ob est représentée de la façon suivante: $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge conflict(p_{t_i}[ob], q_{t_j}[ob])$. Ceci

signifie que les transactions t_i et t_j ont invoqué des opérations $p_{t_i}[ob]$ et $q_{t_j}[ob]$ sur un même objet ob, que l'opération $p_{t_i}[ob]$ précède l'opération $q_{t_j}[ob]$ (selon l'ordre partiel défini par l'histoire globale H_{ct}), et que ces deux opérations sont conflictuelles (définition 4.1). En utilisant notre nouvelle notation pour désigner les objets, quelle est maintenant la signification de l'expression $(p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \wedge conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$? Il existe en fait deux possibilités:

- soit il s'agit de la même instance ob_{t_k} de l'objet logique ob, ce qui nous ramène alors au cas $(p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]) \wedge conflict(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}])$
- soit il s'agit de deux instances différentes du même objet logique ob, i.e. $(p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \wedge conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$ avec $t_{k_1} \neq t_{k_n}$

En ce qui concerne le premier cas de figure, cela ne pose pas de problème particulier puisque nous pouvons utiliser les définitions classiques de la relation $\rightarrow$ de précédence entre opérations et du prédicat *conflict*. C'est évidemment le deuxième cas de figure qui nous intéresse, et plus particulièrement la signification de $conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$. Intuitivement, cela signifie que *"la valeur de l'objet ob de la transaction t_{k_n} (i.e. $ob_{t_{k_n}}$) dépend de la valeur de l'objet ob de la transaction t_{k_1} (i.e. $ob_{t_{k_1}}$)"*. En d'autres termes, nous avons un enchaînement d'opérations tel que présenté figure 4.2 qui nous a permis de "propager", de proche en proche, la valeur de l'objet $ob_{t_{k_1}}$ vers la transaction t_{k_n}.

Figure 4.2 Propagation des Modifications



La figure 4.2 illustre l'exécution des transactions t_{k_1}, t_{k_2}, t_{k_3} et t_{k_4} sous la forme d'un diagramme temporel des messages[26]. Chaque ligne horizontale représente l'exécution d'une transaction (le temps évoluant de la gauche vers la droite), c'est-à-dire l'histoire locale de cette transaction. Une flèche reliant deux transactions représente un échange de données entre ces deux transactions , avec l'événement d'*émission* (ex: opération de lecture) à la

26. Issus du domaine des systèmes distribués, ces diagrammes temporels représentent les échanges (réalisés par envois de messages) entre les différents processus communiquant via un réseau.

base de la flèche et l'événement de *réception* (ex: opération d'écriture) correspondant à la tête de la flèche. En utilisant cette représentation graphique, il est facile de vérifier si deux événements quelconques du système sont liés **causalement**: s'il est possible de tracer un chemin d'un événement à l'autre en procédant (par transitivité) de gauche à droite le long des lignes horizontales (*"transaction-order"* des événements dans les histoires locales) et dans le sens des flèches (relation *"read-from"* entre les transactions), alors ils sont liés; sinon ils sont dits indépendants.

Cette relation de dépendance indirecte (via des opérations dites de transfert) entre les objets $ob_{t_{k_1}}$ et $ob_{t_{k_n}}$ sera notée $\xrightarrow{dep}$. Ainsi, sur la séquence d'opérations présentée ci-dessous, nous avons $\forall i \in \{2..n\}$ $(ob_{t_{k_{i-1}}} \xrightarrow{dep} ob_{t_{k_i}})$. Par transitivité, nous en déduisons que $ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_n}}$.

$$p_{t_i}[ob_{t_{k_1}}] \rightarrow \underbrace{q_{t_{l_1}}[ob_{t_{k_1}}] \rightarrow p_{t_{l_1}}[ob_{t_{k_2}}]}_{ob_{t_{k_1}} \rightsquigarrow ob_{t_{k_2}}} \rightarrow \underbrace{q_{t_{l_2}}[ob_{t_{k_2}}] \rightarrow p_{t_{l_2}}[ob_{t_{k_3}}]}_{ob_{t_{k_2}} \rightsquigarrow ob_{t_{k_3}}} \rightarrow \ldots \rightarrow \underbrace{q_{t_{l_{n-1}}}[ob_{t_{k_{n-1}}}] \rightarrow p_{t_{l_{n-1}}}[ob_{t_{k_n}}]}_{ob_{t_{k_{n-1}}} \rightsquigarrow ob_{t_{k_n}}} \rightarrow q_{t_j}[ob_{t_{k_n}}]$$

DÉFINITION 4.3 (OPÉRATION DE TRANSFERT)
Soit t_l une transaction.
Soient $ob_{t_{k_i}}$ et $ob_{t_{k_j}}$ deux instances d'un même objet logique ob.
Soit $Read^{(ob)}$ l'ensemble [a] des opérations q telles que $State(H^{(ob)} \circ q) = State(H^{(ob)})$.
Soit $Write^{(ob)}$ l'ensemble [b] des opérations p telles que $State(H^{(ob)} \circ p) \neq State(H^{(ob)})$.

Une opération de transfert est alors définie comme étant un couple d'opérations, i.e. $transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}] = (q_{t_l}[ob_{t_{k_i}}], p_{t_l}[ob_{t_{k_i}}])$, telles que $(q \in Read^{(ob)}) \wedge (p \in Write^{(ob)}) \wedge (q_{t_l}[ob] \rightarrow p_{t_l}[ob])$.

[a] $Read^{(ob)}$ représente l'ensemble des opérations permettant d'accéder, sans la modifier, à la valeur de l'objet ob.

[b] $Write^{(ob)}$ représente l'ensemble des opérations permettant de modifier la valeur de l'objet ob.

Nous pouvons remarquer que pour tout objet ob et pour toutes opérations $q \in Read^{(ob)}$ et $p \in Write^{(ob)}$, nous avons $return(H^{(ob)}, q) \neq return(H^{(ob)} \circ p, q)$, et par conséquent $conflict(q_{t_l}[ob], p_{t_l}[ob])$.

De façon plus formelle, nous pouvons définir la notion d'opération de transfert de la manière suivante (définition 4.3). Une opération de transfert est une **opération virtuelle** qui est en fait la succession de deux opérations invoquées par une même transaction t_l, l'une pour accéder à la valeur de l'objet $ob_{t_{k_i}}$ (ex: $read_{t_l}[ob_{t_{k_i}}]$), l'autre pour modifier la valeur de l'objet $ob_{t_{k_j}}$ (ex: $write_{t_l}[ob_{t_{k_i}}]$), de manière à transférer la valeur de l'instance $ob_{t_{k_i}}$ de l'objet ob vers une autre de ses instances $ob_{t_{k_j}}$. Une telle opération sera notée $transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}]$. Ce sont ces opérations qui vont nous permettre de "synchroniser" les histoires locales des différentes transactions. En effet, les relations *"read-from"* entre les transactions seront représentées par le fait qu'une opération $p_{t_l}[ob_{t_k}]$ sera journalisée à la fois par la transaction t_l (celle qui invoque l'opération) et par la transaction t_k (celle qui possède l'objet sur lequel est invoquée l'opération).

Nous pouvons ensuite définir la relation de dépendance sémantique $\xrightarrow{dep}$ entre objets (définition 4.4). Cette relation nous permet de savoir, pour un objet *ob* donné, que la valeur de telle instance de cet objet a été produite à partir de la valeur de telle autre instance. Comme nous pouvons le constater, le graphe des dépendances $\xrightarrow{dep}$ d'un objet peut en fait être assimilé à son graphe de versions (en supposant que l'on n'ait qu'une seule version par transaction).

DÉFINITION 4.4 (RELATION DE DÉPENDANCE SÉMANTIQUE $\xrightarrow{dep}$)
La relation de dépendance sémantique $\xrightarrow{dep}$ entre les différentes instances d'un même objet logique est définie de la manière suivante:

- **Réflexivité:** $\forall ob\ \forall t_k \quad (ob_{t_k} \xrightarrow{dep} ob_{t_k})$
- **Opération de transfert:** $\forall ob\ \forall t_l, t_{k_i}, t_{k_j} \quad transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}] \Rightarrow (ob_{t_{k_i}} \xrightarrow{dep} ob_{t_{k_j}})$
- **Transitivité:** $\forall ob\ \forall t_{k_1}, t_{k_2}, t_{k_3} \quad (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_2}}) \wedge (ob_{t_{k_2}} \xrightarrow{dep} ob_{t_{k_3}}) \Rightarrow (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_3}})$

En utilisant cette relation $\xrightarrow{dep}$, nous pouvons alors définir la notion de conflit entre deux opérations invoquées sur des instances différentes d'un même objet logique (définition 4.5).

DÉFINITION 4.5 (CONFLIT ENTRE OPÉRATIONS)
Soient t_i et t_j deux transactions.
Soient p et q deux opérations.
Soient $ob_{t_{k_i}}$ et $ob_{t_{k_j}}$ deux instances d'un même objet logique *ob*.

$$conflict(p_{t_i}[ob_{t_{k_i}}], q_{t_j}[ob_{t_{k_j}}]) \Leftrightarrow conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (ob_{t_{k_i}} \xrightarrow{dep} ob_{t_{k_j}})$$

Dans la séquence d'opérations que nous avons donnée précédemment, une séquence $q_{t_{l_i}}[ob_{t_{k_i}}] \rightarrow p_{t_{l_i}}[ob_{t_{k_{i+1}}}]$ correspond donc à ce que nous avons appelé une opération de transfert (notée $transfer_{t_{l_i}}[ob_{t_{k_i}}, ob_{t_{k_{i+1}}}]$), effectuée par la transaction t_{l_i}, de la valeur de l'objet $ob_{t_{k_i}}$ vers l'objet $ob_{t_{k_{i+1}}}$, i.e. quelque chose du style $read_{t_{l_i}}[ob_{t_{k_i}}] \rightarrow write_{t_{l_i}}[ob_{t_{k_{i+1}}}]$.

En fait, dans les sections suivantes, nous n'utiliserons cette notion de conflit entre deux opérations $p_{t_i}[ob_{t_{k_i}}]$ et $q_{t_j}[ob_{t_{k_j}}]$ invoquées sur des instances $ob_{t_{k_i}}$ et $ob_{t_{k_j}}$ différentes d'un même objet logique *ob* que pour démontrer que les propriétés garanties par les critères de correction "distribués" sont les mêmes que celles garanties par les critères de correction "centralisés". En pratique, puisque les critères de correction distribués que nous aurons définis ne seront basés que sur des informations locales aux transactions (i.e. sur les histoires locales des transactions), ceux-ci utiliseront simplement la notion de conflit classique entre les opérations d'une même histoire locale (opérations invoquées sur une même instance d'un objet logique).

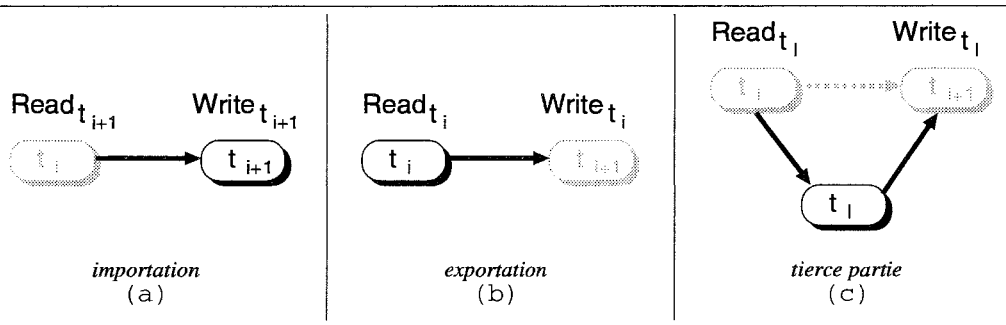
Importation et Exportation d'Objets

Nous pouvons maintenant définir les opérations d'importation et d'exportation d'un objet par une transaction depuis/vers une autre transaction comme étant des opérations de transfert particulières:

- **Importation**: Si $t_{l_i} = t_{k_{i+1}}$, alors $transfer_{t_{k_{i+1}}}[ob_{t_{k_i}}, ob_{t_{k_{i+1}}}] \equiv import_{t_{k_{i+1}}}[ob_{t_{k_i}}]$, i.e. la transaction $t_{k_{i+1}}$ importe (dans son objet $ob_{t_{k_{i+1}}}$) la valeur de l'objet ob de la transaction t_{k_i} (i.e. de l'objet $ob_{t_{k_i}}$).
- **Exportation**: Si $t_{l_i} = t_{k_i}$, alors $transfer_{t_{k_i}}[ob_{t_{k_i}}, ob_{t_{k_{i+1}}}] \equiv export_{t_{k_i}}[ob_{t_{k_{i+1}}}]$, i.e. la transaction t_{k_i} exporte la valeur de l'objet ob (i.e. $ob_{t_{k_i}}$) vers la transaction $t_{k_{i+1}}$ (i.e. vers l'objet $ob_{t_{k_{i+1}}}$).

Intuitivement, nous pouvons interpréter ceci de la façon suivante: en cas d'importation *"c'est le consommateur qui va chercher, de gré, l'information chez le producteur"* (figure 4.3-a), alors qu'en cas d'exportation *"c'est le producteur qui diffuse, de force, l'information chez le consommateur"* (figure 4.3-b).

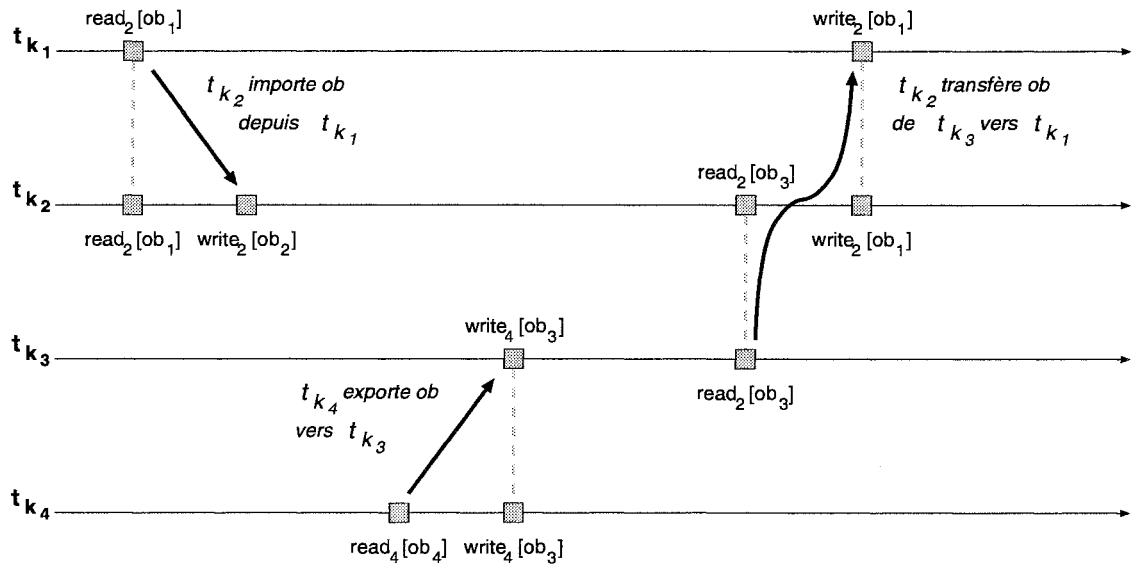
Figure 4.3 Différentes Possibilités de Transfert



A noter que la définition d'une opération de transfert (définition 4.3) autorise également un troisième cas de figure: *"un tiers récupère l'information chez le producteur puis la transmet au consommateur"* (figure 4.3-c). Il s'agit là d'une forme de coopération bien particulière entre le producteur et le consommateur puisque ni l'un ni l'autre n'ont un rôle actif dans cet échange. Bien qu'étant d'un intérêt restreint dans le cadre des entreprises virtuelles, il nous semble toutefois utile de préserver cette possibilité au niveau du modèle de transactions. Ultérieurement, nous pourrons ainsi mettre en œuvre des mécanismes d'échange plus spécifiques (réplication automatique de certaines données, abonnement à des listes de diffusion, communications de groupe, ...).

Ces trois façons de procéder pour échanger des données entre les transactions sont également illustrées figure 4.4. Ce diagramme temporel représente les trois types d'opérations de transfert en termes d'opérations de lecture et d'écriture journalisées dans les histoires locales des différentes transactions. Comme nous pouvons le constater, une opération $p_{t_i}[ob_{t_j}]$ est journalisée à la fois par la transaction t_i (celle qui invoque l'opération) et par la transaction t_j (celle qui possède l'instance de l'objet ob sur laquelle est invoquée l'opération). Un telle opération joue le rôle de "point de synchronisation" entre les histoires locales des transactions t_i et t_j.

Figure 4.4 Opérations de Transfert et Histoires Locales



Conclusion

Ces différentes définitions nous permettent maintenant d'expliquer de quelle manière nous allons traduire l'expression $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge conflict(p_{t_i}[ob], q_{t_j}[ob])$ utilisée dans les modèles de transactions classiques pour représenter une interaction entre les transactions t_i et t_j. En effet, puisque les opérations p et q peuvent avoir été invoquées sur des instances différentes de l'objet ob nous devons utiliser les notations $p_{t_i}[ob_{t_{k_1}}]$ et $q_{t_j}[ob_{t_{k_n}}]$. Ceci fait l'objet du lemme 4.1. Nous pourrons ainsi réutiliser, sur notre modèle de transactions, les propriétés définies pour des modèles plus classiques.

LEMME 4.1
Soient t_i et t_j deux transactions distinctes $(t_i \neq t_j)$.

interaction entre les transactions t_i et t_j dans un modèle de transactions classique:	$\Leftrightarrow$	interaction entre les transactions t_i et t_j dans notre modèle de transactions:
$(p_{t_i}[ob] \rightarrow q_{t_j}[ob])$		$(p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}])$
$\wedge \quad conflict(p_{t_i}[ob], q_{t_j}[ob])$		$\wedge \quad conflict(p_{t_i}[ob], q_{t_j}[ob])$
		$\wedge \quad (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_n}})$

Preuve :

Ce lemme est une conséquence directe de la définition de la notion de conflit entre deux opérations invoquées sur des instances différentes d'un même objet logique (définition 4.5), i.e. de $conflict(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}])$.

□

Remarque

Si l'on impose que tous les objets du système soient stockés dans une seule et unique base de données commune à toutes les transactions (ce qui est le cas dans les modèles de transactions classiques), on retrouve alors la notation habituelle $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge conflict(p_{t_i}[ob], q_{t_j}[ob])$.

4.1.3 Axiomes de Base de notre Modèle

Nous pouvons dès à présent énoncer les axiomes de base de notre modèle de transactions. Puisque nous utilisons le formalisme ACTA, nous devons tout d'abord définir les événements significatifs de nos transactions (`Begin`, `Commit`, `Abort` ...) ainsi que les axiomes fondamentaux relatifs à ces événements. Viennent ensuite les axiomes décrivant plutôt l'organisation des transactions, tels que la définition des événements dont une transaction a connaissance (i.e. son histoire locale, notée $View_t$).

Axiomes Fondamentaux des Transactions

Outre les invocations d'opérations sur les objets, les transactions peuvent également invoquer des primitives de gestion. Par exemple, un modèle simple pourrait utiliser les trois primitives suivantes pour gérer les transactions: `Begin`, `Commit` et `Abort`. La définition de primitives plus spécifiques ainsi que de leur signification dépend du modèle de transactions considéré. L'invocation d'une telle primitive de gestion des transactions est appelée un **événement significatif**.

Un modèle de transaction devra indiquer quels sont les événements significatifs pouvant être invoqués par chacune des transactions de ce modèle. Nous noterons SE_t l'ensemble des événements significatifs valides pour la transaction t. Parmi ces événements, nous pouvons distinguer deux sous-ensembles particuliers: les **événements d'initialisation** (ensemble $IE_t \subset SE_t$) qui peuvent être invoqués pour démarrer l'exécution de la transaction t (ex: `Begin`), et les **événements de terminaison** (ensemble $TE_t \subset SE_t$) qui peuvent être invoqués pour terminer l'exécution de la transaction t (ex: `Commit`, `Abort`). Une transaction est dite *"in progress"* si elle a été démarrée par l'invocation d'un événement d'initialisation et qu'elle n'a pas encore exécuté un des événements de terminaison qui lui sont associés. Une transaction **termine** quand elle exécute un événement de terminaison.

Les invocations d'événements significatifs sont soumises à certaines règles, notamment en ce qui concerne les événements d'initialisation et de terminaison. Il s'agit des **axiomes fondamentaux des transactions** de la définition 4.6. L'axiome I interdit qu'une transaction puisse être démarrée par deux événements différents. L'axiome II établit qu'une transaction terminée doit avoir été demarrée. L'axiome III interdit qu'une transaction puisse être terminée par deux événements de terminaison différents. L'axiome IV signifie que seules les transactions en cours d'exécution peuvent invoquer des opérations sur des objets. Finalement, l'axiome V représente le fait que les invocations d'opérations ne peuvent être effectuées que sur des objets de transactions ayant été initialisées. Cet axiome autorise cependant l'invocation d'opérations sur les objets d'une transaction terminée.

DÉFINITION 4.6 (AXIOMES FONDAMENTAUX DES TRANSACTIONS)
Soit t une transaction. Soit H^t la projection de l'histoire H par rapport à t. Soient α, β, γ et δ des événements significatifs.

$p_t[ob_{t'}]$ représente l'événement correspondant à l'invocation effectuée par la transaction t de l'opération p sur l'objet ob de la transaction t'.

$$(I) \quad \forall \alpha \in IE_t \ (\alpha \in H^t) \ \Rightarrow \ \nexists \beta \in IE_t \ (\alpha \rightarrow \beta)$$
$$(II) \quad \forall \delta \in TE_t \ \exists \alpha \in IE_t \ (\delta \in H^t) \ \Rightarrow \ (\alpha \rightarrow \delta)$$
$$(III) \quad \forall \gamma \in TE_t \ (\gamma \in H^t) \ \Rightarrow \ \nexists \delta \in TE_t \ (\gamma \rightarrow \delta)$$
$$(IV) \quad \forall ob_{t'} \ \forall p \ (p_t[ob_{t'}] \in H) \ \Rightarrow \ ((\exists \alpha \in IE_t \ (\alpha \rightarrow p_t[ob_{t'}])) \wedge (\exists \gamma \in TE_t \ (p_t[ob_{t'}] \rightarrow \gamma)))$$
$$(V) \quad \forall ob_{t'} \ \forall p \ (p_t[ob_{t'}] \in H) \ \Rightarrow \ (\exists \alpha \in IE_{t'} \ (\alpha \rightarrow p_t[ob_{t'}]))$$

Modèle de Transactions

Les axiomes de base de notre nouveau modèle de transactions sont donnés à la définition 4.7. Ceux-ci définissent les événements significatifs, d'initialisation et de terminaison pour nos transactions, ainsi que les ensembles $View_t$ et $ConflictSet_t$ d'une transaction t.

DÉFINITION 4.7 (TRANSACTIONS DISTRIBUÉES)
Soit t une transaction.

(4.1) $SE_t = \{\texttt{Begin}, \texttt{Commit}, \texttt{Abort}\}$

(4.2) $IE_t = \{\texttt{Begin}\}$

(4.3) $TE_t = \{\texttt{Commit}, \texttt{Abort}\}$

(4.4) t satisfait les axiomes fondamentaux I à V

(4.5) $View_t = \{p_t[ob_{t'}] \in H_{ct}\} \ \cup \ \{p_{t'}[ob_t] \in H_{ct}\}$

(4.6) $ConflictSet_t = \{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\}$
$\cup \ \{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \ \wedge \ (\exists q \ q_t[ob_{t''}] \in H_{ct}) \ \wedge \ Inprogress(p_{t'}[ob_{t''}])\}$

L'axiome 4.1 définit trois événements significatifs associés aux transactions de notre modèle: `Begin`, `Commit` et `Abort`. L'axiome 4.2 précise que `Begin` est l'événement de début de ces transactions. L'axiome 4.3 indique que `Commit` et `Abort` sont les deux événements de terminaison associés aux transactions. En outre, les transactions de notre modèle doivent également respecter les axiomes fondamentaux énoncés précédemment (axiome 4.4). Finalement, l'axiome 4.5 définit la vue d'une transaction (i.e. son histoire locale) et l'axiome 4.6 indique, pour une transaction t, les opérations (effectuées par d'autres transactions) avec lesquelles des conflits doivent être envisagés lorsque t invoque une nouvelle opération.

4.2 Critères de Correction Distribués

Nous venons de présenter un modèle de transactions distribuées dans lequel les différentes transactions possèdent chacune leur propre référentiel local (ainsi qu'une histoire locale) et coopèrent par le biais d'échanges de données explicites entre leurs référentiels locaux. Dans un modèle de transactions classique, les critères de correction (sérialisabilité, *COO*-sérialisabilité, ...) permettent de coordonner les accès concurrents aux objets du référentiel commun. Dans le cas de notre modèle, un critère de correction aura plutôt pour rôle de coordonner les échanges de données entre les différentes transactions.

L'objectif de cette seconde partie est donc de définir de quelle manière nous allons contrôler les interactions entre les transactions. Nous allons pour cela nous baser sur deux critères de correction connus: la sérialisabilité et la *COO*-sérialisabilité. Toutefois, il ne s'agit pas uniquement de "traduire" ces critères pour introduire la notion de base locale. Nous voulons également que ce contrôle des interactions soit distribué, i.e. que chaque transaction soit capable d'assurer, **localement**, la correction de ses propres échanges. Cela signifie donc que les contraintes ne devront être exprimées qu'à partir d'informations locales aux transactions.

Nous obtiendrons ainsi un nouveau critère de correction, la *DisCOO*-sérialisabilité (respectivement la *D*-sérialisabilité), qui est une version distribuée de la *COO*-sérialisabilité (respectivement de la sérialisabilité). Bien entendu, ce critère de correction local, s'il est vérifié au niveau de l'histoire locale de chacune des transactions, doit **implicitement** assurer les mêmes propriétés que la *COO*-sérialisabilité (respectivement la sérialisabilité) au niveau de l'histoire globale.

4.2.1 Sérialisabilité Distribuée

Pour s'assurer que les résultats produits par les différentes transactions sont cohérents (ex: pas de mise à jour perdue, pas de lecture impropre, ...), le système doit garantir que l'exécution concourante de transactions (produisant individuellement des résultats cohérents) est correcte. Cela signifie que le fait d'exécuter simultanément plusieurs transactions ne doit pas introduire d'incohérence au niveau des résultats produits. Les règles indiquant quelles sont les exécutions correctes définissent ce que l'on appelle un critère de correction.

Le critère habituellement utilisé dans les bases de données est la sérialisabilité. Celui-ci assure que des transactions concourantes s'exécutent sans interférence, i.e. comme si elles s'exécutaient en série. Après avoir rappelé la définition de la sérialisabilité dans le cadre des modèles de transactions classiques, nous présenterons une version distribuée de ce critère. En effet, notre objectif est de définir un certain nombre de propriétés locales (i.e. qui seront vérifiées sur chaque transaction en ne se basant que sur l'histoire locale de cette transaction) qui garantissent, de manière implicite, la sérialisabilité au niveau global.

Sérialisabilité Classique

Dans les systèmes transactionnels traditionnels, les transactions vérifient généralement les propriétés de sérialisabilité et de *"failure atomicity"*. Ces propriétés assurent que des

transactions concourantes s'exécutent sans interférence comme si elles s'exécutaient en série et que pour une transaction donnée, soit toutes ses opérations sont exécutées soit aucune ne l'est. La sérialisabilité est assurée en empêchant les transactions validées de former des cycles dans les relations $\mathcal{C}$ de dépendance entre les transactions, ceci afin d'éviter que deux transactions soient interdépendantes. Comme nous pouvons le constater dans la définition 4.8, cette propriété est définie sur l'histoire globale du système puisqu'il est nécessaire de "voir" toutes les opérations effectuées par les diverses transactions pour détecter les cycles.

DÉFINITION 4.8 (SÉRIALISABILITÉ)
Soit $\mathcal{C}$ une relation binaire sur un ensemble de transactions T.
Soit H l'histoire des événements relatifs aux transactions de T.

$\forall t_i, t_j \in T \ , \ t_i \neq t_j \ , \ (t_i \ \mathcal{C} \ t_j)$ ssi $\exists ob \ \exists p, q \ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \ \wedge \ (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$

Soit $\mathcal{C}^*$ la fermeture transitive de $\mathcal{C}$, i.e: $(t_i \ \mathcal{C}^* \ t_j)$ ssi $(\ (t_i \ \mathcal{C} \ t_j) \ \vee \ \exists t_k \ (t_i \ \mathcal{C} \ t_k \ \wedge \ t_k \ \mathcal{C}^* \ t_j) \ )$

H est dite *sérialisable* ssi $\forall t \in T \ \neg(t \ \mathcal{C}^* \ t)$

En ce qui concerne la propriété de *"failure atomicity"*, cela signifie que pour une transaction donnée, soit toutes ses opérations sont validées soit aucune ne l'est. Dans cette définition, la clause "toutes" est représentée par la première condition qui indique que si une opération invoquée par une transaction t est validée, alors toutes les opérations invoquées par t doivent être validées. La clause "aucune" est représentée par la seconde condition qui indique que si une opération invoquée par une transaction t est annulée, alors toutes les opérations invoquées par t doivent être annulées.

DÉFINITION 4.9 (FAILURE ATOMICITY)
Une transaction t est dite *"failure atomic"* si

1. $\exists ob \ \exists p \ (Commit[p_t[ob]] \in H) \ \Rightarrow \ \forall ob' \ \forall q \ (\ (q_t[ob'] \in H) \ \Rightarrow \ (Commit[q_t[ob']] \in H) \ )$
2. $\exists ob \ \exists p \ (Abort[p_t[ob]] \in H) \ \Rightarrow \ \forall ob' \ \forall q \ (\ (q_t[ob'] \in H) \ \Rightarrow \ (Abort[q_t[ob']] \in H) \ )$

Pour qu'un objet se comporte "correctement" celui-ci doit garantir que lorsqu'une opération est annulée, toutes les opérations qui en sont *"return_value_dependent"* sont également annulées. Cela garantit le comportement correct des objets en cas d'erreur, en supposant que les effets des opérations sur les objets sont immédiats. Comme dans le cas des transactions, le comportement sérialisable des objets est assuré en empêchant les transactions validées de former des cycles dans les relations $\mathcal{C}$.

Version Distribuée de la Sérialisabilité

La sérialisabilité, telle qu'elle a été énoncée précédemment (définition 4.8), est donc définie sur l'histoire globale du système, i.e. en termes de détection de cycles de dépendances entre les transactions au moment de leur validation. Afin de "distribuer" ce critère

DÉFINITION 4.10 (CORRECTION D'UN OBJET)
Un objet *ob* se comporte *correctement* ssi

$$\forall t_i, t_j \ , \ t_i \neq t_j \ , \ \forall p, q \ (\ return\_value\_dep(p,q) \ \wedge \ (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \ ) \ \Rightarrow$$
$$(\ (Abort[p_{t_i}[ob]] \in H^{(ob)}) \ \Rightarrow \ (Abort[q_{t_j}[ob]] \in H^{(ob)}) \ )$$

Un objet *ob* se comporte de façon *sérialisable* ssi

$$\forall t \in T \ \forall p \ (Commit[p_t[ob]] \in H^{(ob)}) \ \Rightarrow \ \neg(t \ \mathcal{C}^* \ t)$$

Un objet *ob* est dit *atomique* s'il se comporte à la fois *correctement* et de façon *sérialisable*.

de correction, notre objectif est de définir un certain nombre de propriétés locales (i.e. définies sur les histoires locales des transactions) qui, en étant assurées (localement) par chacune des transactions, garantissent de manière implicite la sérialisabilité au niveau global.

Toutefois, garantir la sérialisabilité telle qu'elle a été définie peut, en pratique, poser certains problèmes en cas d'échec d'une transaction. Prenons par exemple les deux exécutions représentées ci-dessous. Elles sont toutes les deux sérialisables (pas de cycle de dépendances). Dans celle de gauche, supposons que la transaction t_0 soit annulée au lieu d'être validée. Puisque la transaction t_1 dépend de la transaction t_0 (t_1 a lu la valeur de l'objet x produite par t_0), elle devrait également être annulée[27]. Or celle-ci a déjà été validée. Nous serions donc obligé de remettre en cause une transaction validée, ce qui est contraire à la propriété de durabilité des transactions ACID.

t_0	t_1
$r_0[x]$	
$w_0[x]$	
	$r_1[x]$
	$w_1[y]$
	`Commit`$_1$
`Commit`$_0$	

t_0	t_1
$r_0[x]$	
$w_0[x]$	
	$r_1[x]$
	$w_1[y]$
`Commit`$_0$	
	`Commit`$_1$

L'idée est de retarder la validation d'une transaction (celle de t_1 en l'occurence) jusqu'à ce que toutes les transactions dont elle dépend (transaction t_0 sur notre exemple) soient elles-même validées. C'est pour cette raison que les systèmes transactionnels utilisent généralement une version restreinte de la sérialisabilité qui impose un ordre sur les événements `Commit` en fonction de leurs dépendances (*"Order Preserving Serializability"*). Ainsi, l'exécution de gauche sera considérée comme incorrecte vis-à-vis de ce critère bien qu'elle soit sérialisable. Par contre, l'exécution (sérialisable) de droite sera acceptée car l'ordre de validation est respecté.

27. Plus exactement, si l'objet x se comporte correctement (définition 4.10), le fait d'annuler l'opération $w_0[x]$ nous impose d'annuler toutes les opérations qui en sont *"return_value_dependent"*, i.e. en particulier $r_1[x]$. Puisque la transaction t_1 est supposée être *"failure atomic"* (définition 4.9), nous devons également annuler toutes les opérations effectuées par t_1, ce qui signifie que nous devons également annuler la transaction t_1.

Par conséquent, au moment de valider une transaction t_j, au lieu de tester son implication dans un cycle de dépendances avec d'autres transactions, i.e. $(\texttt{Commit}_{t_j} \in H) \Rightarrow \neg(t_j\ \mathcal{C}^*\ t_j)$, nous nous assurerons plutôt que toutes les transactions dont la transaction t_j est dépendante sont elles-mêmes validées (axiome 4.11). Sinon, la validation de la transaction t_j est différée.

DÉFINITION 4.11 (TRANSACTIONS DISTRIBUÉES (SUITE))
Soit t une transaction.

(4.7) $\exists ob_{t_k}\ \exists p\ (\texttt{Commit}_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \Rightarrow (\texttt{Commit}_{t_j} \in H)$

(4.8) $(\texttt{Commit}_{t_j} \in H) \Rightarrow \forall ob_{t_k}\ \forall p\ (\ (p_{t_j}[ob_{t_k}] \in H) \Rightarrow (Commit_{t_j}[p_{t_j}[ob_{t_k}]] \in H)\ )$

(4.9) $\exists ob_{t_k}\ \exists p\ (Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \Rightarrow (\texttt{Abort}_{t_j} \in H)$

(4.10) $(\texttt{Abort}_{t_j} \in H) \Rightarrow \forall ob_{t_k}\ \forall p\ (\ (p_{t_j}[ob_{t_k}] \in H) \Rightarrow (Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H)\ )$

(4.11) $(Commit_{t_j}[q_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow$
$conflict(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \wedge (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]) \Rightarrow$
$(Commit_{t_i}[p_{t_i}[ob_{t_k}]] \rightarrow Commit_{t_j}[q_{t_j}[ob_{t_k}]])$

En d'autres termes, l'axiome 4.11 précise qu'une opération ne pourra être validée que si toutes les opérations la précédant et avec lesquelles elle était en conflit ont elles-même été validées. Cet axiome, utilisé conjointement avec les quatre axiomes 4.7 à 4.10 (axiomes permettant de garantir qu'une transaction est *"failure atomic"*), nous assure qu'une transaction ne pourra être validée que si toutes les transactions dont elle dépend sont elles-même validées.

Comme nous pouvons le constater, l'axiome 4.11 utilise uniquement des informations locales à la transaction t_k. En effet, tous les événements utilisés par cet axiome auront été journalisés dans l'histoire locale de t_k (notée H_{t_k}) puisqu'ils concernent tous l'objet ob_{t_k}. Toute exécution respectant l'axiome 4.11 sera dite D-sérialisable.

Nous allons maintenant démontrer qu'en assurant cette propriété locale (axiome 4.11) au niveau de chaque transaction, l'exécution D-sérialisable obtenue est également une exécution sérialisable (lemme 4.3). Pour cela, nous allons raisonner par contra-posée, i.e. nous allons tout d'abord démontrer que si une exécution n'est pas sérialisable, alors elle n'est pas D-sérialisable (lemme 4.2).

LEMME 4.2 ($\neg SR \Rightarrow \neg DSR$)
Si une exécution n'est pas sérialisable, alors elle n'est pas D-sérialisable.

Preuve :

Si une exécution n'est pas sérialisable, cela signifie que l'axiome de validation $(\texttt{Commit}_t \in H) \Rightarrow \neg(t\ \mathcal{C}^*\ t)$ a été violé, i.e. qu'il existe une transaction t_i qui a été validée alors qu'elle était impliquée dans un cycle de dépendances, ce qui se traduit par: $\exists t_i\ (\texttt{Commit}_{t_i} \in H) \wedge (t_i\ \mathcal{C}^*\ t_i)$.

1. D'après la définition de la relation binaire $\mathcal{C}$ (déf. 4.8), nous avons $(t_i\ \mathcal{C}\ t_j)$ si et seulement si $\exists ob\ \exists p, q\ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$. D'où,

en utilisant le lemme 4.1, nous obtenons que

$$(t_i \; \mathcal{C} \; t_j) \Leftrightarrow \exists ob \; \exists p, q \; \exists k_1, k_n \begin{cases} (p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \\ conflict(p_{t_i}[ob], q_{t_j}[ob]) \\ (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_n}}) \end{cases}$$

Nous nous retrouvons donc dans le cas où nous avons une suite d'opérations du style:

$$p_{t_i}[ob_{t_{k_1}}] \rightarrow \underbrace{q_{t_{l_1}}[ob_{t_{k_1}}] \rightarrow p_{t_{l_1}}[ob_{t_{k_2}}]}_{\text{transaction } t_{l_1}} \rightarrow \underbrace{q_{t_{l_2}}[ob_{t_{k_2}}] \rightarrow p_{t_{l_2}}[ob_{t_{k_3}}]}_{\text{transaction } t_{l_2}} \rightarrow \ldots \rightarrow \underbrace{q_{t_{l_{n-1}}}[ob_{t_{k_{n-1}}}] \rightarrow p_{t_{l_{n-1}}}[ob_{t_{k_n}}]}_{\text{transaction } t_{l_{n-1}}} \rightarrow q_{t_j}[ob_{t_{k_n}}]$$

Par conséquent, en procédant de la droite vers la gauche et en utilisant alternativement l'axiome 4.11 (introduit pour remplacer la détection de cycles) puis les axiomes 4.7 et 4.8 présentés ci-dessus, nous obtenons:

$$(t_i \; \mathcal{C} \; t_j) \; \wedge \; (\texttt{Commit}_{t_j} \in H) \; \Rightarrow \; (\texttt{Commit}_{t_i} \rightarrow \texttt{Commit}_{t_j})$$

2. Revenons maintenant à la preuve du lemme 4.2. Si une exécution n'est pas sérialisable, cela signifie qu'il existe une transaction t_i telle que $(\texttt{Commit}_{t_i} \in H) \; \wedge \; (t_i \; \mathcal{C}^* \; t_i)$.
   De par la définition de la fermeture transitive de la relation $\mathcal{C}$ (déf. 4.8) et en utilisant la propriété démontrée dans la première partie de la preuve, nous obtenons que si une exécution n'est pas sérialisable, alors il existe une transaction t_i telle que $(\texttt{Commit}_{t_i} \rightarrow \texttt{Commit}_{t_i})$.
   Or ceci est impossible (cf. axiome fondamental III de la définition 4.6) étant donné que `Commit` est un événement de terminaison.

Par conséquent cette exécution, qui n'était pas sérialisable, n'est pas *D*-sérialisable.

□

LEMME 4.3 (*DSR* $\Rightarrow$ *SR*)
Toute exécution *D*-sérialisable est une exécution sérialisable.

Preuve :

C'est le corollaire du lemme 4.2.

□

Bilan

Cette nouvelle définition de la sérialisabilité, la *D*-sérialisabilité, nous permet donc de garantir une plus grande autonomie aux transactions de notre modèle. En effet, pour déterminer si une transaction peut être validée (`Commit`), seules des informations "locales" à cette transaction sont utilisées, i.e. des informations journalisées soit dans sa propre histoire locale, soit dans les histoires locales des transactions avec lesquelles elle a échangé des données. Il n'est donc plus nécessaire d'avoir une vue globale du système ni de déterminer la présence de cycles de dépendances entre les transactions pour garantir qu'une exécution est (*order preserving*) sérialisable.

4.2.2 *DisCOO*: Version Distribuée du Critère *COO*

Puisqu'une transaction t_i est capable d'exécuter une opération *op* sur un objet *ob* d'une autre transaction t_j (opération notée $op_{t_i}[ob_{t_j}]$), cela signifie que les deux transactions t_i et t_j pourront "interagir" via l'objet ob_{t_j} au cours de leur exécution. Il nous faut donc coordonner ces interactions (concurrence d'accès). Toutefois, un critère de correction tel que la sérialisabilité (ou la *D*-sérialisabilité) est trop restrictif puisqu'il n'autorise pas les exécutions dans lesquelles deux transactions seraient interdépendantes[28].

Afin de permettre aux transactions de coopérer par le biais d'échanges de valeurs intermédiaires (i.e. de résultats produits en cours d'exécution et potentiellement inconsistants vis-à-vis du système), un nouveau critère de correction, *COO*, a été défini dans [Mol96]. En brisant la propriété d'isolation des transactions, ce critère accepte un certain nombre d'exécutions non sérialisables. Il permet ainsi aux transactions de coopérer selon trois paradigmes: client/serveur, rédacteur/relecteur, écriture coopérative. Toutefois, bien que les différentes transactions du système puissent être géographiquement distribuées, le contrôle de leur interactions reste centralisé puisque ce critère est défini sur l'histoire globale du système. Notre objectif est donc de fournir une version distribuée[29] du critère de correction *COO*.

Nous rappelons tout d'abord ci-dessous les principaux axiomes des *COO*-transactions (définition 4.12). Nous définissons ensuite un ensemble d'axiomes équivalent (i.e. assurant les mêmes propriétés au niveau global), mais en fondant ces axiomes uniquement sur les histoires locales des transactions.

Critère de Correction *COO*

La *COO*-sérialisabilité a été définie dans [Mol96] pour permettre à un ensemble de transactions de s'exécuter de manière coopérative en relâchant la propriété d'isolation. Cela signifie que les différentes transactions peuvent coopérer en s'échangeant, au cours de leur exécution, des données par l'intermédiaire d'un référentiel commun. Dans *COO* nous distinguons donc deux types de résultats: les résultats **intermédiaires** produits par une transaction en cours d'exécution (ces résultats préliminaires sont potentiellement inconsistants et sujets à de nouvelle modifications), et les résultats **finaux** produits par une transaction à la fin de son exécution. Ce critère de correction peut être vu comme une extension de la sérialisabilité classique pour supporter la notion de résultat intermédiaire. Intuitivement, une exécution coopérative sera considérée comme étant correcte si elle respecte les règles de synchronisation suivantes:

1. Si une transaction produit un résultat intermédiaire, alors elle doit produire le résultat final correspondant lorsqu'elle est validée.

28. Dans le cas de la sérialisabilité, les interdépendances sont explicitement prohibées par la définition 4.8. En ce qui concerne la *D*-sérialisabilité, nous avons démontré dans la preuve du lemme 4.2 que si une transaction t_j dépend d'une transaction t_i (i.e. $t_i \; C \; t_j$), alors le `Commit` de t_i doit précéder celui de t_j (i.e. $\texttt{Commit}_{t_i} \rightarrow \texttt{Commit}_{t_j}$). Par conséquent, en cas d'interdépendance entre les deux transactions, ni l'une ni l'autre ne pourra être validée.

29. *DisCOO* est l'abréviation de "Distributed *COO*".

2. Si une transaction accède à un résultat intemédiaire d'une autre transaction, alors elle doit lire le résultat final correspondant avant de pouvoir produire ses propres résultats finaux. Lorsqu'une transaction accède à un résultat intemédiaire d'une autre transaction, elle devient dépendante de cette transaction. Lorsqu'elle lit le résultat final correspondant, cette dépendance est levée.
3. En cas de cycle dans le graphe des dépendances entre transactions (échanges bidirectionnels entre deux transactions par exemple), les transactions impliquées dans le cycle sont groupées. Les transactions d'un même groupe (noté T_{coo}) ont alors l'obligation de terminer leur exécution en même temps de manière indivisible.

DÉFINITION 4.12 (PRINCIPAUX AXIOMES DES *COO*-TRANSACTIONS)

1. Pour une séquence d'opérations donnée, seule la dernière opération a pour obligation d'être validée. Seules les opérations de la sous-trace utile d'une transaction sont donc prises en compte.
   $(\texttt{Commit}_t \in H) \Rightarrow$
   $\forall ob\ \forall Q_t[ob]\ \exists q \in Q_t\ (\forall q' \in Q_t\ ,\ q' \neq q\ ,\ q'_t[ob] \rightarrow q_t[ob])\ \wedge\ (Commit_t[q_t[ob]] \in H)$
2. Toute opération validée doit dépendre (si dépendance il y a) d'une opération validée.
   $(Commit_t[q_t[ob]] \in H) \Rightarrow$
   $\exists p\ (\ return\_value\_dep(p_{t'}[ob], q_t[ob])\ \wedge\ (p_{t'}[ob] \rightarrow q_t[ob])\ ) \Rightarrow$
   $(Commit_{t'}[p_{t'}[ob]] \in H)$
3. Toutes les transactions d'un groupe doivent converger vers un état final unique.
   $(\texttt{Commit}_t \in H)\ \wedge\ t \in T_{coo} \Rightarrow$
   $\forall ob\ \forall q\ (State(H^{(ob)}) \neq State(H^{(ob)} \circ q))\ \wedge\ (Commit_{t_i}[q_{t_i}[ob]] \in H)\ \wedge\ (t_i \neq t) \Rightarrow$
   $\exists p\ (\ return\_value\_dep(q_{t_i}[ob], p_t[ob])\ \wedge\ (State(H^{(ob)}) = State(H^{(ob)} \circ p))\ \wedge$
   $(q_{t_i}[ob] \rightarrow p_t[ob])\ \wedge\ (Commit_t[p_t[ob]] \in H)$
   $)$
4. Dans un groupe de transactions, soit toutes les transactions sont validées (`Commit`), soit aucune ne l'est (`Abort`).
   $\forall t_i, t_j \in T_{coo}\ ,\ t_i \neq t_j\ ,\ (t_i\ \mathcal{SCD}\ t_j)\ \wedge\ (t_i\ \mathcal{AD}\ t_j)$
5. Si une opération est annulée, alors toutes les opérations qui en dépendent sont également annulées.
   $(Abort_t[p_t[ob]] \in H) \Rightarrow$
   $(\ return\_value\_dep(p_t[ob], q_{t_i}[ob])\ \wedge\ (p_t[ob] \rightarrow q_{t_i}[ob])\ ) \Rightarrow (Abort_{t_i}[q_{t_i}[ob]] \in H)$

Plus formellement, la définition axiomatique des *COO*-transactions est exprimée en ACTA de la manière suivante (la définition 4.12 ne représente que les axiomes permettant de garantir la *COO*-sérialisabilité). Cette définition repose sur deux idées de base: la notion de sous-trace "utile" d'une transaction et la notion de groupe de transactions. Nous considérons tout d'abord qu'une opération $q_t[ob]$ est généralement invoquée plusieurs fois par la transaction t sur l'objet ob au cours de son exécution (écriture des différents résultats intermédiaires de l'objet ob produits par la transaction t par exemple). Ces différentes occurences successives de l'opération $q_t[ob]$ forment une séquence d'opérations que l'on

notera $Q_t[ob]$. Le principe de *COO* est donc de ne considérer que la dernière occurence de chaque séquence d'opérations (la sous-trace utile) pour déterminer si l'exécution d'une transaction est correcte. Le second principe est de grouper les transactions impliquées dans un même cycle de dépendances (cf. règle $n°2$). Ce groupe de transactions est alors considéré comme une seule transaction virtuelle vis-à-vis des autres transactions du système. Au sein d'un tel groupe, les transactions doivent respecter la règle de synchronisation $n°3$. De plus amples détails concernant *COO* sont disponibles dans [Can98c, Mol97, Mol96, Can96, God96].

L'axiome 1 indique que l'on ne valide pas toutes les opérations effectuées par une transaction mais, pour chaque série d'opérations sur un objet, uniquement la dernière occurence de cette série. L'axiome 2 précise simplement que si une opération est validée et si elle est dépendante d'une autre opération, alors cette autre opération doit également être validée. Si plusieurs transactions sont groupées, la convergence de ce groupe de transactions vers un état final unique est assuré par l'axiome 3. Celui-ci impose que pour chaque objet, la dernière valeur écrite par une transaction du groupe soit relue par toutes les autres transactions de ce groupe. En outre, l'axiome 4 indique que si plusieurs transactions sont groupées, soit toutes les transactions du groupe sont validées (`Commit`), soit aucune ne l'est (`Abort`). Finalement, en cas d'échec d'une transaction (`Abort`), toutes les opérations qu'elle aura invoquées seront annulées, ainsi que toutes les opérations qui en dépendent (axiome 5).

DÉFINITION 4.13 (CRITÈRE DE CORRECTION *COO* (*COO*-SÉRIALISABILITÉ))
Soit H l'histoire des événements relatifs aux transactions d'un ensemble T.
Soit $\mathcal{C}_{cs}$ une relation binaire sur les transactions de T définie par:

$$\forall t_i, t_j \in T \ , \ t_i \neq t_j \ ,$$

$$(t_i \ \mathcal{C}_{cs} \ t_j) \quad \text{ssi} \quad \exists ob \ \exists p, q \begin{cases} & (conflict(p_{t_i}[ob], q_{t_j}[ob]) \\ \wedge & (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \\ \wedge & (Commit_{t_i}[p_{t_i}[ob]] \ \wedge \ Commit_{t_j}[q_{t_j}[ob]]) \end{cases}$$

Soit T_{coo} un groupe de transactions coopératives, $T_{coo} \subseteq T$.
Soit $\mathcal{C}_{coo}$ une relation binaire sur les transactions de T définie par:

$$\forall t_i, t_j, t_k \in T \ , \ t_i \neq t_j \ , \ t_i \neq t_k \ , \ t_j \neq t_k \ , \quad \forall T_{coo} \subseteq T \ ,$$

$$(t_i \ \mathcal{C}_{coo} \ t_j) \quad \text{ssi} \quad \begin{cases} & (t_i \notin T_{coo} \ \wedge \ t_j \notin T_{coo} \ \wedge \ (t_i \ \mathcal{C}_{cs} \ t_j)) \\ \vee & (t_i \notin T_{coo} \ \wedge \ t_j \in T_{coo} \ \wedge \ t_k \in T_{coo} \ \wedge \ (t_i \ \mathcal{C}_{cs} \ t_k)) \\ \vee & (t_i \in T_{coo} \ \wedge \ t_j \notin T_{coo} \ \wedge \ t_k \in T_{coo} \ \wedge \ (t_k \ \mathcal{C}_{cs} \ t_j)) \end{cases}$$

La première clause établit la dépendance entre deux transactions qui n'appartiennent pas à un même groupe. Les deux suivantes établissent les dépendances entre un groupe de transactions et des transactions externes.

L'histoire H est dite *COO-sérialisable* ssi $\forall t \in T \ \neg(t \ \mathcal{C}^*_{coo} \ t)$

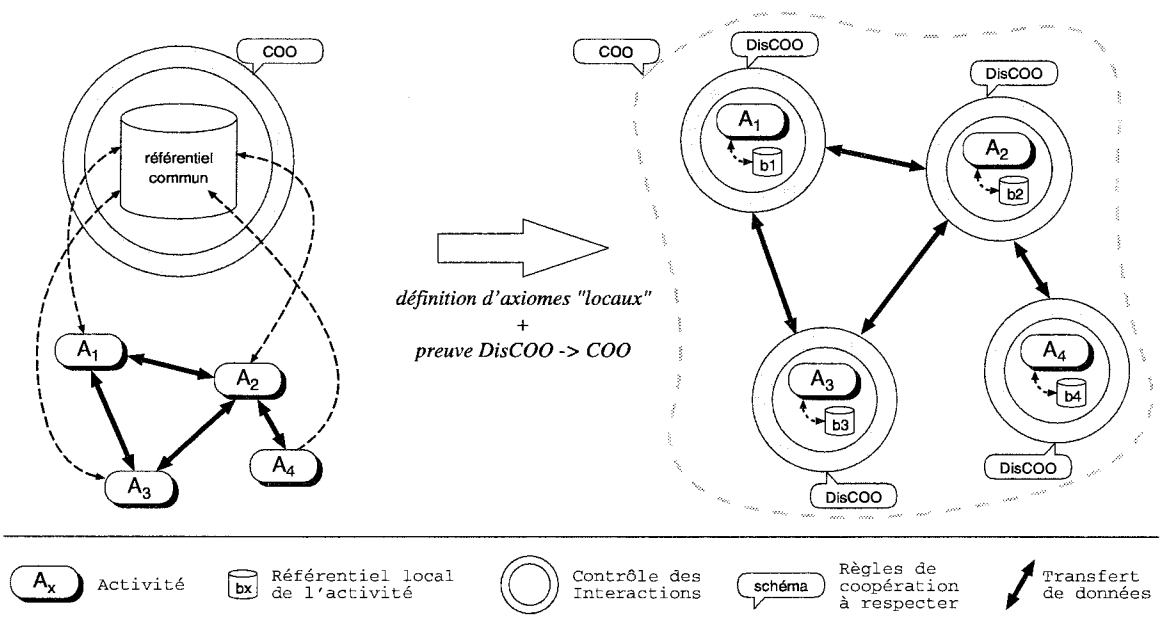
Etant donné qu'à chaque fois que l'on détecte un cycle de dépendances les transactions impliquées dans ce cycle sont groupées, le critère de correction *COO* repose donc essentiellement sur les axiomes de la définition 4.12.

Version Distribuée: *DisCOO*

Le modèle des *COO*-transactions est donc un modèle de transactions classique du point de vue de la manière dont les transactions accèdent aux objets (référentiel commun centralisé) et dont le critère de correction, la *COO*-sérialisabilité, coordonne les interactions (axiomes définis sur l'histoire globale du système). Fondamentalement, cela signifie qu'il est nécessaire d'avoir connaissance de **toutes** les opérations invoquées par **toutes** les transactions sur **tous** les objets partagés pour contrôler les interactions entre ces transactions.

Notre objectif étant d'assurer ce contrôle non plus de façon centralisée mais par le biais de contrôles effectués localement par chaque transaction, nous allons maintenant présenter une version distribuée de la *COO*-sérialisabilité: la *DisCOO*-sérialisabilité. En d'autres termes, par "distribuer un critère de correction" nous voulons exprimer le fait de définir des axiomes basés uniquement sur les informations contenues dans les histoires locales des transactions et qui, en étant ainsi vérifiés localement par chaque transaction, assurent implicitement les mêmes propriétés, d'un point de vue global, que le critère de correction initial. L'idée est donc de fournir une définition axiomatique qui soit équivalente à la définition 4.12 du point de vue des propriétés garanties mais dont les axiomes peuvent être vérifiés localement par chaque transaction. Par "équivalent" il s'agit essentiellement de prouver qu'assurer la *DisCOO*-sérialisabilité à chaque nœud du réseau de transactions nous assure également la *COO*-sérialisabilité entre les transactions, i.e. que toute exécution *DisCOO*-sérialisable est également *COO*-sérialisable.

Figure 4.5 Distribution du Critère de Correction *COO*



Intuitivement, nous pouvons interpréter les deux premiers axiomes des *COO*-transactions de la manière suivante: une transaction ne peut être validée (`Commit`) que si *"elle est à jour"*, i.e. que les résultats lus sont bien les derniers résultats en date produits par leurs transactions respectives, et que ces résultats sont "stabilisés" (i.e. que

les opérations qui les ont produits ont été validées). En d'autres termes, si une opération q est dépendante d'une opération p (i.e. $return\_value\_dep(p, q)$), alors la dernière occurence de q doit dépendre de la dernière occurence de p. Ceci sera représenté par le prédicat *up_to_date* défini ci-dessous[30] et représentant le fait que la transaction t_j est "à jour" par rapport à la transaction t_k en ce qui concerne les objets de l'ensemble O. Cette définition utilise les fonctions *LastOcc* et *Sequence* (définitions 4.15 et 4.14) qui nous fournissent respectivement la dernière occurence d'une opération dans une séquence d'opérations donnée et la séquence d'opérations à laquelle appartient l'occurence d'une opération donnée.

$$
\begin{array}{l}
up\_to\_date(t_j, t_k, O) \equiv \forall ob \in O \quad \forall Q_{t_j}[ob_{t_k}] \\
\quad (\exists t_i \; \exists p \; rvd(p_{t_i}[ob_{t_k}], LastOcc(Q_{t_j}[ob_{t_k}])) \; \wedge \; (p_{t_i}[ob_{t_k}] \rightarrow LastOcc(Q_{t_j}[ob_{t_k}])) \;) \Rightarrow \\
\quad (\; \nexists p' \in Sequence(p_{t_i}[ob_{t_k}]) \quad (LastOcc(Q_{t_j}[ob_{t_k}]) \rightarrow p'_{t_i}[ob_{t_k}]) \;)
\end{array}
$$

Définition 4.14 (Séquence d'Occurences d'une Opération)
Une même opération $p_{t_i}[ob_{t_j}]$ peut être invoquée plusieurs fois au cours de l'exécution. Bien que notées de manière identique dans l'histoire, il s'agit d'occurences différentes. L'ensemble des occurences de cette opération $p_{t_i}[ob_{t_j}]$ (ordonné selon la relation de précédence $\rightarrow$) est appelé **séquence** de l'opération $p_{t_i}[ob_{t_j}]$. Cette séquence sera également notée $P_{t_i}[ob_{t_j}]$.
La fonction *Sequence* nous renvoie donc l'ensemble des occurences d'une opération donnée à partir de l'une de ses occurences, i.e.

$$Sequence(occ) = \{p_{t_i}[ob_{t_j}] \in H \;\; | \;\; occ \equiv p_{t_i}[ob_{t_j}]\}$$

Définition 4.15 (Dernière Occurence d'une Opération)
La fonction *LastOcc* nous renvoie la dernière occurence d'une séquence d'opérations, i.e.

$$LastOcc(S) = occ \in S \quad \text{telle que} \quad \nexists \; occ' \in S \; (occ' \neq occ) \quad occ \rightarrow occ'$$

Nous pouvons maintenant exprimer le fait que la validation (`Commit`) d'une transaction n'est possible que si cette transaction est *up_to_date* par rapport à toutes les transactions dont elle dépend et si celles-ci sont elles-même validées (et ainsi de suite de manière récursive) par les deux axiomes définis ci-après.

$$
\begin{array}{l}
(\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \forall ob \quad up\_to\_date(t_j, t_k, \{ob\}) \\
(\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \\
\quad (\exists p, q \; (rvd(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \; \wedge \; (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}])) \;) \Rightarrow \\
\quad (\texttt{Commit}_{t_i} \in H_{t_k})
\end{array}
$$

30. Nous utiliserons l'abréviation *rvd* pour désigner le prédicat *return_value_dependent*.

En cas de cycle dans les dépendances (i.e. les transactions impliquées dans ce cycle forment un groupe au sens *COO*), cela signifie que toutes les transactions devront être *up_to_date* les unes par rapport aux autres, i.e. devront avoir atteint un consensus sur les valeurs des objets partagés. Ceci nous garantit donc la convergence du groupe vers un état final unique. Le deuxième axiome précise également qu'une transaction t_j dépendant d'une transaction t_i ne peut être validée ($\texttt{Commit}_{t_j} \in H_{t_k}$) que si $\texttt{Commit}_{t_i} \in H_{t_k}$. Comme nous pouvons le constater, cet axiome n'impose pas que la validation de t_i **précède** celle de t_j (i.e. $\texttt{Commit}_{t_i} \rightarrow_{H_{t_k}} \texttt{Commit}_{t_j}$). Intuitivement, cela signifie qu'en cas de cycle il n'y aura pas d'interblocage. Il suffira tout simplement que tous les `Commit` des transactions impliquées dans le cycle "apparaissent" **en même temps** dans l'histoire (i.e. que ces transactions soient validées en même temps).

Finalement, il nous faut également garantir que si une opération p est annulée, alors toutes les opérations q qui en dépendent (i.e. $return\_value\_dep(p, q)$) seront à leur tour être annulées (cf. axiome ci-dessous). Cette condition, utilisée conjointement avec les axiomes 4.9 et 4.10, impose que toutes les transactions d'un groupe soit annulées si l'une d'entre elles venait à être annulée.

$$(Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow \\ return\_value\_dep(p_{t_j}[ob_{t_k}], q_{t_i}[ob_{t_k}]) \wedge (p_{t_j}[ob_{t_k}] \rightarrow q_{t_i}[ob_{t_k}]) \Rightarrow \\ (Abort_{t_i}[q_{t_i}[ob_{t_k}]] \in H_{t_k})$$

Dans la définition 4.11, le critère de correction (la *D*-sérialisabilité) était garanti par l'axiome 4.11. Pour garantir la *DisCOO*-sérialisabilité, nous allons donc le remplacer par les axiomes 4.11 à 4.13 de la définition 4.16.

DÉFINITION 4.16 (TRANSACTIONS COOPÉRATIVES DISTRIBUÉES)
Soit t une transaction.

(4.11) $(\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \forall ob \quad up\_to\_date(t_j, t_k, \{ob\})$

(4.12)
$$(\texttt{Commit}_{t_j} \in H_{t_k}) \Rightarrow \\ (\ \exists p, q\ (rvd(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \wedge (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]))\ ) \Rightarrow \\ (\texttt{Commit}_{t_i} \in H_{t_k})$$

(4.13)
$$(Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow \\ return\_value\_dep(p_{t_j}[ob_{t_k}], q_{t_i}[ob_{t_k}]) \wedge (p_{t_j}[ob_{t_k}] \rightarrow q_{t_i}[ob_{t_k}]) \Rightarrow \\ (Abort_{t_i}[q_{t_i}[ob_{t_k}]] \in H_{t_k})$$

Comme nous l'avons déjà précisé, l'objectif de la *DisCOO*-sérialisabilité est d'assurer entre les transactions les mêmes propriétés que la *COO*-sérialisabilité mais de manière décentralisée. Par conséquent, toute exécution *DisCOO*-sérialisable doit également être une exécution *COO*-sérialisable. C'est effectivement le cas (lemme 4.4) à un détail près. En effet, dans le cas où plusieurs transactions impliquées dans un même cycle de dépendances sont groupées, les axiomes des *COO*-transactions imposent un consensus explicite de toutes ces transactions sur tous les objets manipulés par au moins une d'elles. Cela signifie que certaines transactions de ce groupe peuvent avoir à donner leur avis sur des objets dont elles n'ont même pas connaissance. A l'inverse, dans le même cas de figure, la *DisCOO*-sérialisabilité n'impose un consensus sur la valeur finale produite pour un objet *ob* qu'entre

les transactions du groupe qui ont effectivement partagé cet objet *ob*. Ce détail ne remet toutefois pas en cause les fondements de la *COO*/*DisCOO*-sérialisabilité. Il s'agit plutôt à proprement parler d'une extension aux *COO*-transactions définies dans [Mol96]. Cette extension est décrite plus précisément dans la preuve du lemme 4.4 (figure 4.6).

LEMME 4.4 (*DisCOO* $\Rightarrow$ *COO*)

Toute exécution *DisCOO*-sérialisable peut être considérée[a] comme étant également une exécution *COO*-sérialisable.

[a] Mis à part le détail mentionné ci-dessous concernant le consensus d'un groupe de *COO*-transactions sur **tous** les objets partagés dans ce groupe.

Preuve :

En supposant que les axiomes 4.11 à 4.13 des *DisCOO*-transactions (définition 4.16) sont vérifiés, nous allons démontrer que les cinq axiomes des *COO*-transactions (définition 4.12) sont également vérifiés.

1. D'après l'axiome 4.8, toutes les opérations invoquées par une transaction doivent être validées lorsque la dite transaction est validée. Donc, en particulier, la dernière occurence de chaque opération est validée.

2. Supposons $(Commit_{t_j}[q_{t_j}[ob_{t_{k_n}}]] \in H_{t_{k_n}})$. D'après l'axiome 4.7, cela signifie que $(\texttt{Commit}_{t_j} \in H_{t_{k_n}})$. Supposons en outre qu'il existe une opération $p_{t_i}[ob_{t_{k_1}}]$ dont dépende $q_{t_j}[ob_{t_{k_n}}]$, i.e. :

$$return\_value\_dep(p_{t_i}[ob_{t_{k_1}}], q_{t_j}[ob_{t_{k_n}}]) \ \wedge \ (p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}])$$

En utilisant le lemme 4.1, cela signifie que avons la suite d'opérations suivante:

$$p_{t_i}[ob_{t_{k_1}}] \rightarrow \underbrace{q_{t_{l_1}}[ob_{t_{k_1}}] \rightarrow p_{t_{l_1}}[ob_{t_{k_2}}]}_{\text{transaction } t_{l_1}} \rightarrow \ldots \rightarrow \underbrace{q_{t_{l_{n-1}}}[ob_{t_{k_{n-1}}}] \rightarrow p_{t_{l_{n-1}}}[ob_{t_{k_n}}]}_{\text{transaction } t_{l_{n-1}}} \rightarrow q_{t_j}[ob_{t_{k_n}}]$$

Par conséquent, en procédant de la droite vers la gauche et en utilisant alternativement l'axiome 4.12 (sur l'histoire $H_{t_{k_x}}$ de la transaction à laquelle appartient l'objet $ob_{t_{k_x}}$) puis les axiomes 4.7 et 4.8 (sur l'histoire $H_{t_{l_x}}$), nous obtenons que $(Commit_{t_i}[p_{t_i}[ob_{t_{k_1}}]] \in H_{t_{k_1}})$.
Ainsi, toute opération validée ne dépendra (si dépendance il y a) que d'opérations elles-même validées (axiome 2 des *COO*-transactions).

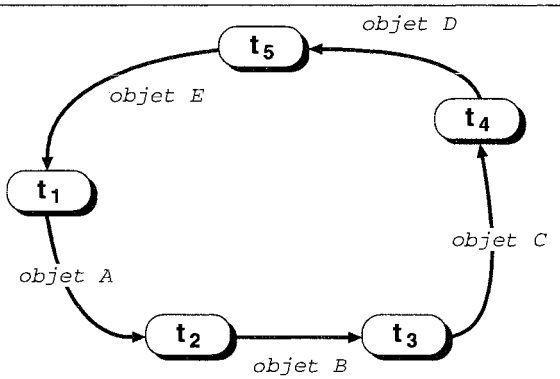
3. Soit une transaction t_j étant impliquée dans un cycle de dépendances (au sens *COO*). Si nous avons $(\texttt{Commit}_{t_j} \in H_{t_k})$ alors, d'après l'axiome 4.11, cela signifie que la transaction t_j est *up_to_date*. Par définition du prédicat *up_to_date*, si t_j a lu un résultat intermédiaire (opération $q_{t_j}[ob_{t_k}]$ telle que

$State(H^{(ob_{t_k})}) = State(H^{(ob_{t_k})} \circ q))$ produit par une transaction t_i du groupe (opération $p_{t_i}[ob_{t_k}]$ telle que $State(H^{(ob_{t_k})}) \neq State(H^{(ob_{t_k})} \circ p)$), i.e.

$$return\_value\_dep(p_{t_i}[ob], q_{t_j}[ob]) \ \wedge \ (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}])$$

alors la dernière occurence de $p_{t_i}[ob_{t_k}]$ devra précéder la dernière occurence de $q_{t_j}[ob_{t_k}]$. En outre, d'après l'axiome 4.12, avoir ($\texttt{Commit}_{t_j} \in H_{t_k}$) implique que l'on ait ($\texttt{Commit}_{t_i} \in H_{t_k}$), et donc ($Commit_{t_i}[p_{t_i}[ob_{t_k}]] \in H_{t_k}$).
En fait ceci ne correspond pas "exactement" à l'axiome 3 des *COO*-transactions. En effet, ce dernier impose un consensus explicite de **toutes les transactions** d'un groupe sur **tous les objets** manipulés par au moins une de ces transactions. Pour l'exemple donné figure 4.6, cela signifie que les transactions t_1, t_2, t_3, t_4 et t_5 doivent toutes être d'accord sur les valeurs finales des objets A, B, C, D et E. A l'inverse, la *DisCOO*-sérialisabilité n'imposera un consensus qu'entre t_1 et t_2 pour A, entre t_2 et t_3 pour B, ...

Figure 4.6



Il serait possible de modifier les définitions des axiomes 4.11 et 4.12 des *DisCOO*-transactions pour obtenir le même comportement que les *COO*-transactions. Cela ne nous semble cependant pas judicieux du point de vue de l'autonomie des transactions. En effet, il se pourrait alors qu'une transaction, pour pouvoir terminer, soit obligée de relire la valeur d'un objet dont elle n'a que faire (l'objet C pour la transaction t_1 par exemple).

4. L'axiome 4.12 implique qu'une transaction ne peut être validée (`Commit`) que si toutes les transactions dont elle dépend sont elles-même validées (leur `Commit` doit déjà figurer dans l'histoire). En cas de cycle, cela signifie que toutes les transactions du groupe devront être validées en même temps (tous les `Commit` devront "apparaître" en même temps dans l'histoire).
   En outre, l'axiome 4.13 impose que toute opération dépendant d'une opération annulée soit elle-même annulée. Utilisée conjointement avec les axiomes 4.9 et 4.10, cette condition impose que toutes les transactions d'un groupe soient annulées si l'une d'entre elles venait à être annulée (`Abort`).

5. L'axiome 5 des *COO*-transactions (abandons en cascade) correspond directement à l'axiome 4.13 des *DisCOO*-transactions.

Par conséquent, en mettant de côté le cas décrit figure 4.6, toute exécution *DisCOO*-sérialisable est une exécution *COO*-sérialisable.

□

Ainsi, mis à part le détail concernant le consensus d'un groupe de transactions, la *DisCOO*-sérialisabilité n'accepte pas d'exécution qui ne serait pas *COO*-sérialisable. Nous conservons donc les propriétés garanties par la *COO*-sérialisabilité. Nous pouvons en outre démontrer que l'inverse est également vrai, c'est-à-dire que toute exécution *COO*-sérialisable est également une exécution *DisCOO*-sérialisable (lemme 4.5), ce qui signifie que ces deux critères de correction sont donc équivalents.

LEMME 4.5 ($COO \Rightarrow DisCOO$)
Toute exécution *COO*-sérialisable est une exécution *DisCOO*-sérialisable.

Preuve :

L'axiome 4.13 des *DisCOO*-transactions est identique à l'axiome 5 des *COO*-transactions. Pour prouver les axiomes 4.11 et 4.12, il nous faut démontrer que lorsqu'une transaction est validée (`Commit`), alors elle est *up_to_date* (i.e. les dernières valeurs qu'elle a lue sont bien les dernières valeurs produites) et toutes les transactions dont elle dépend sont également validées.

- L'axiome 3 des *COO*-transactions implique que toute publication d'un résultat final (i.e. opération $p_{t_i}[ob]$ telle que $State(H^{(ob)}) \neq State(H^{(ob)} \circ p)$ et $Commit_{t_i}[p_{t_i}[ob]] \in H$) effectuée par une transaction d'un groupe doit être suivie d'une relecture de ce résultat (i.e. opération $q_{t_j}[ob]$ telle que $State(H^{(ob)}) = State(H^{(ob)} \circ q)$) de la part de chacune des transactions de ce groupe avant que celles-ci ne puissent terminer (`Commit`). En outre, si $Commit_{t_i}[p_{t_i}[ob]] \in H$, cela signifie que $p_{t_i}[ob]$ est la dernière occurence d'une séquence d'opérations (cf. axiome 1 des *COO*-transactions).
  Avant de pouvoir être validée (`Commit`), une transaction faisant partie d'un groupe doit donc être *up_to_date* par rapport à **toutes les transactions du groupe**. En particulier, elle doit donc l'être par rapport aux transactions dont elle dépend directement. Le critère de correction *COO* est donc plus contraignant que le critère de correction *DisCOO* (cf. exemple de la figure 4.6).

- L'axiome 4 des *COO*-transactions impose que si une transaction d'un groupe est validée (`Commit`), alors toutes les transactions du groupe le sont également.

Ainsi, toute exécution *COO*-sérialisable est également *DisCOO*-sérialisable. Le modèle des *DisCOO*-transactions peut donc être considéré comme étant une extension du modèle des *COO*-transactions.

□

Fondamentalement, la *DisCOO*-sérialisabilité garantit donc les mêmes propriétés que la *COO*-sérialisabilité définie dans [Mol96], à savoir que toute lecture d'un résultat intermédiaire effectuée par une transaction sera suivie d'une relecture du résultat final correspondant par cette même transaction et que toute opération validée ne dépend que d'opérations elles-même validées. Les *DisCOO*-transactions, dont la définition axiomatique complète est fournie en annexe B, peuvent toutefois être considérées comme **une extension des *COO*-transactions**, non seulement du point de vue de la **distribution du critère de correction**, mais également du point de vue du **consensus sur les valeurs finales des objets** dans le cas d'un groupe de transactions. Alors que les *COO*-transactions exigent que **toutes les transactions** du groupe s'accordent sur les valeurs de **tous les objets** partagés au sein du groupe, les *DisCOO*-transactions imposent seulement que deux transactions du groupe ayant partagé un objet s'accordent sur sa valeur finale, sans tenir compte des objets manipulés par les autres transactions du groupe.

Apports de *DisCOO* par rapport à *COO*

Par rapport à la *COO*-sérialisabilité, la *DisCOO*-sérialisabilité nous permet ainsi d'offrir une plus grande autonomie aux transactions de notre modèle. En effet, pour contrôler que les interactions d'une transaction avec ses transactions partenaires sont correctes, au moment de sa validation (`Commit`) par exemple, seules des **informations locales** à cette transaction sont utilisées, c'est-à-dire des informations journalisées soit dans sa propre histoire locale, soit dans les histoires locales des transactions avec lesquelles elle a échangé des données. Cela signifie en particulier que la *DisCOO*-sérialisabilité n'est plus fondée sur la détection de cycles de dépendances entre les transactions, car déterminer de tels cycles nécessite d'avoir une vue globale du système. La *DisCOO*-sérialisabilité effectue plutôt un contrôle "de proche en proche". Par conséquent, la notion de groupe de transactions, bien que toujours sous-jacente, n'est plus utilisée explicitement par le modèle des *DisCOO*-transactions.

Du point de vue de la mise en œuvre, la principale difficulté consistera à éviter les interblocages au niveau des `Commit` pouvant être engendrés par l'axiome 4.12 en cas de cycle entre les *DisCOO*-transactions (cf. section 5.1.3).

4.2.3 Résumé

Après avoir présenté dans la première partie de notre travail un modèle de transactions distribuées dans lequel les différentes transactions possèdent chacune leur propre référentiel local et coopèrent par le biais d'échanges de données explicites entre leurs référentiels locaux, l'objectif de cette seconde partie était de proposer un moyen de distribuer deux critères de correction existants, la sérialisabilité puis la *COO*-sérialisabilité, c'est-à-dire de définir des contraintes sur ces échanges de manière à garantir qu'ils n'introduisent pas d'incohérence au niveau des données produites vis-à-vis, respectivement, de la sérialisabilité et de la *COO*-sérialisabilité.

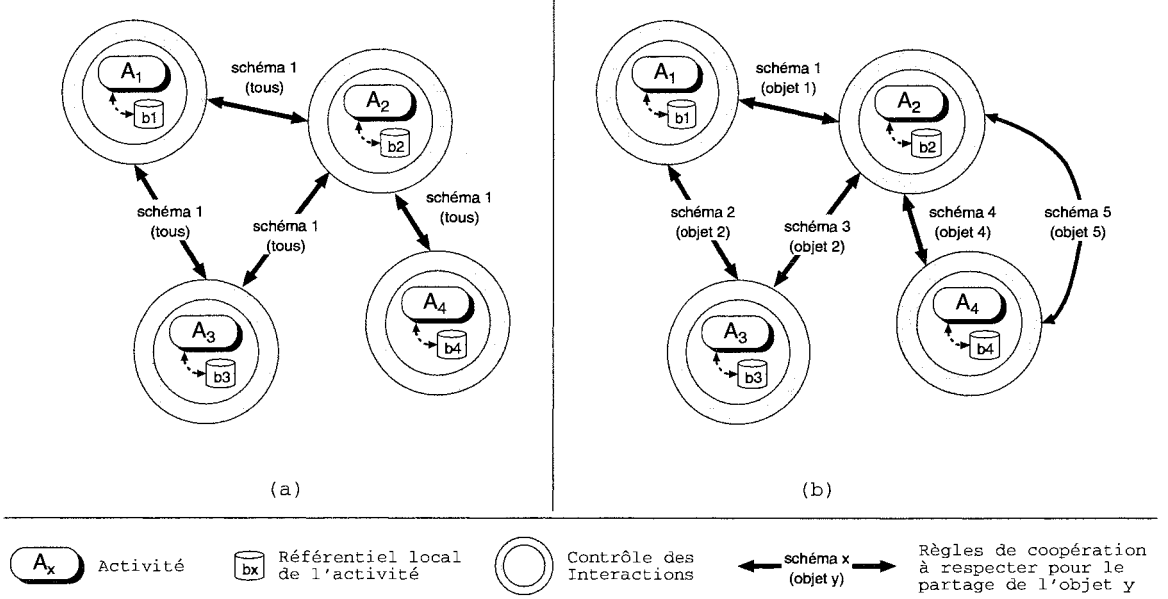
La *DisCOO*-sérialisabilité (respectivement la *D*-sérialisabilité) se distingue de la *COO*-sérialisabilité (respectivement de la sérialisabilité) par le fait que ces contraintes ne sont basées que sur des informations locales aux activités. En effet, tant la *COO*-sérialisabilité

que la sérialisabilité classique définissent des axiomes basés sur l'histoire globale du système, notamment en ce qui concerne la détection des cycles de dépendances entre les transactions (notion de groupe de transactions pour la *COO*-sérialisabilité; définition 4.8 pour la sérialisabilité). Comme nous l'avons démontré, la *DisCOO*-sérialisabilité (respectivement la *D*-sérialisabilité) est donc un critère de correction local qui, s'il est vérifié au niveau de l'histoire locale de chacune des transactions, assure les mêmes propriétés que la *COO*-sérialisabilité (respectivement la sérialisabilité) au niveau de l'histoire globale.

4.3 Système Hétérogène

L'objectif de ce chapitre ne se limite toutefois pas uniquement à reformuler, certes de manière **distribuée** ou **décentralisée**, des critères de correction existants tels que la sérialisabilité ou la *COO*-sérialisabilité (figure 4.7-a). En définissant des critères de correction distribués nous sommes en fait passés d'un contrôle des interactions (entre deux transactions quelconques du système) à un **contrôle des échanges** entre une transaction donnée et ses transactions partenaires (opérations figurant dans l'histoire locale de cette transaction). Nous avons pour cela défini des règles de coopération (les axiomes de la *D*-sérialisabilité puis de la *DisCOO*-sérialisabilité) permettant de garantir, en n'utilisant que des informations locales aux transactions, les mêmes propriétés que dans un système où le contrôle est centralisé.

Figure 4.7 Hétérogénéïté des Règles de Coopération



Cette troisième partie du chapitre développe cette idée en définissant les notions de **schéma de coopération** et de **négociation** d'un tel schéma entre deux transactions, l'objectif final étant de permettre aux différentes transactions du système de coopérer en utilisant des règles de coopération éventuellement différentes pour coordonner leurs interactions (figure 4.7-b). Les transactions de notre modèle sont ainsi organisées en réseau, les

nœuds réprésentant les transactions et les arcs réprésentant les schémas de coopération négociés entre les transactions. Par conséquent, il n'existe plus **un seul** critère de correction contrôlant tous les échanges mais **plusieurs** critères de correction distribués devant cohabiter au sein du même système. Cette approche nous évite également de devoir définir un seul et unique schéma de coopération qui serait utilisé par toutes les transactions pour le partage de tous leurs objets.

Toutefois, comme nous pouvons le constater sur la figure 4.7-b, cela signifie que la couche "contrôle des interactions" d'une transaction donnée devra tenir compte des différents schémas de coopération négociés avec les autres transactions pour le partage des différents objets. Or il est tout à fait possible que, durant l'exécution de cette transaction, un échange de donnée avec une autre transaction soit autorisé par le schéma de coopération qu'elles auront négocié, mais que ce même échange ne soit pas correct vis-à-vis d'un autre schéma de coopération négocié avec une autre transaction (et éventuellement pour le partage d'un objet différent). Ainsi, le fait de permettre aux transactions de coopérer en respectant des schémas de coopération différents pour le partage de leurs objets peut provoquer des **interactions** entre ces schémas de coopération, voire même des **conflits** comme nous l'avons expliqué précédemment. Quel est alors le comportement global du système? En d'autres termes, puisque les interactions entre transactions ne sont pas toutes soumises aux mêmes règles, quelles sont les propriétés que nous pouvons garantir entre deux transactions quelconque du système?

Cette troisième partie de notre travail se déroule en deux étapes. Nous définissons tout d'abord les notions de "schéma de coopération" puis de "négociation d'un schéma de coopération entre deux transactions pour le partage d'un objet". Nous présentons également des exemples de schémas de coopération dérivés de la *D*-sérialisabilité et de la *DisCOO*-sérialisabilité. Nous étudions ensuite de quelle manière peuvent apparaître des interactions entre plusieurs schémas et, si elles sont source de conflit, de quelle façon nous pouvons gérer ces conflits entre schémas de coopération.

4.3.1 Schémas de Coopération

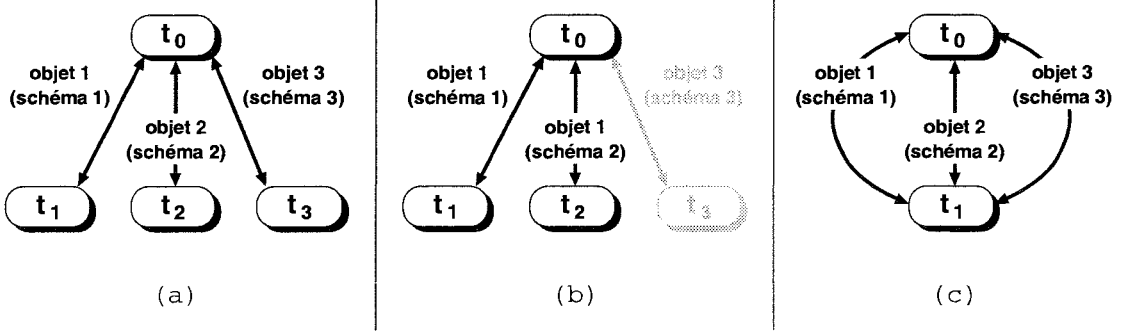
L'idée est de ne plus utiliser les mêmes règles de coopération (i.e. un seul et unique critère de correction) pour contrôler toutes les interactions de toutes les transactions du système, mais de permettre à chaque transaction de "choisir" à quelles règles seront soumises ses interactions avec les autres transactions. Nous pouvons ainsi adapter le contrôle des échanges aux transactions impliquées dans ces échanges et aux objets échangés[31].

Etant donné que nous nous plaçons dans une architecture d'égal à égal, cela signifie que les transactions devront s'accorder sur les règles à respecter lors de leurs échanges. Nous parlerons alors de **négociation** d'un **schéma de coopération** entre deux transactions. De cette manière, une même transaction pourra négocier différents schémas de

31. Dans le cas de la construction d'un bâtiment par exemple, l'architecte chargé d'établir les plans de ce bâtiment ne coopèrera pas de la même manière que ce soit avec les différents corps de métier impliqués dans le projet (plomberie, électricité, chauffage, ...) ou que ce soit avec les organismes de contrôle (procès verbaux de respect des normes de sécurité, accord de la commune sur les aspects d'intégration urbaine, ...). Il s'agit en quelque sorte de prendre en compte, au niveau de la coordination des échanges entre ces différents partenaires, leurs rôles et responsabilités respectifs dans le projet.

coopération pour partager ses objets avec les autres transactions du système (figure 4.8-a). En particulier, elle pourra partager un même objet avec plusieurs transactions en utilisant à chaque fois un schéma de coopération différent (figure 4.8-b). En outre, puisque les schémas de coopération seront négociés entre deux transactions pour le partage d'objets donnés, deux transactions auront également la possibilité d'utiliser des schémas de coopération différents pour le partage d'objets distincts (figure 4.8-c).

Figure 4.8 Négociation de Différents Schémas de Coopération



Un schéma de coopération est en fait défini comme étant un ensemble de règles de coopération, exprimées sous la forme de contraintes sur les histoires locales des transactions, qui seront ensuite utilisées en tant que pré-conditions aux événements invoqués par les transactions (opérations sur des objets ou événements significatifs). Par rapport à un modèle de transactions et à un critère de correction classiques (définis en ACTA), cela revient à utiliser différents ensembles d'axiomes pour contrôler les différentes transactions de notre modèle. Un schéma de coopération est ainsi représenté par un prédicat $s(t_i, t_j, O)$ définissant la liste (i.e. une conjonction) des contraintes à respecter entre les transactions t_i et t_j pour le partage des objets $ob \in O$. Par exemple, le schéma de coopération correspondant à la D-sérialisabilité serait exprimé par le prédicat $DSeria$ défini ci-dessous. Celui-ci impose que les transactions t_i et t_j soient sérialisables par rapport aux opérations qu'elles auront invoquées pour s'échanger les objets $ob \in O$.

$$DSeria(t_i, t_j, O) \equiv \left\{ \begin{array}{l} \forall ob \in O \quad \forall t_1, t_2 \ (t_1 \in \{t_i, t_j\}, \ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k \ (t_k \neq t_1) \\ \quad (Commit_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \quad\quad conflict(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}]) \ \wedge \ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}]) \ \Rightarrow \\ \quad\quad (Commit_{t_k}[p_{t_k}[ob_{t_2}]] \rightarrow_{H_{t_2}} Commit_{t_1}[q_{t_1}[ob_{t_2}]]) \end{array} \right.$$

Comme nous l'avons déjà mentionné, le schéma de coopération s utilisé par deux transactions t_i et t_j pour partager les objets $ob \in O$ devra avoir été negocié entre ces deux transactions. Il s'agit effectivement d'un contrat passé entre les deux transactions, chacune d'elles s'engageant à respecter les règles de coopération fixées par ce contrat. Celui-ci sera représenté par l'événement $Contract[t_i, t_j, O, s]$ dans les histoires des deux transactions. Cela signifie que les schémas de coopération ne sont pas fixés au démarrage d'une transaction, mais qu'ils sont bien négociés dynamiquement par les différentes transactions au cours de leur exécution.

La négociation d'un schéma de coopération doit cependant respecter certaines règles et induit elle-même certains contrôles sur les échanges de données utltérieurs (définition 4.17). Tout d'abord, le contrat doit être accepté par les deux transactions, i.e. l'événement *Contract* doit apparaître dans les histoires locales des deux parties (axiome 1). Ensuite, il est nécessaire de garantir que deux transactions ont négocié un schéma de coopération avant toute invocation d'une opération d'échange de données et que les règles de coopération définies par ce schéma sont bien vérifiées (axiome 2). L'axiome 3 limite la négociation à un seul schéma de coopération par objet et par couple de transactions. Cela signifie qu'il ne sera pas possible, dans un premier temps, de renégocier un schéma de coopération. Finalement, une transaction ne pourra être terminée (`Commit`, `Abort`, ...) que si toutes les règles de coopération de tous les schémas qu'elle aura négociés le lui permettent (axiome 4).

DÉFINITION 4.17 (AXIOMES DE RESPECT DES SCHÉMAS NÉGOCIÉS)

1. Un contrat est ratifié simultanément par les deux partenaires, c'est-à-dire que l'événement *Contract* est journalisé en même temps par les deux transactions:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)\quad \forall O\quad \forall s$$
$$(Contract[t_i, t_j, O, s] \in H_{t_1})\ \Rightarrow\ (Contract[t_i, t_j, O, s] \in H_{t_2})$$

2. Une interaction entre deux transactions ne peut avoir lieu que si ces deux transactions ont préalablement négocié un schéma de coopération et si les règles de ce schéma sont vérifiées:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad (p_{t_i}[ob_{t_j}] \in H_t)\ \Rightarrow$$
$$\exists t_1 \in \{t_i, t_j\}\quad \exists t_2 \in \{t_i, t_j\} - t_1\quad \exists O\ (ob \in O)\quad \exists s$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} p_{t_i}[ob_{t_j}])\ \wedge\ s(t_1, t_2, O)$$

3. Le partage d'un objet donné est contrôlé par un et un seul schéma de coopération (i.e. pas de renégociation possible pour le moment):

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad \forall ob\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)$$
$$(Contract[t_1, t_2, O, s] \in H_t)\ \wedge\ (ob \in O)\ \Rightarrow\ \nexists s'$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_1, t_2, O', s']\ \wedge\ (ob \in O'))$$
$$\vee\ (Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_2, t_1, O', s']\ \wedge\ (ob \in O'))$$

4. Une transaction ne peut être terminée (`Commit`, `Abort`, ...) que si tous les schémas de coopération qu'elle a négociés avec d'autres transactions sont vérifiés:

$$\forall \alpha \in TE_{t_i}\quad (\alpha \in H_{t_i})\ \Rightarrow\ \forall t_j\quad \forall ob\quad \forall s\quad \forall t \in \{t_i, t_j\}$$
$$(Contract[t_i, t_j, O, s] \in H_t)\ \wedge\ (ob \in O)\ \Rightarrow\ s(t_i, t_j, O)$$
$$\vee\ (Contract[t_j, t_i, O, s] \in H_t)\ \wedge\ (ob \in O)\ \Rightarrow\ s(t_j, t_i, O)$$

Un schéma de coopération ne permet donc que de contrôler les échanges de certains objets entre deux transactions données. Ceci constitue une différence essentielle par rapport à un critère de correction classique. Ainsi, un schéma de coopération (négocié pour le partage d'un ensemble d'objets O) ne garantit la correction des exécutions de deux transactions que **par rapport** aux opérations leur ayant permis de s'échanger des valeurs des objets $ob \in O$. Cela signifie qu'un schéma de coopération n'assure aucune propriété au ni-

veau de l'exécution globale, du moins pas de manière explicite. En effet, nous démontrons ci-après que si **toutes** les transactions du système ont négocié le schéma de coopération *DSeria* (respectivement *DisCOO*) pour le partage de **tous** leurs objets, alors l'exécution (globale) est *D*-sérialisable (respectivement *DisCOO*-sérialisable).

Exécution *D*-Sérialisable

Il nous faut définir un schéma de coopération qui, lorsqu'il est négocié entre deux transactions pour un ensemble d'objets O, assure que les exécutions de ces deux transactions sont *D*-sérialisables par rapport aux opérations leur ayant permis de s'échanger des valeurs des objets $ob \in O$. Si toutes les interactions entre les transactions sont contrôlées par ce schéma de coopération, celui-ci doit en outre garantir **implicitement** la *D*-sérialisabilité de l'exécution globale.

La *D*-sérialisabilité, version distribuée de la sérialisabilité, assure que des transactions concourantes s'exécutent sans interférence comme si elles s'exécutaient en série. Pour cela, le schéma de coopération *DSeria* (négocié pour l'ensemble d'objets O) impose que toute opération invoquée par une transaction t_1 sur un objet $ob \in O$ d'une transaction t_2 et dont le résultat dépend d'une autre opération effectuée précédemment par une transaction t_k ($t_k \neq t_1$) sur ce même objet ob_{t_2} ne puisse être validée (`Commit`) qu'une fois que l'opération de la transaction t_k dont elle dépend aura été validée. En d'autres termes, la transaction t_2 a pour rôle de garantir que toutes les transactions t_k dont dépend la transaction t_1 pour les objets de l'ensemble O sont validées avant que la transaction t_1 ne puisse elle-même être validée. La transaction t_1 est ainsi sérialisée par rapport à la transaction t_2 du point de vue des opérations invoquées sur les objets $ob \in O$ de la transaction t_2.

$$DSeria(t_i, t_j, O) \equiv \left\{ \begin{array}{l} \forall ob \in O \quad \forall t_1, t_2 \ (t_1 \in \{t_i, t_j\}, \ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k \ (t_k \neq t_1) \\ \quad (Commit_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \quad\quad conflict(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}]) \ \wedge \ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}]) \ \Rightarrow \\ \quad\quad (Commit_{t_k}[p_{t_k}[ob_{t_2}]] \rightarrow_{H_{t_2}} Commit_{t_1}[q_{t_1}[ob_{t_2}]]) \end{array} \right.$$

Intuitivement, du point de vue des schémas de coopération, pour pouvoir garantir la *D*-sérialisabilité de l'exécution globale, il est nécessaire que toutes les transactions du système aient négocié le schéma de coopération *DSeria* pour le partage de tous leurs objets. De cette manière, toutes les opérations effectuées par les différentes transactions ne pourront être validées (`Commit`) que si toutes les opérations dont elles dépendent ont été elles-mêmes validées.

LEMME 4.6 (EXÉCUTION *D*-SÉRIALISABLE)
Si toutes les interactions entre les transactions sont contrôlées par le schéma de coopération *DSeria*, ce qui est représenté par l'axiome suivant:

$$\forall t_i, t_j \ (t_i \neq t_j) \quad \forall t \in \{t_i, t_j\} \quad \forall O \quad (Contract[t_i, t_j, O, s] \in H_t) \ \Rightarrow \ (s = DSeria)$$

alors l'exécution obtenue est *D*-sérialisable.

Preuve :

D'après l'axiome de négociation 2 (définition 4.17), une interaction entre deux transactions ne peut avoir lieu que si ces deux transactions ont préalablement négocié un schéma de coopération. Or, par hypothèse, le seul schéma de coopération pouvant être négocié est le schéma *DSeria*. Par conséquent, d'après l'axiome de négociation 4 et la définition du prédicat *DSeria*, une transaction ne pourra être validée que si toutes les opérations dont dépendent ses propres opérations sont elles-mêmes validées, i.e. $\forall t_j \quad \forall ob \quad \forall t_k$

$$(Commit_{t_j}[q_{t_j}[ob_{t_k}]] \in H_{t_k}) \Rightarrow$$
$$conflict(p_{t_i}[ob_{t_k}], q_{t_j}[ob_{t_k}]) \wedge (p_{t_i}[ob_{t_k}] \rightarrow q_{t_j}[ob_{t_k}]) \Rightarrow$$
$$(Commit_{t_i}[p_{t_i}[ob_{t_k}]] \rightarrow Commit_{t_j}[q_{t_j}[ob_{t_k}]])$$

Or ceci correspond précisément à l'axiome nous permettant d'établir si une exécution est ou n'est pas *D*-sérialisable (axiome 4.11 de la définition 4.11).

□

Comme nous pouvons le constater, la *D*-sérialisabilité, en tant que schéma de coopération, est bien une propriété locale. En effet, il s'agit d'une propriété que chaque transaction concernée devra évaluer localement sur sa propre histoire. Lorsqu'une transaction t_j désirera terminer (`Commit`) elle devra valider toutes les opérations qu'elle aura effectuées. Etant donnée qu'une opération $op_{t_j}[ob_{t_i}]$ aura été journalisée à la fois dans l'histoire locale de t_j et dans l'histoire locale de t_i, son "`Commit`" apparaîtra également dans les deux histoires. De ce fait, les deux transactions t_j et t_i évalueront le prédicat *DSeria*, chacune sur sa propre histoire locale (figure 4.1-b). A l'inverse d'un système centralisé (figure 4.1-a), il n'y a donc plus de "serveur" qui contrôle toutes les interactions entre toutes les transactions du système.

Exécution *DisCOO*-Sérialisable

Nous allons maintenant définir le schéma de coopération *DisCOO* que deux transactions t_i et t_j pourront négocier pour un ensemble d'objets O afin de garantir que leurs exécutions sont *DisCOO*-sérialisables par rapport à leurs interactions concernant les objets $ob \in O$. Ces deux transactions pourront alors, au cours de leur exécution, s'échanger mutuellement des valeurs intermédiaires des objets $ob \in O$. Le schéma *DisCOO* assurera qu'elles seront toutes les deux à jour l'une par rapport à l'autre en ce qui concerne les objets $ob \in O$ avant qu'elles ne soient validées.

Au paragraphe 4.2.2, les axiomes 4.11 à 4.13 de la définition 4.16 ont été établis de manière à garantir qu'une exécution (globale) est *DisCOO*-sérialisable. Par rapport aux axiomes de la *COO*-sérialisabilité (définition 4.12), le principal avantage est que pour déterminer si une transaction peut être validée (`Commit`) ils n'utilisent que des informations "locales" à cette transaction, i.e. des informations journalisées soit dans sa propre histoire locale, soit dans les histoires locales des transactions avec lesquelles elle a échangé des données. Par conséquent, ces axiomes constituent à peu de chose près les règles de coopération du schéma de coopération *DisCOO*.

1. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Si la transaction t_1 désire être validée (`Commit`), elle doit être à jour par rapport à la transaction t_2. Cela signifie que les résultats de t_2 dont la transaction t_1 a eu connaissance sont bien les derniers résultats en date au niveau de la transaction t_2. Nous représentons cette propriété par la règle de coopération suivante:

$$\begin{array}{l}\forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \\ \quad (\mathtt{Commit}_{t_1} \in H_{t_2}) \ \Rightarrow\ up\_to\_date(t_1, t_2, O)\end{array}$$

2. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Une opération de la transaction t_1 ne peut être validée (`Commit`) qu'à condition que toutes les opérations invoquées par des transactions t_k ($t_k \neq t_1$) sur des objets $ob \in O$ de la transaction t_2 et dont elle est dépendante soient elles-mêmes validées. Cette propriété est matérialisée par la règle de coopération suivante:

$$\begin{array}{l}\forall ob \in O \quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1) \\ \quad (\mathtt{Commit}_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \qquad rvd(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}]) \ \wedge\ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}]) \ \Rightarrow \\ \qquad (\mathtt{Commit}_{t_k}[p_{t_k}[ob_{t_2}]] \in H_{t_2})\end{array}$$

3. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Si une opération de la transaction t_1 est annulée (`Abort`), alors toutes les opérations, au niveau de la transaction t_2, qui en dépendent doivent également être annulées, i.e.:

$$\begin{array}{l}\forall ob \in O \quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1) \\ \quad (Abort_{t_1}[p_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \qquad rvd(p_{t_1}[ob_{t_2}], q_{t_k}[ob_{t_2}]) \ \wedge\ (p_{t_1}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_k}[ob_{t_2}]) \Rightarrow \\ \qquad (Abort_{t_k}[q_{t_k}[ob_{t_2}]] \in H_{t_2})\end{array}$$

Le schéma de coopération *DisCOO* peut alors être représenté par le prédicat du même nom qui est en fait la conjonction des trois règles de coopération énoncées ci-dessus. Cela signifie que tous les événements (invocations d'opérations et événements significatifs) qui sont sous le contrôle de ce schéma de coopération (cf. définition 4.17) doivent vérifier ces trois axiomes en plus des axiomes de base de notre modèle de transactions.

$$DisCOO(t_i, t_j, O) \equiv \left\{ \begin{array}{l}\forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \\ \quad (\mathtt{Commit}_{t_1} \in H_{t_2}) \ \Rightarrow\ up\_to\_date(t_1, t_2, O) \\ \\ \forall ob \in O \quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1) \\ \quad (\mathtt{Commit}_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \qquad rvd(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}]) \ \wedge\ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}]) \ \Rightarrow \\ \qquad (\mathtt{Commit}_{t_k}[p_{t_k}[ob_{t_2}]] \in H_{t_2}) \\ \\ \forall ob \in O \quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k\ (t_k \neq t_1) \\ \quad (Abort_{t_1}[p_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \qquad rvd(p_{t_1}[ob_{t_2}], q_{t_k}[ob_{t_2}]) \ \wedge\ (p_{t_1}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_k}[ob_{t_2}]) \Rightarrow \\ \qquad (Abort_{t_k}[q_{t_k}[ob_{t_2}]] \in H_{t_2})\end{array} \right.$$

Comme dans le cas de la D-sérialisabilité, le schéma de coopération $DisCOO$ a été défini de telle sorte que si toutes les transactions du système ont négocié le schéma de coopération $DisCOO$ pour le partage de tous leurs objets, alors l'exécution (globale) obtenue est $DisCOO$-sérialisable.

LEMME 4.7 (EXÉCUTION $DisCOO$-SÉRIALISABLE)
Si toutes les interactions entre les transactions sont contrôlées par le schéma de coopération $DisCOO$, ce qui est représenté par l'axiome suivant

$$\forall t_i, t_j \ (t_i \neq t_j) \quad \forall t \in \{t_i, t_j\} \quad \forall O \quad (Contract[t_i, t_j, O, s] \in H_t) \Rightarrow (s = DisCOO)$$

alors l'exécution est $DisCOO$-sérialisable.

Preuve :

Si l'on suppose que le seul schéma de coopération pouvant être négocié entre les différentes transactions est le schéma de coopération $DisCOO$, alors, en se basant sur les axiomes de négociation de la définition 4.17, nous retrouvons les axiomes garantissant la $DisCOO$-sérialisabilité (définition 4.16).

□

Variantes de $DisCOO$: Schémas Client/Serveur et Rédacteur/Relecteur

Le schéma de coopération $DisCOO$ a été défini de manière à garantir la $DisCOO$-sérialisabilité entre deux transactions pour le partage d'un ensemble d'objets donné. Il nous permet ainsi de supporter le mode de coopération appelé "**écriture coopérative**" dans lequel deux transactions modifient simultanément un même ensemble d'objets. Toutefois, il ne s'agit là que du cas le plus général d'exécutions $DisCOO$-sérialisables. En effet, le critère de correction $DisCOO$ étant une version distribuée du critère COO, celui-ci supporte également les deux autres modes de coopération autorisés par la COO-sérialisabilité: les modes **"client/serveur"** et **"rédacteur/relecteur"**. Ces deux modes correspondent en fait au mode "écriture coopérative" que l'on a spécialisé en ajoutant de nouvelles contraintes (règles de coopération) sur les échanges.

- mode **"client/serveur"**: dans ce cas de figure, une transaction appelée "serveur" produit différents résultats intermédiaires successifs pour un ensemble d'objets. Une autre transaction, appelée "client", lit (certains de) ces résultats intermédiaires, ce qui lui permet de commencer à travailler sans avoir à attendre que le serveur ait produit le résultat final. Bien entendu, le client devra tenir compte des résultats finaux correspondants produits par le serveur avant de pouvoir produire ses propres résultats finaux.
- mode **"rédacteur/relecteur"**: ce mode de coopération est similaire au mode "écriture coopérative", excepté le fait que les interactions entre les deux transactions ont lieu sur deux ensembles d'objets différents: le rédacteur produit des "documents" relus par le "relecteur" qui produit à son tour des "revues" dont doit tenir compte

le rédacteur pour mettre à jour ses documents. Les transactions sont également inter-dépendantes et doivent donc se terminer en même temps.

Dans le cas du schéma client/serveur, les échanges de données ne peuvent avoir lieu que du serveur vers le client. Supposons que deux transactions t_s (le serveur) et t_c (le client) décident d'utiliser ce schéma pour se partager les objets de l'ensemble O. La transaction cliente t_c ne sera pas autorisée à effectuer d'opérations qui modifieraient la valeur d'un objet $ob \in O$ possédé par la transaction serveur t_s (deuxième règle du prédicat *client_server*). De son côté, la transaction serveur t_s ne pourra lire la valeur d'un objet $ob \in O$ possédé par la transaction cliente t_c (troisième règle du prédicat *client_server*). Les échanges de données concernant les objets de l'ensemble O sont ainsi orientés du serveur (transactions t_s) vers le client (transactions t_c).

$$client\_server(t_c, t_s, O) \equiv \begin{cases} DisCOO(t_c, t_s, O) \\ \forall ob \in O \quad \nexists\, p \in Write^{(ob)} \quad \text{tel que} \quad p_{t_c}[ob_{t_s}] \in H \\ \forall ob \in O \quad \nexists\, q \in Read^{(ob)} \quad \text{tel que} \quad q_{t_s}[ob_{t_c}] \in H \end{cases}$$

En ce qui concerne le schéma rédacteur/relecteur, nous pouvons considérer qu'il s'agit de l'association de deux schémas client/serveur: l'un du rédacteur vers le relecteur pour échanger les documents à relire, et l'un du relecteur vers le rédacteur pour transmettre les revues de ces documents. Ce schéma de coopération se différencie toutefois de ceux que nous avons vu jusqu'à maintenant puisqu'il distingue deux catégories dans l'ensemble des objets pour lequel il est négocié: les "documents" (sous-ensemble *Doc*) et les "revues" (sous-ensemble *Rev*). Comme expliqué précédemment, ce sont les deuxième et troisème règles du prédicat *client_server* qui nous permettent d'orienter les échanges de ces différents objets entre les deux transactions t_{wri} (le rédacteur) et t_{rev} (le relecteur).

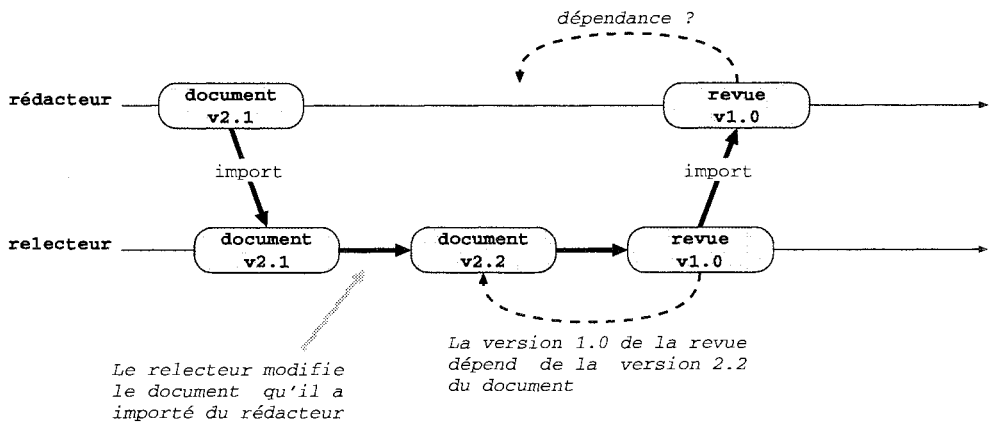
$$writer\_reviewer(t_{wri}, t_{rev}, Doc \cup Rev) \equiv \begin{cases} client\_server(t_{rev}, t_{wri}, Doc) \\ client\_server(t_{wri}, t_{rev}, Rev) \\ \vdots \end{cases}$$

Un point important du schéma rédacteur/relecteur est que la dernière valeur de chaque revue doit être produite à partir de la dernière valeur des documents à relire et, qu'à son tour, la dernière valeur de chaque document doit être produite à partir de la dernière valeur des revues. Ceci est assuré par le prédicat *DisCOO* des deux prédicats *client_server*. En effet, étant donné que nous avons introduit un cycle de dépendances entre ces deux transactions, elles devront chacune être *up_to_date* l'une par rapport à l'autre avant de pouvoir être validées. Cela nous assure que le relecteur (la transaction t_{rev}) aura bien relu la dernière version de chaque document (objets $doc \in Doc$) et que le rédacteur (la transaction t_{wri}) aura bien eu connaissance de la dernière version de chaque revue (objets $rev \in Rev$).

Un second aspect du schéma rédacteur/relecteur est qu'il induit une inter-dépendance entre les documents et les revues. Cela signifie qu'il sera nécessaire de vérifier qu'une revue du relecteur qui sera lue par le rédacteur aura bien été produite "à partir" de documents fournis par le rédacteur, et non d'une version modifiée d'un de ces documents. En effet, le schéma client/serveur utilisé entre le rédacteur et le relecteur pour partager les documents

n'impose des contraintes que sur les échanges entre ces deux activités et non sur la manière dont elles doivent utiliser les données partagées. En d'autres termes, le relecteur sera tout à fait libre de modifier (dans sa base locale) un des documents qu'il aura importé du rédacteur, mais ne pourra en aucun cas "propager" ces modifications au rédacteur.

Figure 4.9 Dépendances entre les Documents et les Revues



Toutefois, dans le cas du schéma rédacteur/relecteur, cette contrainte n'est pas suffisante. Supposons en effet que le relecteur importe un document du rédacteur, modifie ce document, puis rédige une revue en se basant sur la version modifiée de ce document. Il s'agit du cas représenté sur la figure 4.9. Jusque là, ceci est parfaitement conforme à notre schéma de coopération. Par contre, lorsque le rédacteur désirera importer cette revue, celui-ci sera confronté à un problème de cohérence de ses données: cette revue a été produite à partir d'une version du document qu'il ne possède pas dans sa base locale. En outre, du fait des schémas client/serveur utilisés, il lui est impossible d'importer la version du document modifiée par le relecteur! Il est donc nécessaire de contrôler l'importation de la revue, faite par le rédacteur, en imposant que cette revue ait été rédigée à partir de versions non modifiées, i.e. fournies par le rédacteur, des documents. Ceci est représenté par la règle de coopération suivante:

$$\begin{array}{l} \forall\ rev \in Rev \quad (op \in Read_{t_{wri}}[rev_{t_{rev}}] \cup Write_{t_{rev}}[rev_{t_{wri}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\ doc \in Doc \quad \nexists\ p \in Write^{(doc)} \\ \qquad (LastOcc(Read_{t_{rev}}[doc_{t_{wri}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{wri}}[doc_{t_{rev}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \end{array}$$

Cela signifie que si l'on désire "propager" la valeur d'un objet $rev \in Rev$ du relecteur vers le rédacteur, pour chaque objet $doc \in Doc$, la "version courante" de l'objet doc au niveau du relecteur doit provenir du rédacteur, i.e. soit le relecteur a lu cette version chez le rédacteur (opération $read_{t_{rev}}[doc_{t_{wri}}]$), soit le rédacteur a écrit cette version chez le relecteur (opération $write_{t_{wri}}[doc_{t_{rev}}]$), et que l'objet $doc_{t_{rev}}$ n'a pas été modifié depuis. S'il y a eu une telle modification, cela signifie que la version de la revue a été produite à partir d'une version d'un document dont le rédacteur n'a pas connaissance. Par conséquent, il serait incohérent de lui envoyer cette revue.

Bien entendu, le problème est symétrique et se pose également au moment de l'importation, par le relecteur, d'un document produit par le rédacteur et basé sur les revues

précédemment rédigées par le relecteur. Ceci se traduit par la règle de coopération suivante:

$$\begin{array}{l}\forall\ doc \in Doc \quad (op \in Read_{t_{rev}}[doc_{t_{wri}}] \cup Write_{t_{wri}}[doc_{t_{rev}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\ rev \in Rev \quad \nexists\ p \in Write^{(rev)} \\ \qquad (LastOcc(Read_{t_{wri}}[rev_{t_{rev}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{rev}}[rev_{t_{wri}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op)\end{array}$$

Nous pouvons maintenant donner la définition complète du schéma de coopération rédacteur/relecteur (i.e. du prédicat *writer_reviewer*). Les deux premières règles de coopération nous permettent d'orienter les échanges de documents et de revues entre le rédacteur et le relecteur via l'utilisation de deux schémas de coopération client/serveur. Les deux dernières règles de coopération nous assurent quant à elles que les revues sont produites à partir des bonnes valeurs des documents et vice-versa.

$$\begin{array}{l}writer\_reviewer(t_{wri}, t_{rev}, Doc \cup Rev) \equiv \\ \left\{\begin{array}{l} client\_server(t_{rev}, t_{wri}, Doc) \\ \\ client\_server(t_{wri}, t_{rev}, Rev) \\ \\ \forall\ rev \in Rev \quad (op \in Read_{t_{wri}}[rev_{t_{rev}}] \cup Write_{t_{rev}}[rev_{t_{wri}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\ doc \in Doc \quad \nexists\ p \in Write^{(doc)} \\ \qquad (LastOcc(Read_{t_{rev}}[doc_{t_{wri}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{wri}}[doc_{t_{rev}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \\ \\ \forall\ doc \in Doc \quad (op \in Read_{t_{rev}}[doc_{t_{wri}}] \cup Write_{t_{wri}}[doc_{t_{rev}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\ rev \in Rev \quad \nexists\ p \in Write^{(rev)} \\ \qquad (LastOcc(Read_{t_{wri}}[rev_{t_{rev}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{rev}}[rev_{t_{wri}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \end{array}\right.\end{array}$$

Comme nous pouvons le constater, les schémas de coopération client/serveur et rédacteur/relecteur sont tous deux basés sur le schéma de coopération écriture coopérative (schéma *DisCOO*). Pour formaliser ces deux nouveaux schémas, nous avons en effet ajouté de nouvelles règles de coopération à celles du prédicat *DisCOO* de manière à définir de nouvelles contraintes sur les interactions entre les transactions. Par conséquent, ces trois schémas de coopération vérifient les règles de coopération du prédicat *DisCOO*. Ils assurent ainsi tous les trois la *DisCOO*-sérialisabilité. Cela signifie que nous pouvons utiliser ces trois schémas de coopération simultanément au sein d'un même système tout en garantissant la *DisCOO*-sérialisabilité de l'exécution au niveau global (lemme 4.8).

LEMME 4.8 (EXÉCUTION *DisCOO*-SÉRIALISABLE (BIS))
Si toutes les interactions entre les transactions sont contrôlées par un des schémas de coopération écriture coopérative, client/serveur ou rédacteur/relecteur, ce qui est représenté par l'axiome suivant

$$\forall t_i, t_j \ (t_i \neq t_j) \quad \forall t \in \{t_i, t_j\} \quad \forall O \quad (Contract[t_i, t_j, O, s] \in H_t) \Rightarrow \\ (s \in \{DisCOO, client\_server, writer\_reviewer\})$$

alors l'exécution est *DisCOO*-sérialisable.

Preuve :

La preuve de ce lemme est identique à celle du lemme 4.7 en utilisant également le fait que les schémas de coopération client/serveur et rédacteur/relecteur sont fondés sur le schéma de coopération écriture coopérative (schéma *DisCOO*), ce qui se traduit de la manière suivante au niveau des prédicats *DisCOO*, *client_server* et *writer_reviewer*:

$$client\_server(t_i, t_j, O) \Rightarrow DisCOO(t_i, t_j, O)$$
$$writer\_reviewer(t_i, t_j, O) \Rightarrow DisCOO(t_i, t_j, O)$$

□

Bilan

Nous venons de présenter les notions de "schéma de coopération" et de "négociation d'un schéma de coopération entre deux transactions pour le partage d'un ensemble d'objets". Les différentes transactions du système ne sont plus obligées de toutes respecter exactement les mêmes règles de coopération pour contrôler leurs interactions, mais peuvent au contraire les négocier au coup par coup en accord avec leurs partenaires.

L'exemple que nous avons développé dans [Ben98b] présente le cas d'une entreprise virtuelle dans laquelle les différents acteurs du système s'échangent des données selon des règles de coopération différentes. Toutefois, les "différents" schémas utilisés dans cet exemple (client/serveur, rédacteur/relecteur, écriture coopérative) correspondent tous les trois à un unique critère de correction (la *DisCOO*-sérialisabilité) auquel ont été ajoutées certaines contraintes (échanges unidirectionnels dans le cas du schéma client/serveur par exemple). Ces schémas ne sont donc pas fondamentalement différents du point de vue des propriétés garanties. Comme nous l'avons expliqué ci-dessus, il est d'ailleurs possible de les utiliser simultanément au sein d'un même système tout en garantissant que l'exécution obtenue est *DisCOO*-sérialisable.

D'autres comportements coopératifs ont cependant été étudiés dans [Can98b] et nous ont permis de mettre en évidence de nouveaux schémas de coopération (sessions alternatives, sessions mutuelles, ...). Ces schémas utilisant des règles de coopération différentes (et parfois même contradictoires), quelles sont les conséquences de cette hétérogénéïté? Nous sommes en fait confrontés à deux problèmes: déterminer les **interactions** entre les différents schémas de coopération et étudier le **comportement global** du système.

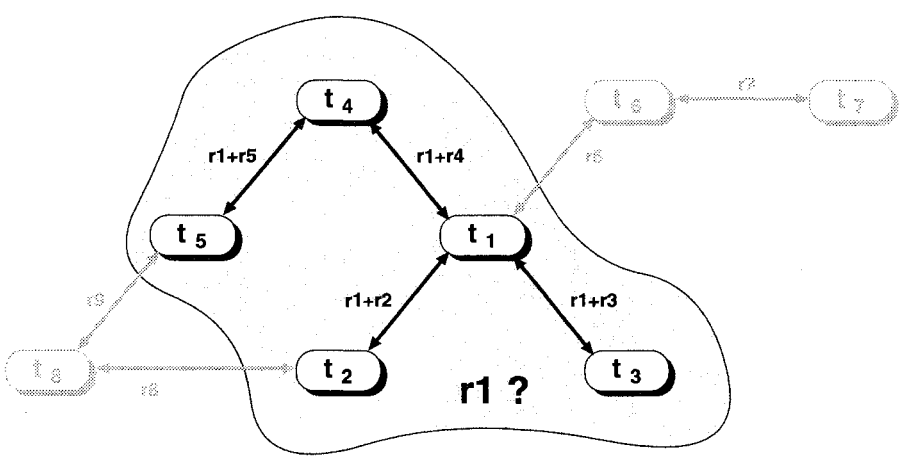
4.3.2 Comportement Global du Système

Le mécanisme de négociation de schémas de coopération permet aux différentes transactions du système de partager des objets en utilisant des règles de coopération éventuellement différentes pour chacun d'eux. Cela nous évite ainsi de devoir définir un seul et unique schéma de coopération qui serait utilisé par toutes les transactions pour le partage de tous leurs objets. Toutefois, quel est maintenant le comportement global du système? En d'autres termes, puisque les interactions entre transactions ne sont pas toutes soumises aux mêmes règles, quelles sont les propriétés que nous pouvons garantir entre deux transactions quelconque du système?

L'objectif n'est bien évidemment pas de reconstruire le critère de correction global du système puisque nous avons justement défini la notion de négociation de schémas de coopération entre les transactions de manière à éviter la définition d'un tel critère de correction global. L'idée est plutôt de mettre en évidence des propriétés qui seraient vérifiées sur un ensemble de transactions donné, c'est à dire des propriétés qui seraient garanties entre deux transactions quelconques de cet ensemble (et n'ayant éventuellement aucune interaction directe entre elles).

Nous pouvons reformuler le problème de la façon suivante: *"Si une règle de coopération est vérifiée par tous les échanges de données au sein d'un ensemble de transactions, cette règle est-elle également vérifiée entre deux transactions quelconques de cet ensemble?"*

Figure 4.10 Propriété Locale $\rightsquigarrow$ Propriété "Globale"



Prenons l'exemple illustré par la figure 4.10. Dans ce système, un certain nombre de transactions ($t_1..t_5$) coopèrent en respectant, entre autres, la règle de coopération r_1. Si nous considérons cette partition du réseau (représentée par la partie grisée) dans laquelle tous les schémas de coopération négociés vérifient la règle r_1, peut-on affirmer que cette règle de coopération est respectée "globalement" sur cette partition, i.e. entre deux transactions quelconques de cette partition? Supposons par exemple que la règle r_1 soit le prédicat *DSeria* et prenons les transactions t_3 et t_5. Si ces deux transactions sont amenées à se partager les objets d'un ensemble O (indirectement, via les transactions t_1 et t_4), le prédicat $DSeria(t_3, t_5, O)$ est-il vérifié? En d'autres termes, les transactions t_3 et t_5

sont-elles sérialisables l'une par rapport à l'autre en ce qui concerne leurs "interactions" relatives aux objets $ob \in O$?

Comme nous l'avons expliqué dans la section 4.1.2 (lemme 4.1) lors de la définition des opérations de transfert, une interaction entre deux transactions t_i et t_j est représentée de la manière suivante:

$$\exists\, p, q \quad \exists\, ob \quad (p_{t_i}[ob_{t_{k_1}}] \rightarrow q_{t_j}[ob_{t_{k_n}}]) \ \wedge\ conflict(p_{t_i}[ob], q_{t_j}[ob]) \ \wedge\ (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_n}})$$

Cela signifie que nous avons eu un enchaînement d'opérations tel que présenté ci-dessous dans lequel les transactions t_{l_i} nous ont permis de "propager", de proche en proche, la valeur de l'objet $ob_{t_{k_1}}$ vers la transaction t_{k_n}, rendant ainsi l'opération $q_{t_j}[ob_{t_{k_n}}]$ dépendante (du point de vue de la valeur de retour) de l'opération $p_{t_i}[ob_{t_{k_1}}]$. Du point de vue du graphe des transactions, nous avons donc un chemin de la transaction t_i vers la transaction t_j dans lequel deux opérations successives sont conflictuelles:

$$p_{t_i}[ob_{t_{k_1}}] \rightarrow \underbrace{q_{t_{l_1}}[ob_{t_{k_1}}] \rightarrow p_{t_{l_1}}[ob_{t_{k_2}}]}_{ob_{t_{k_1}} \rightsquigarrow ob_{t_{k_2}}} \rightarrow \underbrace{q_{t_{l_2}}[ob_{t_{k_2}}] \rightarrow p_{t_{l_2}}[ob_{t_{k_3}}]}_{ob_{t_{k_2}} \rightsquigarrow ob_{t_{k_3}}} \rightarrow \ldots \rightarrow \underbrace{q_{t_{l_{n-1}}}[ob_{t_{k_{n-1}}}] \rightarrow p_{t_{l_{n-1}}}[ob_{t_{k_n}}]}_{ob_{t_{k_{n-1}}} \rightsquigarrow ob_{t_{k_n}}} \rightarrow q_{t_j}[ob_{t_{k_n}}]$$

Supposons maintenant que nous ayons une règle de coopération *rule* qui soit vérifiée par tous les schémas de coopération négociés par les couples de transactions (t_i, t_{l_1}), (t_{l_1}, t_{l_2}), $\ldots$, $(t_{l_{n-1}}, t_j)$ pour le partage de l'objet *ob*. Si cette règle *rule* est **transitive**, i.e.

$$rule(t_1, t_2, \{ob\}) \ \wedge\ rule(t_2, t_3, \{ob\}) \ \Rightarrow\ rule(t_1, t_3, \{ob\})$$

alors nous pouvons en déduire que la règle *rule* est également vérifiée entre les transactions t_i et t_j. C'est le cas notamment des règles de coopération qui imposent un ordre entre les événements du système (`Begin`/`Commit`/`Abort` d'une transaction, invocations d'opérations sur les objets, ...) telles que le prédicat *DSeria* (une transaction ne peut être validée qu'une fois que toutes les transactions dont elle dépend ont été validées) ou le prédicat *DisCOO* (une transaction doit être *up_to_date* avant d'être validée, ...). La notion de propriété (règle de coopération) globale peut ainsi être définie par le lemme 4.9 dont nous venons de présenter la preuve.

Exemples de Propriétés Transitives

Parmis les règles de coopération transitives, nous pouvons citer deux exemples que sont les prédicats *DSeria* et *DisCOO*. Comme nous l'avons expliqué au paragraphe 4.3.1, ces deux schémas de coopération ont d'ailleurs été définis en conséquence. Il s'agissait effectivement d'obtenir des propriétés locales (la *D*-sérialisabilité et la *DisCOO*-sérialisabilité) qui, lorsqu'elles étaient vérifiées au niveau de chaque transaction du système, garantissaient implicitement la propriété globale équivalente (respectivement la sérialisabilité et la *COO*-sérialisabilité) au niveau du système complet. Les preuves qui nous ont permis d'établir que ces critères de corrections locaux garantissaient les mêmes propriétés que leurs homologues "globaux" étaient d'ailleurs basées sur la transitivité de ces propriétés.

LEMME 4.9 (RÈGLE DE COOPÉRATION GLOBALE)
Soit T un ensemble de transactions.
Soit *ob* un objet partagé par les transactions de cet ensemble.
Soit *rule* une règle de coopération **transitive.**

Si la règle *rule* est vérifiée pour l'objet *ob* par tous les schémas de coopération négociés par tous les couples (t_i, t_j) de transactions de l'ensemble T, i.e.

$$\forall t_i, t_j \in T \quad (Contract[t_i, t_j, O, s] \in H) \wedge (ob \in O) \quad \Rightarrow \quad (\, s(t_i, t_j, O) \Rightarrow rule(t_i, t_j, \{ob\})\,)$$

alors cette règle *rule* est également vérifiée pour l'objet *ob* entre deux transactions quelconques de l'ensemble T, i.e.

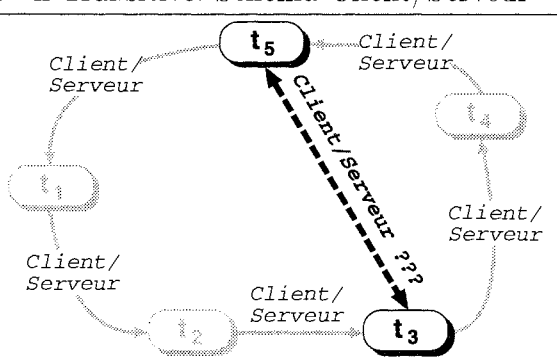
$$\forall t_1, t_2 \in T \quad rule(t_1, t_2, \{ob\})$$

La règle de coopération *rule* est alors dite *globale* sur l'ensemble de transactions T pour l'objet *ob.*

Exemples de Propriétés Non Transitives

Le schéma de coopération client/serveur (prédicat *client_server* défini au paragraphe 4.3.1) constitue quant à lui un exemple de propriété non transitive. Supposons en effet que nous ayons la situation décrite figure 4.11. Il s'agit de plusieurs transactions (t_1 à t_5) impliquées dans un cycle de relations client/serveur pour le partage d'un même objet *ob.* Par conséquent, la propriété "client/serveur" est assurée sur tous les échanges de données entre ces transactions.

Figure 4.11 Propriété Non Transitive: Schéma Client/Serveur



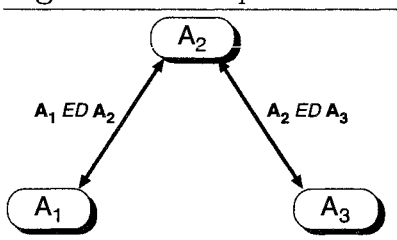
Nous ne pouvons cependant pas en déduire que cette propriété est garantie entre deux transactions quelconques de cet ensemble. Si nous prenons le cas des transactions t_3 et t5 par exemple, il est évident que leurs interactions (indirectes) ne respectent pas le schéma client/serveur pour le partage de l'objet *ob.* La transaction t_3 peut en effet avoir connaissance de la valeur de l'objet *ob* produite par la transaction t_5 (via les transactions t_1 et t_2). De même, la transaction t_5 peut également accéder à la valeur de l'objet *ob* produite par la transaction t_3 (via la transaction t_4).

$$client\_server(t_c, t_s, O) \equiv \begin{cases} DisCOO(t_c, t_s, O) \\ \forall ob \in O \quad \nexists\, p \in Write^{(ob)} \quad \text{tel que} \quad p_{t_c}[ob_{t_s}] \in H \\ \forall ob \in O \quad \nexists\, q \in Read^{(ob)} \quad \text{tel que} \quad q_{t_s}[ob_{t_c}] \in H \end{cases}$$

Par conséquent, parmis les trois règles de coopération garanties par le schéma de coopération client/serveur , seule la première (le prédicat $DisCOO$) est vérifiée globalement sur cet ensemble de transactions. Les deux autres règles, dont le rôle est de contrôler l'orientation des échanges, ne sont quant à elles pas transitives.

Il existe également d'autres règles de coopération qui ne sont pas transitives. C'est la cas par exemple de la dépendance d'exclusion (cf. figure 4.12) qui permet de représenter le fait qu'entre deux activités, au plus une des deux pourra être validée. Comme nous le voyons sur cet exemple, le fait d'avoir A_1 $\mathcal{ED}$ A_2 et A_2 $\mathcal{ED}$ A_3 ne nous donne pas A_1 $\mathcal{ED}$ A_3. Supposons en effet que A_1 soit validée. Du fait de la relation $\mathcal{ED}$, l'activité A_2 doit donc être annulée. Or ceci autorise la validation de l'activité A_3. Il est ainsi possible que A_1 et A_3 soient toutes deux validées. Par conséquent, nous n'avons pas A_1 $\mathcal{ED}$ A_3.

Figure 4.12 Propriété Non Transitive: Dépendance d'Exclusion



Exclusion Dependency (t_i $\mathcal{ED}$ t_j)

Si l'activité t_i est validée et si l'activité t_j est démarrée, alors t_j doit être annulée (les activités t_i et t_j ne peuvent pas être validées toutes les deux), i.e.:

$(\texttt{Commit}_{t_i} \in H) \Rightarrow ((\texttt{Begin}_{t_j} \in H) \Rightarrow (\texttt{Abort}_{t_j} \in H))$

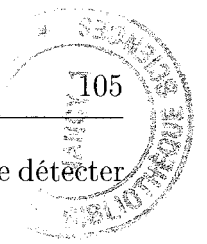
Bilan

La transitivité est une condition nécessaire pour nous permettre de définir, au sein d'un réseau d'activités, les partitions de ce réseau sur lesquelles une règle de coopération donnée est "globalement" vérifiée. Ainsi, à défaut de pouvoir définir **LE** comportement global du système, nous pouvons tout de même identifier certaines règles de coopérations (propriétés) garanties sur des sous-ensembles de ce système.

4.3.3 Interactions entre Schémas

Le fait d'utiliser plusieurs schémas de coopération simultanément dans un même système peut également provoquer des interactions entre ces différents schémas. En effet, un schéma de coopération définit des contraintes (les règles de coopération) sur les histoires des activités. Etant donné que ces contraintes sont utilisées en tant que préconditions aux événements du système[32], il est donc possible qu'en multipliant les contraintes (par l'utilisation simultanée de plusieurs schémas) on se retrouve dans une situation où tout nouvel événement serait refusé par au moins une des contraintes.

32. Lors de l'invocation d'un nouvel événement (opération sur un objet ou événement significatif), si une des contraintes n'est pas vérifiées, alors cet événement est refusé, i.e. il n'est pas exécuté.



L'objectif de cette section est de présenter de quelle manière nous proposons de détecter et de gérer de tels conflits entre différents schémas de coopération.

Détection des Conflits entre Schémas

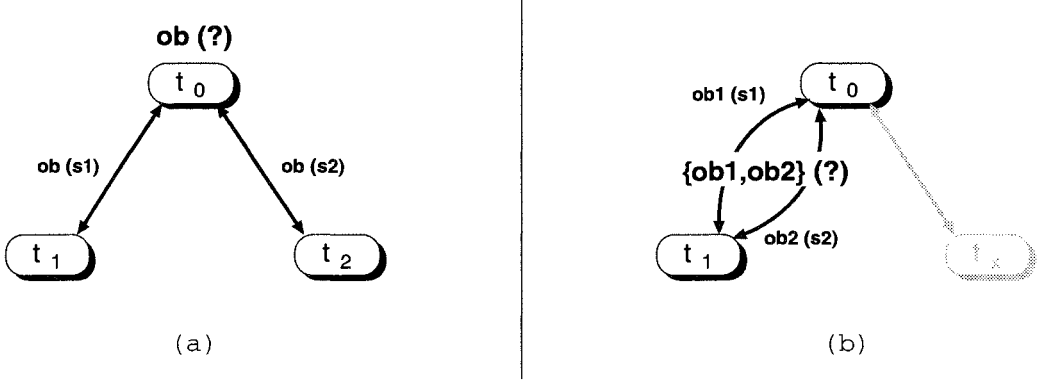
DÉFINITION 4.18 (INTERACTION ENTRE SCHÉMAS DE COOPÉRATION)
Soient t_0, t_1 et t_2 trois transactions (avec éventuellement $t_1 = t_2$).
Soit s_1 le schéma de coopération négocié entre t_0 et t_1 pour le partage de l'objet ob_1.
Soit s_2 le schéma de coopération négocié entre t_0 et t_2 pour le partage de l'objet ob_2.
(On peut éventuellement avoir $s_1 = s_2$ et/ou $ob_1 = ob_2$)

Il existe un conflit entre les schémas de coopération s_1 et s_2 lorsqu'à un instant donné nous avons soit $s_1(t_0, t_1, \{ob_1\}) \wedge \neg s_2(t_0, t_2, \{ob_2\})$ soit $\neg s_1(t_0, t_1, \{ob_1\}) \wedge s_2(t_0, t_2, \{ob_2\})$.

En premier lieu, qu'est ce qu'une interaction entre deux schémas de coopération? Un schéma de coopération a été défini comme étant un ensemble de règles de coopération, chacune de ces règles étant un prédicat à évaluer sur les histoires locales des transactions. Par conséquent, une interaction entre deux schémas de coopération signifie qu'à un instant donné l'exécution d'une transaction (représentée par son histoire locale) est correcte par rapport à l'un des schémas (i.e. à toutes ses règles) alors qu'une des règles de l'autre schéma n'est pas vérifiée sur cette histoire (définition 4.18).

Pour illustrer ceci, nous présentons ci-dessous deux exemples d'interactions entre schémas de coopération: les interactions **"intra-transaction"** (figure 4.13-a) et les interactions **"inter-transactions"** (figure 4.13-b). Dans le premier cas, les interactions ont lieu au niveau d'une seule transaction, alors que dans le deuxième cas les interactions interviennent entre deux transactions.

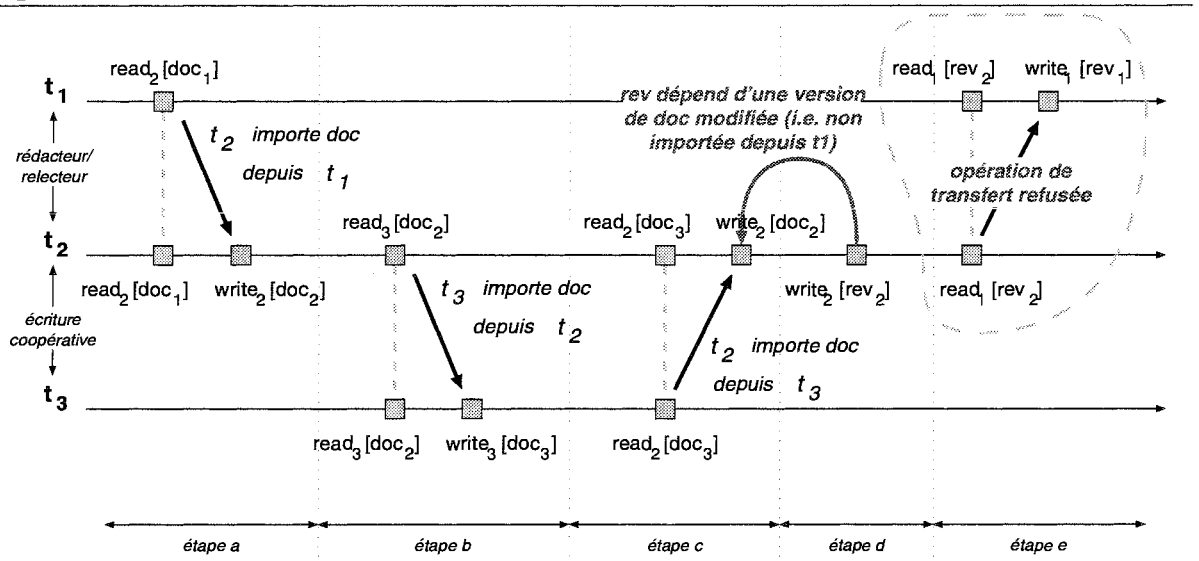
Figure 4.13 Interactions entre Schémas de Coopération



- **Interactions intra-transaction :** Une transaction t_0 partage un même objet ob avec deux autres transactions t_1 et t_2 en utilisant des schémas de coopération s_1 et s_2 éventuellement différents (figure 4.13-a). Quelles sont alors, au niveau de la

transaction t_0, les règles qui contrôlent les accès des autres transactions à l'objet *ob*? Il se peut également que certaines règles de coopération assurées par les schémas s_1 et s_2 soient "incompatibles" et produisent un verrou mortel (i.e. ni la transaction t_1 ni la transaction t_2 ne peut plus accéder à l'objet *ob*).
Prenons par exemple le cas représenté sur la figure 4.14. Les transactions t_1 et t_2 ont négocié le schéma de coopération "rédacteur/relecteur" pour le partage des objets *doc* et *rev*: la transaction t_1 rédige le document *doc*, lequel est relu par la transaction t_2 qui renvoie son avis *rev*. De son côté, la transaction t_2 coopère également avec la transaction t_3. Elles utilisent pour cela le schéma de coopération "écriture coopérative" pour partager l'objet *doc*.

Figure 4.14 Exemple de Conflit Intra-Transaction



(a) La transaction t_2 importe l'objet *doc* depuis la transaction t_1.
(b) La transaction t_3 importe l'objet *doc* depuis la transaction t_2.
(c) La transaction t_2 importe une nouvelle version de l'objet *doc* produite par la transaction t_3.
(d) La transaction t_2 rédige son avis *rev*.
(e) Une interaction entre les deux schémas de coopération négociés interviendra dès que la transaction t_1 tentera d'importer la revue *rev* de la transaction t_2 puisque cette version de l'objet *rev* aura été produite à partir d'une version modifiée (par rapport à la transaction t_1) de l'objet *doc* (cf. définition du prédicat *writer_reviewer* au paragraphe 4.3.1).

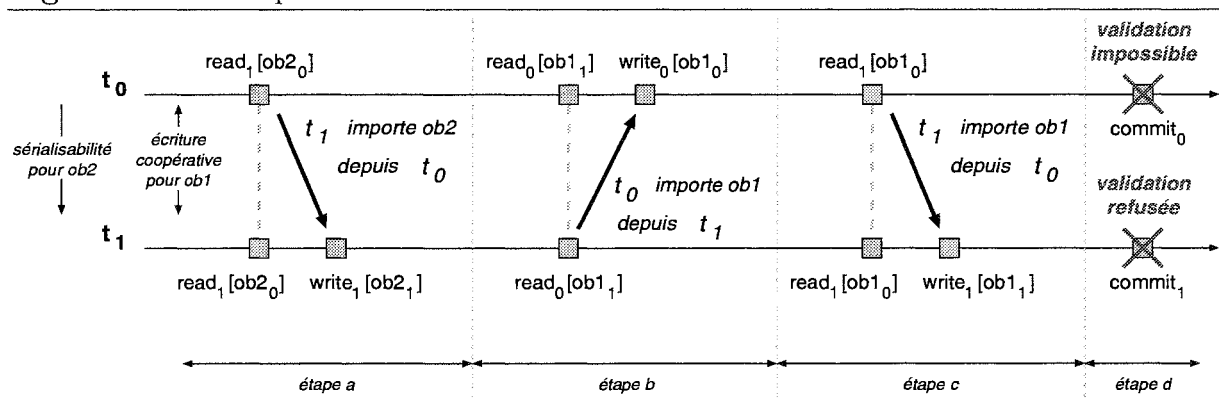
Remarque : Si nous avions défini le prédicat *writer_reviewer* de telle sorte que le relecteur (i.e. la transaction t_2) ne soit pas autorisé à modifier les documents provenants du rédacteur (dans notre cas: l'objet *doc* importé de la transaction t_1), l'interaction entre les deux schémas aurait eu lieu dès que la transaction t_2 aurait voulu importer l'objet *doc* de la transaction t_3 (étape c).

- **Interactions inter-transactions :** Deux transactions t_0 et t_1 partagent deux objets $ob1$ et $ob2$ en utilisant des schémas de coopération s_1 et s_2 éventuellement différents (figure 4.13-b). Il est fort probable qu'il existe des exécutions qui soient correctes vis-à-vis de l'un des schémas et pas vis-à-vis de l'autre. Il est en effet possible que certaines règles de coopération garanties par les schémas s_1 et s_2 soient en contradiction sur certains points et puissent dans certains cas générer des verrous mortels tels que:

  - Tous les échanges entre les transactions t_0 et t_1 concernant les objets $ob1$ ou $ob2$ sont refusés.
  - Aucune des transactions t_0 et t_1 ne peut être validée.

  Comme exemple, il suffit de considérer le cas décrit sur la figure 4.15 où les deux transactions t_0 et t_1 coopèrent sur l'objet $ob1$ en utilisant le mode "écriture coopérative" (schéma *DisCOO*) et sur l'objet $ob2$ en utilisant le mode "sérialisabilité" (schéma *DSeria*).

Figure 4.15 Exemple de Conflit Inter-Transactions



(a) La transaction t_1 importe l'objet $ob2$ depuis la transaction t_0. Du point de vue du schéma de coopération *DSeria*, la transaction t_0 devra absolument être validée avant la transaction t_1.

(b) La transaction t_0 importe l'objet $ob1$ depuis la transaction t_1.

(c) La transaction t_1 importe l'objet $ob1$ depuis la transaction t_0. Les transactions t_0 et t_1 deviennent ainsi interdépendantes en ce qui concerne l'objet $ob1$. Le schéma de coopération *DisCOO* obligera donc les transactions t_0 et t_1 à être validées en même temps.

(d) La validation de la transaction t_1 est refusée par le schéma de coopération *DSeria* puisque la transaction t_0 n'est pas encore validée. Or il est impossible de valider la transaction t_0 puisque cela nécessite également de valider la transaction t_1 (schéma de coopération *DisCOO*).

Remarque : Il aurait été possible de prévenir et éviter ce problème à l'étape b lorsque la transaction t_0 a importé l'objet $ob1$ depuis la transaction t_1. Le schéma

de coopération *DSeria* aurait en effet pu détecter qu'une telle opération allait développer un "cycle" de dépendance entre les deux transactions et donc refuser cette opération de transfert. Cela aurait toutefois signifié qu'une opération concernant l'objet *ob*1 aurait été refusée par le schéma de coopération *DSeria* négocié entre t_0 et t_1 pour le partage de l'objet *ob*2.

Dans un cas comme dans l'autre, il est donc nécessaire de définir une politique de détection et (surtout) de gestion de ces conflits. Afin de préserver l'autonomie des activités, il est nécessaire que chacune d'elles soit elle-même responsable de la détection des conflits au sein de son référentiel local. Elle peut, pour cela, procéder de deux façons: soit elle s'assure, lorsqu'elle les négocie, que tous les schémas de coopération qu'elle utilise avec d'autres activités sont "compatibles", soit elle effectue la vérification au moment de l'exécution:

- Détection lors de la **négociation**: Il est nécessaire dans ce cas de tester, au niveau d'une transaction, si les spécifications de deux schémas de coopération sont compatibles, i.e. que toutes les exécutions acceptées par un des schémas sont également considérées comme correctes par l'autre schéma. Du point de vue formel, cela revient à tester la consistance de l'union de deux spécifications (union de deux ensembles d'axiomes). Outre la complexité de ce test due aux problèmes d'indécidabilité inhérents à ce genre d'exercice, cela peut mettre un frein à la coopération entre les transactions. En effet, il est possible que deux transactions désirant coopérer en utilisant un certain schéma ne puissent pas négocier ce schéma car celui-ci, dans certains cas extrêmement rares, pourrait entrer en conflit avec d'autres schémas déjà négociés par l'une ou l'autre de ces deux transactions.
- Détection à l'**exécution**: Lors de la phase de négociation d'un schéma de coopération entre deux transactions, aucune importance n'est accordée à d'éventuels autres schémas déjà utilisés par ces deux transactions. Les conflits, si conflit il y a, seront détectés à l'exécution au moment des échanges. Il y a alors deux politiques possibles lorsqu'une transaction t_0 partage un même document avec plusieurs autres activités $t_1..t_n$ en utilisant des schémas de coopération $s_1..s_n$:
  - Politique **pessimiste**: Chaque échange $t_0 \leftrightarrow t_i$ doit vérifier tous les schémas de coopération $s_1..s_n$. Cela signifie cependant que la transaction t_i pourra être perturbée (au niveau de ses échanges avec la transaction t_0) par un schéma autre que celui qu'elle aura négocié avec t_0 (i.e. le schéma s_i).
    Toutefois, si nous considérons l'exemple d'interaction intra-transaction que nous avons présenté ci-dessus, nous remarquons que l'intégrité des données ne sera pas obligatoirement toujours assurée. Un conflit entre deux schémas peut en effet n'être détecté que tardivement. Pour remédier à ce problème, il faudrait définir des schémas de coopération plus stricts, mais cela reviendrait finalement à prévoir à l'avance toutes les interactions possibles, ce qui serait contraire à nos objectifs.
  - Politique **optimiste**: Un échange $t_0 \leftrightarrow t_i$ n'est contrôlé que par le schéma s_i. Les échanges $t_0 \leftrightarrow t_i$ respectent donc strictement le schéma de coopération qui

aura été négocié entre les transactions t_0 et t_i. Si un échange $t_0 \leftrightarrow t_i$ génère un conflit au sens du schéma s_j $(i \neq j)$, ce conflit ne sera détecté que lorsque la transaction t_0 tentera un nouvel échange avec t_j. La transaction t_0 est ainsi la seule à avoir connaissance de ce conflit.

Quelle que soit la méthode adoptée pour détecter un conflit, nous constatons qu'effectivement seule la transaction au niveau de laquelle se produit ce conflit est responsable de sa résolution. Le choix de l'une ou l'autre de ces méthodes permet simplement de contrôler (limiter) les perturbations que pourra générer un tel conflit sur les autres transactions du système.

Gestion des Conflits entre Schémas

Deux transactions quelconques peuvent coopérer en s'échangeant des objets au cours de leur exécution. Afin de coordonner ces communications, elles doivent au préalable négocier un schéma de coopération. Elles sont alors assurées que leurs échanges respecteront ce schéma tout en restant indépendantes l'une de l'autre en ce qui concerne leur tâche: en cas de violation d'une des règles de ce schéma, seule l'opération d'échange est refusée. Aucune contrainte sur la tâche elle-même n'est imposée par d'autres activités. Toutefois, si une transaction coopère avec plusieurs autres transactions (en utilisant éventuellement des schémas de coopération différents), elle peut se retrouver dans une situation où toutes les opérations d'échange qu'elle tentera violeront au moins une des règles d'un des schémas de coopération négociés et seront donc refusées. Par conséquent, puisque nous voulons absolument éviter de devoir abandonner une transaction, nous devons fournir aux différentes transactions des mécanismes permettant de prévenir et de gérer ces conflits entre schémas de coopération.

Par "prévenir" les conflits, nous entendons le fait de vérifier qu'une opération invoquée entre deux transactions, bien qu'étant acceptée par le schéma de coopération négocié par ces deux transactions, n'est pas une source potentielle de conflit vis-à-vis d'un autre schéma de coopération négocié par l'une de ces deux transactions. Reprenons l'exemple de conflit intra-transaction que nous avons présenté au paragraphe précédent en supposant que le schéma de coopération "rédacteur/relecteur" négocié entre les transactions t_1 et t_2 soit srict (c'est-à-dire que la transaction t_2 n'a pas le droit de modifier l'objet *doc* importé de t_1 et que la transaction t_1 n'est pas autorisée à modifier la revue *rev* rédigée par t_2) et que la détection des conflits soit réalisée à l'exécution avec une politique optimiste (i.e. lors d'un échange de données entre deux transactions, celles-ci ne vérifient que le schéma de coopération qu'elles ont négocié). Alors, quand la transaction t_2 décide d'importer la nouvelle version de l'objet *doc* produite par la transaction t_3, bien que cette opération soit correcte vis-à-vis du schéma de coopération "écriture coopérative" négocié entre t_2 et t_3, il serait tout de même intéressant d'informer la transaction t_2 que cette opération risque toutefois de provoquer un conflit avec le schéma de coopération "rédacteur/relecteur" (strict) qu'elle a négocié avec la transaction t_1.

Toutefois, comme nous l'avons expliqué au paragraphe précédent, il n'est pas toujours possible de prévoir qu'une opération acceptée à un instant donné sera ultérieurement la source d'un conflit (cas du schéma de coopération "rédacteur/relecteur" non strict par

exemple). Par conséquent, nous devons également proposer des mécanismes pour résoudre de tels conflits:

- **Re-négociation** des schémas de coopération: Une situation de blocage peut être due au fait que le schéma qui a été négocié entre deux transactions a été défini de manière trop restrictive. Les transactions doivent donc avoir la possibilité de re-négocier le schéma de coopération utilisé pour leurs échanges (que ce soit pour "l'assouplir" ou au contraire "le durcir").

  Dans le cas du conflit intra-transaction présenté ci-dessus, une solution serait effectivement que les deux transactions t_1 et t_2 re-négocient le schéma de coopération "rédacteur/relecteur" en schéma "écriture coopérative", ce qui permettrait à la transaction t_1 d'importer les mises-à-jours effectuées sur l'objet *doc* par la transaction t_2 (et indirectement par la transaction t_3).

- **Gestion des versions** des objets: Il peut être intéressant de permettre à une transaction de "revenir en arrière" pour revenir dans une situation cohérente. Cela signifie qu'elle a la possibilité soit d'annuler (à éviter) soit de compenser certaines étapes de son travail. Cela nécessite toutefois que les référentiels locaux utilisés par les transactions soient capables de gérer les versions successives des objets.

  Sur notre exemple de conflit intra-transaction, la transaction t_2 pourrait ainsi rédiger la revue *rev* destinée à la transaction t_1 en faisant référence non pas à la version modifiée de l'objet *doc*, mais à la version de l'objet *doc* qu'elle a importée de la transaction t_1. De cette manière, quand la transaction t_1 importera cette revue *rev*, la cohérence des objets *doc* et *rev* imposée par le schéma de coopération "rédacteur/relecteur" sera assurée.

Bilan

Comme nous pouvons le constater, permettre l'utilisation de schémas de coopération différents pour contrôler les échanges de données entre transactions au sein d'un même système peut provoquer des interactions entre ces différents schémas, voire même des conflits. De plus, étant donné que les schémas de coopération ne sont pas censés prévoir toutes ces interactions (car cela reviendrait finalement à définir un seul et unique schéma), certains conflits peuvent n'être détectés que tardivement.

Pour débloquer une situation de conflit nous avons proposé deux solutions. La solution du retour à une ancienne version suppose simplement que les espaces de coopération fournissent des fonctionnalités de gestion des versions des objets partagés. Par contre, la solution de la re-négociation du schéma de coopération nécessite de formaliser l'opération de re-négociation (schémas pouvant être re-négociés, conditions à remplir avant de re-négocier, conséquences de la re-négociation sur les conflits ultérieurs et les techniques de recouvrement, ...). Nous ne détaillerons pas d'avantage ce concept de re-négociation qui fait partie des perspectives aux travaux présentés dans cette thèse.

4.4 Conclusion

Nous avons présenté dans ce chapitre un nouveau modèle de transactions avancé supportant les exécutions de transactions distribuées géographiquement, coopérant par échanges de résultats intermédiaires et négociant les règles de coopération à respecter lors de ces échanges. Le problème de la coopération (lié à la propriété d'isolation des transactions) avait déjà été abordé dans le cadre du projet *COO*. Toutefois, le modèle des *COO*-transactions obtenu, à l'image des systèmes transactionnels classiques, supposait que les transactions interagissaient via l'utilisation d'un référentiel commun et que ces interactions étaient toutes contrôlées par un unique critère de correction (la *COO*-sérialisabilité en l'occurence). L'objectif de cette thèse était donc de définir un nouveau modèle de transactions avancé qui, outre la coopération au sens *COO*, prenne également en compte les aspects liés à:

- la **distribution des objets** partagés: Chaque transaction possède dorénavant sa propre copie des objets qu'elle manipule. Les transactions coopèrent ainsi par des échanges de données explicites entre leurs référentiels locaux. Nous avons pour cela enrichi le formalisme ACTA utilisé pour décrire notre modèle de transaction et défini la notion d'opération de transfert de manière à identifier et synchroniser les différentes copies d'un même objet logique.
- la **distribution du contrôle de la concurrence**: Afin de rendre les transactions les plus autonomes possible, la coordination de leurs échanges est réalisée localement. Cela signifie que chaque transaction est responsable de la consistance des données contenues dans son référentiel local. Nous avons à cet effet défini de nouveaux critères de correction locaux (la *D*-sérialisabilité et la *DisCOO*-sérialisabilité) qui, lorsqu'ils sont assurés au niveau de chaque transaction, garantissent les mêmes propriétés au niveau du système complet que leurs homologues "globaux" classiques (la sérialisabilité et la *COO*-sérialisabilité).
- l'**hétérogénéïté des relations de coopération** entre les différentes transactions: Dans le cadre des applications coopératives distribuées, toutes les activités ne sont pas forcément guidées par les mêmes règles de coopération. Nous avons donc défini la notion de négociation d'un schéma de coopération entre deux transactions pour le partage d'un ensemble d'objets donné. Nous avons ensuite étudié de quelle manière ces différents schémas pouvaient cohabiter au sein d'un même système.

Cette approche transactionnelle du problème de la coopération dans le cadre des applications distribuées nous permet donc de simplifier leur programmation en cachant au maximum la complexité introduite par l'exécution concurrente des différentes activités. La programmation de telles applications peut ainsi être considérée comme un assemblage d'activités autonomes, les connexions entre ces activités représentant les schémas de coopération à respecter lors de leurs échanges de données.

Chapitre 5

Services de Coopération

Table des matières

Au chapitre précédent nous avons présenté de manière formelle notre nouveau modèle de transactions coopératives distribuées, des versions distribuées de critères de correction tels que la sérialisabilité et la *COO*-sérialisabilité, ainsi que la notion de négociation de schémas de coopération et les problèmes d'interactions entre plusieurs schémas. Le chapitre 5 est quant à lui consacré à la mise en œuvre de ces modèles de transactions et de contrôle de la concurrence. Notre objectif est de réaliser une infrastructure intégrant les mécanismes de base nécessaires à la coopération. Cette infrastructure servira ensuite de base au développement d'applications coopératives distribuées (à large échelle) et hétérogènes.

Ce chapitre est constitué de deux parties: l'architecture générale de notre système de coopération et sa mise en œuvre sous forme de services de base pour la coopération. Dans la première partie nous présentons les différents composants d'une activité ainsi que leurs rôles respectifs lors des échanges de données entre les activités. Cela nous permet également d'expliquer le fonctionnement du contrôle réalisé localement et fondé sur les schémas de coopération négociés par les activités. La seconde partie concerne le prototype *DisCOO* que nous avons développé et les choix technologiques que nous avons effectués (plate-forme logicielle, langages de programmation, ...).

5.1 Architecture

Afin d'illustrer l'architecture générale de notre système de coopération nous utiliserons l'exemple développé dans [Ben98b]. Nous commencerons donc par présenter cet exemple support qui met en évidence les relations de coopération entre les différents partenaires d'une entreprise-projet dans le domaine du bâtiment.

Nous définirons ensuite les différents composants d'une activité, à savoir: de quelle manière sont stockées ses données, comment un utilisateur peut y accéder, qui contrôle les interactions avec les autres activités, ... Nous présenterons en particulier les tables de coopération qui permettent, pour chaque activité, de mémoriser les différents schémas de coopération négociés avec les autres activités du système.

Une fois les aspects structurels de notre architecture présentés, nous pourrons ensuite expliquer le fonctionnement de ces différents composants et leurs rôles respectifs lors des échanges de données entre les activités, de la phase de terminaison des activités (ex: détection de cycles dans le graphe de dépendances entre activités), ou encore lors de la détection et de la résolution des conflits entre schémas de coopération.

5.1.1 Présentation de l'Exemple Support

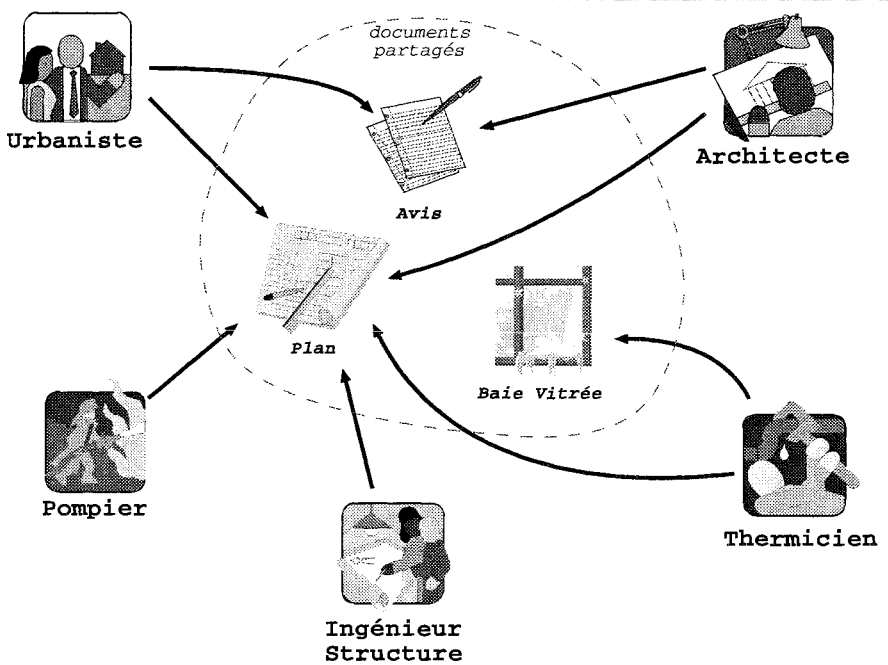
L'exemple sur lequel nous allons nous baser est celui développé dans [Big98, Ben98b] (lui-même inspiré de celui présenté dans [Ros96]). L'application considérée a pour objectif la conception d'un appartement sur un niveau ayant une salle de séjour et dont l'un des côtés est constitué intégralement d'une baie vitrée. Plusieurs partenaires interviennent lors de cette conception, formant ainsi une entreprise-projet. Ceux-ci travaillent sur trois documents qu'ils sont amenés à s'échanger: le **plan** rédigé conjointement par l'architecte et l'ingénieur-structure, l'**avis** de l'urbaniste et, dans une moindre mesure sur cet exemple, les spécifications de la **baie** vitrée définies par le thermicien. Dans cet exemple, pour des raisons de simplification, nous représenterons chaque partenaire par une activité (figure 5.1).

- l'**architecte** : Il s'occupe de la conception du bâtiment en termes de disposition des murs (disposition et taille des pièces), d'emplacements de fenêtres (luminosité), ..., c'est-à-dire de dessiner le plan du bâtiment en ne considérant que les aspects volume, espace et luminosité des appartements.
- l'**ingénieur-structure** : Son activité consiste à spécifier les éléments structurels de l'appartement et à garantir la stabilité de la construction. De tels éléments (murs porteurs, poutres ou colonnes de soutènement, ...) seront choisis de manière à respecter le plus possible l'harmonie et les choix de l'architecte.
- le **thermicien** : Il est en charge de la climatisation du bâtiment (chauffage, isolation thermique, ...). Nous limitons son intervention au choix du type de baie vitrée (matériaux, épaisseur) en fonction du climat, de l'exposition et de la surface vitrée.
- l'**urbaniste** : Il contrôle le plan produit par l'architecte et émet un avis sur son travail en fonction de critères d'intégration urbaine du bâtiment. L'architecte doit

alors tenir compte de cet avis pour rédiger un nouveau plan acceptable, et accepté, par l'urbaniste.

- le **pompier** : Avant d'émettre son avis, l'urbaniste peut demander conseil à un pompier pour vérifier que l'appartement est conforme aux normes incendie en vigueur (emplacement des issues de secours, des trappes de désenfumage, ...).

Figure 5.1 Partenaires et Documents Partagés

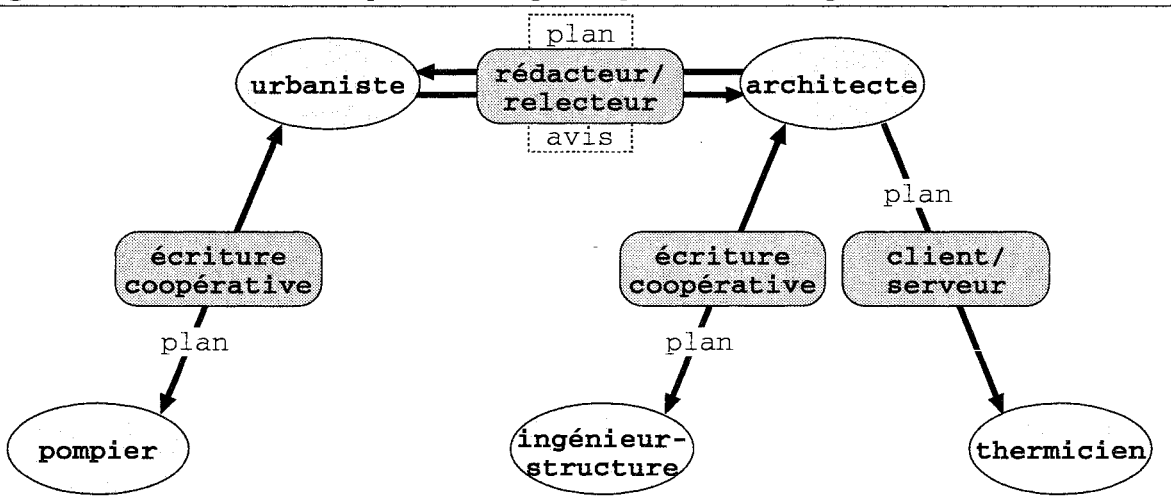


Les différents échanges de données entre les partenaires ne sont toutefois pas soumis aux mêmes règles. S'il est souhaitable, dans certains cas, d'encourager une coopération maximale entre les partenaires, dans d'autres il peut être nécessaire de fixer certaines contraintes. C'est le cas par exemple de l'urbaniste et de l'architecte: ils se partagent certains documents, mais seulement en lecture, chacun ne pouvant modifier les documents de l'autre. Les règles de coopération négociées pour contrôler ces échanges de documents entre les différents partenaires sont représentées sur la figure 5.2.

- **client/serveur** : L'architecte (le "serveur") fournit différentes versions successives du plan au thermicien (le "client"). Les échanges se font donc uniquement de l'architecte vers le thermicien. Ce schéma de coopération a été défini par le prédicat *client_server* au paragraphe 4.3.1.
- **rédacteur/relecteur** : L'urbaniste (le "relecteur") lit, mais ne modifie pas, le plan qui lui est fourni par l'architecte (le "rédacteur"). Il rédige alors son avis qu'il transmet à l'architecte qui le lit, mais ne le modifie pas, pour mettre à jour son plan, et ainsi de suite. Ce schéma de coopération a été défini par le prédicat *writer_reviewer* au paragraphe 4.3.1.

- **écriture coopérative** : L'architecte et l'ingénieur-structure travaillent tous les deux à la rédaction du plan (même objet logique). Durant toute la durée de l'activité de conception ils le modifient, éventuellement simultanément, intégrent les modifications effectuées par l'un ou l'autre, dans le but de fournir au final une version de ce plan qui les satisfait tous les deux. Ce schéma de coopération a été défini par le prédicat *DisCOO* au paragraphe 4.3.1.

Figure 5.2 Schémas de Coopération Négociés pour les Echanges de Documents



5.1.2 Anatomie d'une Activité

Notre objectif est donc de concevoir une infrastructure pour **coordonner**, via l'utilisation de différents **schémas de coopération**, les **échanges de données** entre différentes activités. Il nous faut pour cela pouvoir stocker des données au niveau d'une activité, définir des protocoles d'échange, puis appliquer ces protocoles pour contrôler les interactions entre les activités.

Accès aux Données

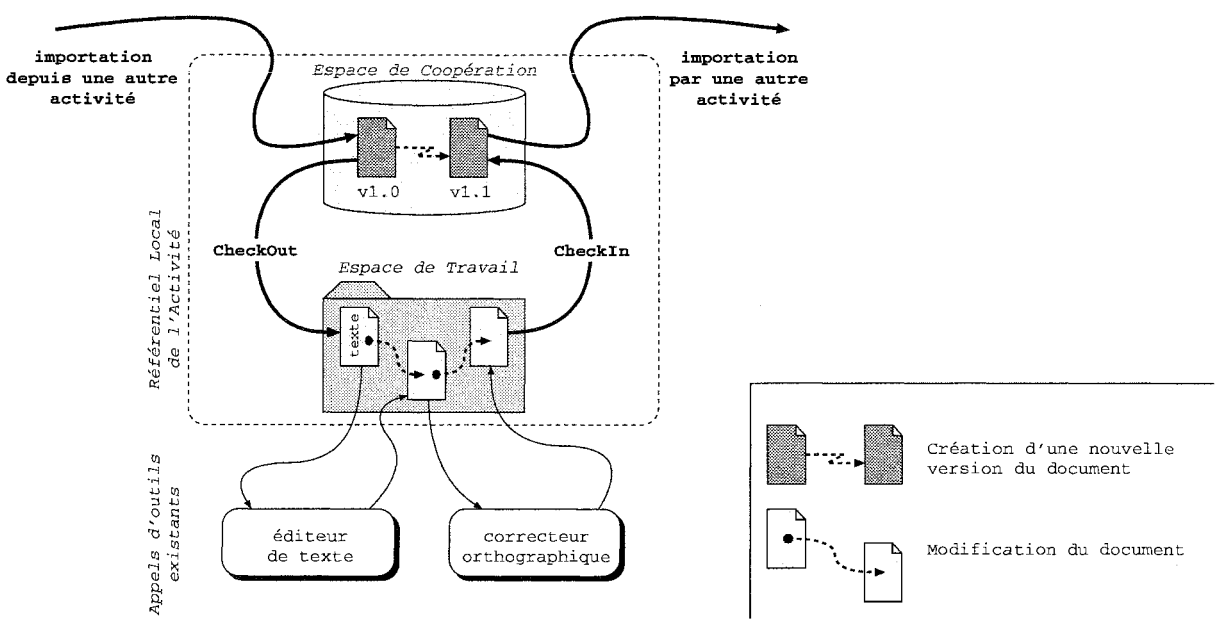
Du point de vue des acteurs du système, c'est-à-dire les utilisateurs, rappelons qu'il est nécessaire qu'ils puissent utiliser leurs applications courantes (Autocad, Word, emacs, gcc, ...) sur les données partagées. Ces données doivent donc être accessibles dans leur format natif, i.e. des fichiers `.DXF` ou `.DOC`, voire même des tuples dans une base de données ou bien des objets dans un environnement orienté objet (ex: CORBA).

D'un autre côté, outre le "contenu" des objets partagés, il nous faut également mémoriser, au niveau de chaque activité, un certain nombre d'informations supplémentaires nécessaires, entre autres, à la vérification des schémas de coopération. Les informations les plus importantes sont bien entendu les **histoires locales** des différentes activités. Elles nous permettent de retracer l'historique des échanges et par conséquent de déterminer les dépendances entre les différentes activités du système et d'évaluer des propriétés sur ces échanges. Parmi les autres informations utiles se trouvent également les histoires de

chacun des objets, tant du point de vue de la gestion des versions (au sens "gestionnaires de configurations") que des méta-données (responsabilités, annotations, ...).

Le référentiel local d'une activité est composé de deux parties: une partie publique, nommée **espace de coopération**, et une partie privée, nommée **espace de travail**. L'espace de coopération contient les versions des objets rendues publiques par l'activité, c'est-à-dire les versions pouvant être importées par d'autres activités, ainsi que les relations entre ces différentes versions le cas échéant. L'espace de coopération correspond à la base locale de l'activité telle que nous l'avions définie au paragraphe 4.1.1. Les échanges de données entre activités seront réalisés entre leurs espaces de coopération respectifs. L'espace de travail permet quant à lui de présenter à l'utilisateur les objets de l'espace de coopération sous une forme utilisable par ses applications existantes (fichiers `.DXF` pour Autocad ou `.DOC` pour Word par exemple). C'est l'endroit où les applications vont effectivement manipuler les données.

Figure 5.3 Espace de Coopération & Espace de Travail



Comme cela est représenté figure 5.3, l'utilisateur devra tout d'abord importer depuis une autre activité le document (sous forme d'objet avec ses "informations de contrôle") qu'il désire utiliser. Ce document sera alors stocké dans son espace de coopération. Afin de pouvoir le manipuler avec ses applications courantes, il devra ensuite transférer (opération `Check_Out`) ce document dans son espace de travail. Les modifications qu'il effectuera alors sur ce document ne seront pas visibles aux autres activités (l'espace de travail est une zone privée). Quand il jugera son travail terminé, il publiera (opération `Check_In`) la nouvelle version de ce document, ce qui aura pour effet de mettre à jour l'objet document stocké dans son espace de coopération. A partir de cet instant, ce document pourra être à son tour importé par d'autres activités.

Le référentiel local d'une activité est ainsi représenté par deux composants distincts: une zone d'échange, l'espace de coopération, accessible aux autres activités et dans laquelle

sont stockés les objets partagés; une zone privée, l'espace de travail, sur les données de laquelle l'activité peut travailler (appels d'outils existants) pour accomplir sa tâche. Les transferts entre ces deux zones sont réalisés à l'initiative de l'activité (et donc de l'utilisateur). En d'autres termes, c'est l'utilisateur qui décide du moment où il publie ses résultats (intermédiaires ou finaux) ainsi que du moment où il intègre, au niveau de son espace de travail, les modifications effectuées sur les objets partagés par les autres activités (préalablement importées dans son espace de coopération).

Contrôle des Echanges

Lorsqu'une activité veut partager un objet donné avec une autre activité, il y a tout d'abord une phase de négociation pour définir le schéma de coopération qui contrôlera les échanges concernant cet objet entre ces deux activités. Il s'agit, au minimum, de s'assurer que le schéma que l'une des activités désire utiliser est connu de l'autre activité. Le résultat de cette négociation est un **contrat** passé entre les deux activités et fixant les règles de coopération à respecter pour le partage de l'objet concerné.

Prenons par exemple le cas de l'architecte et du thermicien: les deux partenaires négocient le schéma de coopération "client/serveur" pour partager l'objet `plan`. Comme nous l'avons indiqué au chapitre 4, la négociation d'un schéma de coopération doit cependant respecter certaines règles et induit elle-même certains contrôles sur les échanges de données utltérieurs. Ces axiomes sont rappelés par la définition 5.1. D'après l'axiome $n°$ 1, cette opération de négociation aura pour conséquence de journaliser l'événement ci-dessous dans les histoires locales de chacune des transactions `architecte` et `thermicien`:

$$Contract[architecte, thermicien, \{plan\}, client\_server]$$

avec le prédicat $client\_server$ défini de la manière suivante:

$$client\_server(t_c, t_s, O) \equiv \begin{cases} DisCOO(t_c, t_s, O) \\ \forall ob \in O \quad \nexists\, p \in Write^{(ob)} \quad \text{tel que} \quad p_{t_c}[ob_{t_s}] \in H \\ \forall ob \in O \quad \nexists\, q \in Read^{(ob)} \quad \text{tel que} \quad q_{t_s}[ob_{t_c}] \in H \end{cases}$$

Nous obtenons ainsi pour chaque activité du système une **table de coopération** contenant tous les contrats signés par cette activité. Chaque activité peut donc déterminer par elle-même les règles à respecter pour échanger les valeurs de tel objet avec telle autre activité (axiome $n°$ 2 de la définition 5.1). Si nous reprenons notre exemple support dans le domaine du bâtiment (figure 5.2), les tables de coopération construites sont représentées sur la figure 5.4. Ainsi, si une opération d'échange concernant l'objet `plan` est invoquée entre les activités `thermicien` et `architecte`, le thermicien voit dans sa table de coopération qu'il doit se comporter en tant que "client" vis-à-vis de l'architecte, tandis que l'architecte, consultant sa propre table de coopération, sait qu'il doit se comporter en tant que "serveur" vis-à-vis du thermicien.

Dans le cas où le schéma de coopération négocié est "rédacteur/relecteur", la situation est quelque peu particulière. Comme nous l'avons expliqué dans la section 4.3.1, ce schéma de coopération considère que l'ensemble des objets concernés est divisé en deux sous-ensembles: le premier est celui des documents, le second est celui des revues. Dans la

DÉFINITION 5.1 (AXIOMES DE RESPECT DES SCHÉMAS NÉGOCIÉS)

1. Un contrat est ratifié simultanément par les deux partenaires, c'est-à-dire que l'événement *Contract* est journalisé en même temps par les deux transactions:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)\quad \forall O\quad \forall s$$
$$(Contract[t_i, t_j, O, s] \in H_{t_1}) \Rightarrow (Contract[t_i, t_j, O, s] \in H_{t_2})$$

2. Une interaction entre deux transactions ne peut avoir lieu que si ces deux transactions ont préalablement négocié un schéma de coopération et si les règles de ce schéma sont vérifiées:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad (p_{t_i}[ob_{t_j}] \in H_t) \Rightarrow$$
$$\exists t_1 \in \{t_i, t_j\}\quad \exists t_2 \in \{t_i, t_j\} - t_1\quad \exists O\ (ob \in O)\quad \exists s$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} p_{t_i}[ob_{t_j}]) \wedge s(t_1, t_2, O)$$

3. Le partage d'un objet donné est contrôlé par un et un seul schéma de coopération (i.e. pas de renégociation possible pour le moment):

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad \forall ob\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)$$
$$(Contract[t_1, t_2, O, s] \in H_t) \wedge (ob \in O) \Rightarrow \nexists s'$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_1, t_2, O', s'] \wedge (ob \in O'))$$
$$\vee\ (Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_2, t_1, O', s'] \wedge (ob \in O'))$$

4. Une transaction ne peut être terminée (`Commit`, `Abort`, ...) que si tous les schémas de coopération qu'elle a négociés avec d'autres transactions sont vérifiés:

$$\forall \alpha \in TE_{t_i}\quad (\alpha \in H_{t_i}) \Rightarrow \forall t_j\quad \forall ob\quad \forall s\quad \forall t \in \{t_i, t_j\}$$
$$(Contract[t_i, t_j, O, s] \in H_t) \wedge (ob \in O) \Rightarrow s(t_i, t_j, O)$$
$$\vee\ (Contract[t_j, t_i, O, s] \in H_t) \wedge (ob \in O) \Rightarrow s(t_j, t_i, O)$$

table de coopération de l'activité `urbaniste` cela signifie donc que l'urbaniste, vis-à-vis de l'architecte, joue le rôle de "rédacteur" pour l'avis (objet du premier sous-ensemble) et le rôle de "relecteur" pour le plan de l'architecte (objet du second sous-ensemble). De manière symétrique (table de coopération de l'activité `architecte`), l'architecte joue le rôle de "rédacteur" pour le plan et le rôle de "relecteur" pour l'avis de l'urbaniste.

Comme nous allons le constater dans la section 5.1.3, lors d'un échange de données entre deux activités (opération de transfert, cf. paragraphe 4.1.2), chacune de ces deux activités contrôlera localement (i.e. à partir des informations contenues dans sa table de coopération) que cet échange est correct par rapport au contrat qu'elles auront négocié toutes les deux. Si l'une ou l'autre détecte une violation du contrat (ou plus exactement du schéma de coopération "figurant" sur le contrat) cette opération d'échange sera refusée.

Une activité comporte donc deux composants dédiés au contrôle de ses échanges avec les autres activités: un **protocole** chargé de la gestion de la table de coopération de l'activité ainsi que de l'évaluation des schémas de coopération figurant dans cette table; un **coordinateur** pour contrôler que tous les accés réalisés sur l'espace de coopération respectent le protocole. Le coordinateur correspond donc à la couche "contrôle des interactions" représentée autour de chaque activité sur les figures 4.1-b et 4.7-b. C'est ce composant qui est chargé de faire respecter les axiomes de la définition 4.17.

Figure 5.4 Tables de Coopération pour la Figure 5.2

activité	partenaire	objets	schéma	rôle
architecte	ing.-structure	{plan}	écriture coopérative	~
architecte	thermicien	{plan}	client/serveur	serveur
architecte	urbaniste	{{plan},{avis}}	rédacteur/relecteur	~

activité	partenaire	objets	schéma	rôle
urbaniste	architecte	{{avis},{plan}}	rédacteur/relecteur	~
urbaniste	pompier	{plan}	écriture coopérative	~

activité	partenaire	objets	schéma	rôle
ing.-structure	architecte	{plan}	écriture coopérative	~

activité	partenaire	objets	schéma	rôle
thermicien	architecte	{plan}	client/serveur	client

activité	partenaire	objets	schéma	rôle
pompier	urbaniste	{plan}	écriture coopérative	~

Nous distinguons ainsi d'un côté les contraintes à vérifier sur les échanges de données (schémas de coopération journalisés par le protocole) et d'un autre côté la manière dont ces contraintes sont vérifiées (axiomes de respect des schémas négociés de la définition 4.17 garantis par le coordinateur). **Les schémas de coopération sont donc réellement les paramètres du contrôle des interactions réalisé par le coordinateur.** De ce fait, le contrôle des échanges est conforme au modèle des *DisCOO*-transactions défini au chapitre 4.

Architecture Retenue

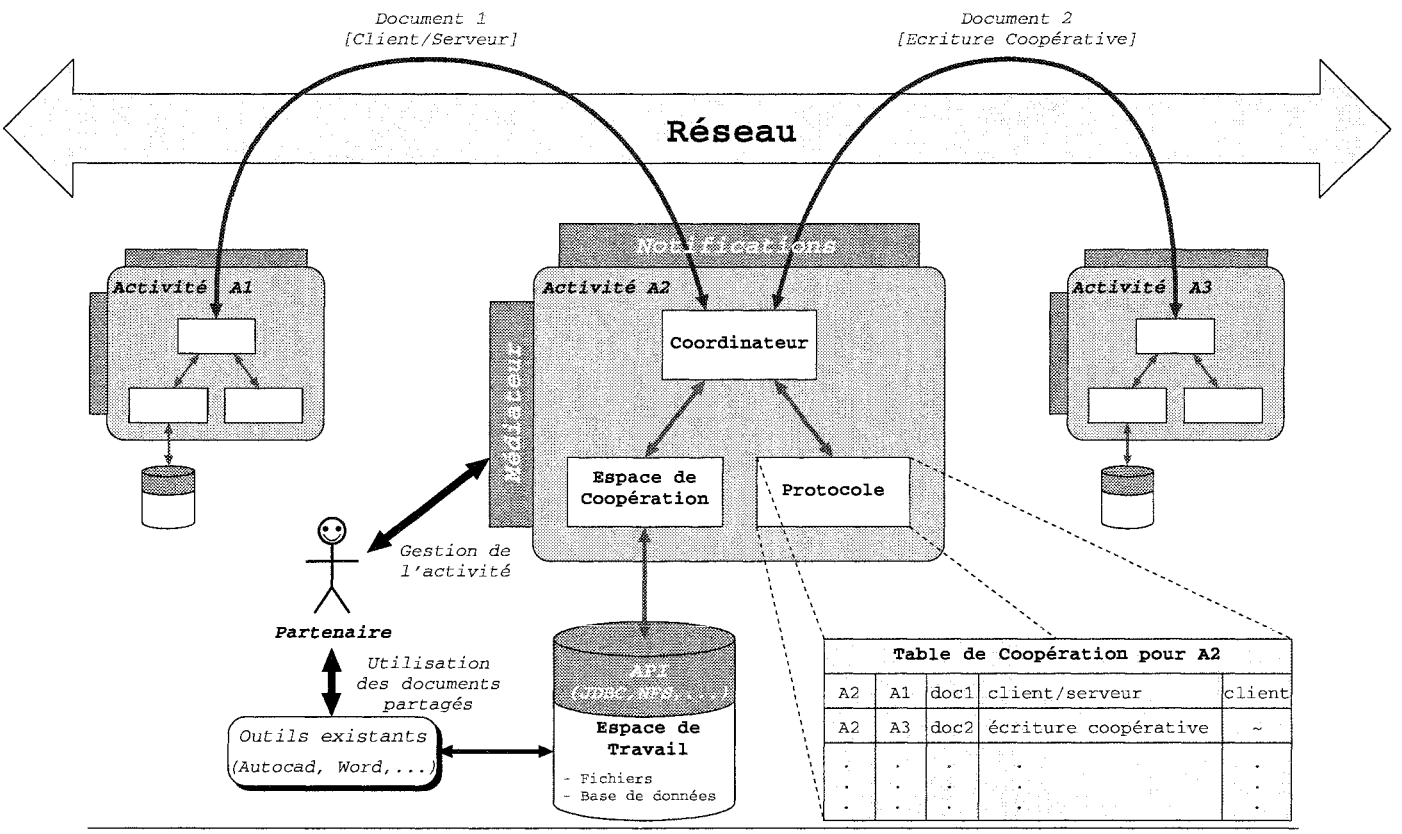
Comme cela est représenté sur la figure 5.5, une activité est donc principalement constituée de quatre composants: un **espace de coopération** et un **espace de travail** pour stocker et manipuler les objets partagés; un **protocole** et un **coordinateur** pour assurer le contrôle de ses échanges avec les autres activités.

- L'**espace de coopération** est la base de données locale de l'activité (cf. paragraphe 4.1.1). Ses fonctions sont la création et la destruction des objets, appelés ressources, manipulés par l'activité, l'identification des ressources et de leurs copies, la gestion (éventuellement) des versions et des configurations de ces ressources locales.

- L'**espace de travail** présente à l'utilisateur les objets partagés sous une forme utilisable par ses applications habituelles (fichiers `.DXF` pour Autocad ou `.DOC` pour Word par exemple).

- Le **protocole** définit les règles de coopération que cette activité doit respecter pour échanger des ressources avec les autres activités du système. Il s'agit en fait de savoir quels sont les schémas de coopération que l'activité concernée a négociés avec les

Figure 5.5 Architecture Générale de Notre Système de Coopération



autres activités. Il est également chargé de vérifier, à la demande du coordinateur (cf. axiome $n°$ 2 de la définition 5.1), si une opération concernant l'activité associée doit être acceptée ou refusée. Il se base pour cela sur l'histoire locale de l'activité (cf. paragraphe 4.1.1), les différentes règles des schémas de coopération étant utilisées en tant que pré-conditions pour cette opération.

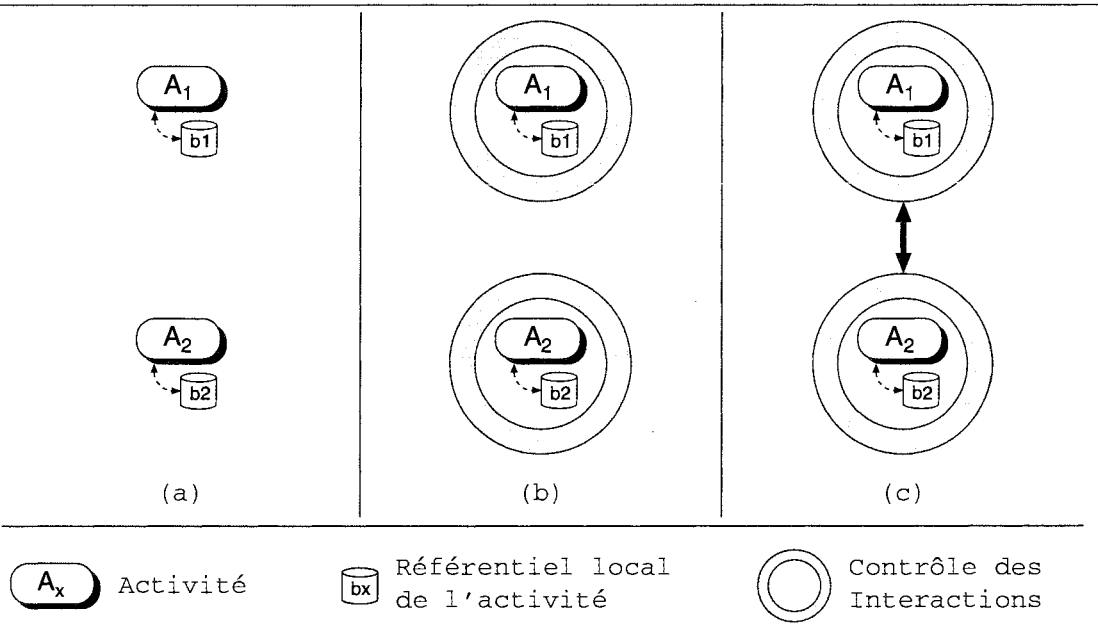
- Le **coordinateur** joue en quelque sorte le rôle d'interface de cette activité vis-à-vis des autres activités du système (cf. couche "contrôle des interactions"). Toutes les communications entre activités doivent passer par leurs coordinateurs respectifs. Ceci évite que deux activités ne puissent s'échanger des ressources directement (i.e. entre leurs espaces de coopération) sans que ces échanges ne soient contrôlés par le protocole. En outre, ceci nous permet de définir de manière indépendante la partie "stockage des données" de la partie "contrôle des échanges".

Afin d'illustrer le rôle de chacun des composants, tant au sein de l'activité elle-même que lors des interactions entre activités, la section suivante présente leur fonctionnement dans différentes situations telles que la négociation d'un schéma de coopération, l'importation d'un document, l'utilisation de l'espace de travail, la terminaison des activités ou encore la détection des conflits entre schémas.

5.1.3 Fonctionnement des Différents Composants

Nous venons d'expliquer de quelle manière les différentes activités stockent leurs données (espaces de coopération) et de quelle façon elles les mettent à la disposition des utilisateurs (espaces de travail). C'était l'étape a sur la figure 5.6. Nous avons ensuite présenté le composant appelé protocole dont le rôle est d'indiquer quelles sont les règles de coopération à respecter en termes de contrats et de schémas de coopération négociés entre les activités (étape b). La troisième étape consiste maintenant à étudier la **coordination des échanges** entre les activités du système, c'est-à-dire le fonctionnement de la couche "contrôle des interactions" qui sera représentée par le composant appelé coordinateur (étape c).

Figure 5.6 Anatomie d'une Activité: Stockage ⇝ Protocole ⇝ Coordination



Transferts de Données

Comme nous l'avons déjà expliqué à plusieurs reprises, notre objectif principal est de rendre les activités les plus autonomes possible malgré les contraintes imposées par la coopération et la distribution. Cela signifie en particulier que chaque activité doit être capable de contrôler elle-même ses propres interactions avec les autres activités du système. Il s'agit là en effet d'une différence fondamentale par rapport aux systèmes distribués classiques basés sur des architectures client/serveur: un transfert de données entre deux activités ne sera plus contrôlé par un site central mais directement par les deux activités impliquées dans cet échange. Elles se baseront pour cela sur le contrat qu'elles auront préalablement négocié et dans lequel elles auront défini les règles de coopération qu'elles doivent respecter lors de leurs échanges.

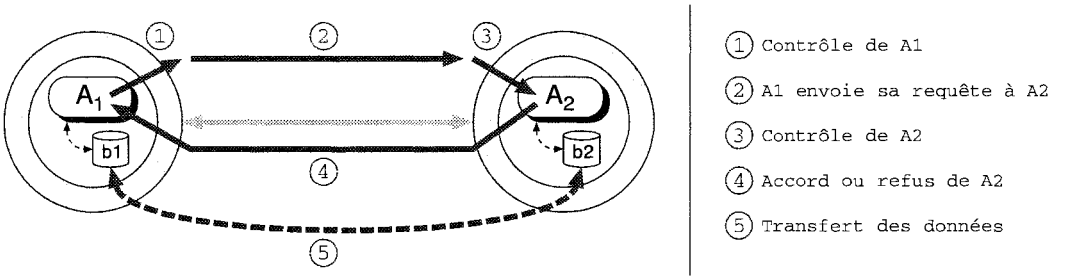
Les activités ne sont toutefois pas censées avoir une confiance aveugle les unes dans les autres, c'est-à-dire que si deux activités ont négocié un schéma de coopération, il ne suffit

pas que l'une d'entre d'elles prétende qu'un échange soit conforme à ce schéma pour qu'un échange soit accepté par l'autre activité sans autre vérification. Chacune d'elles contrôlera de son côté que cet échange est conforme au schéma (cf. axiome $n°$ 2 de la définition 5.1). Si les deux activités sont d'accord, alors l'échange peut avoir lieu; sinon il est refusé.

Comme cela est représenté sur la figure 5.7, un échange de données entre deux activités se déroulera en cinq étapes. Les quatre premières étapes concernent la vérification des schémas de coopération, alors que la cinquième constitue le transfert des données proprement dit.

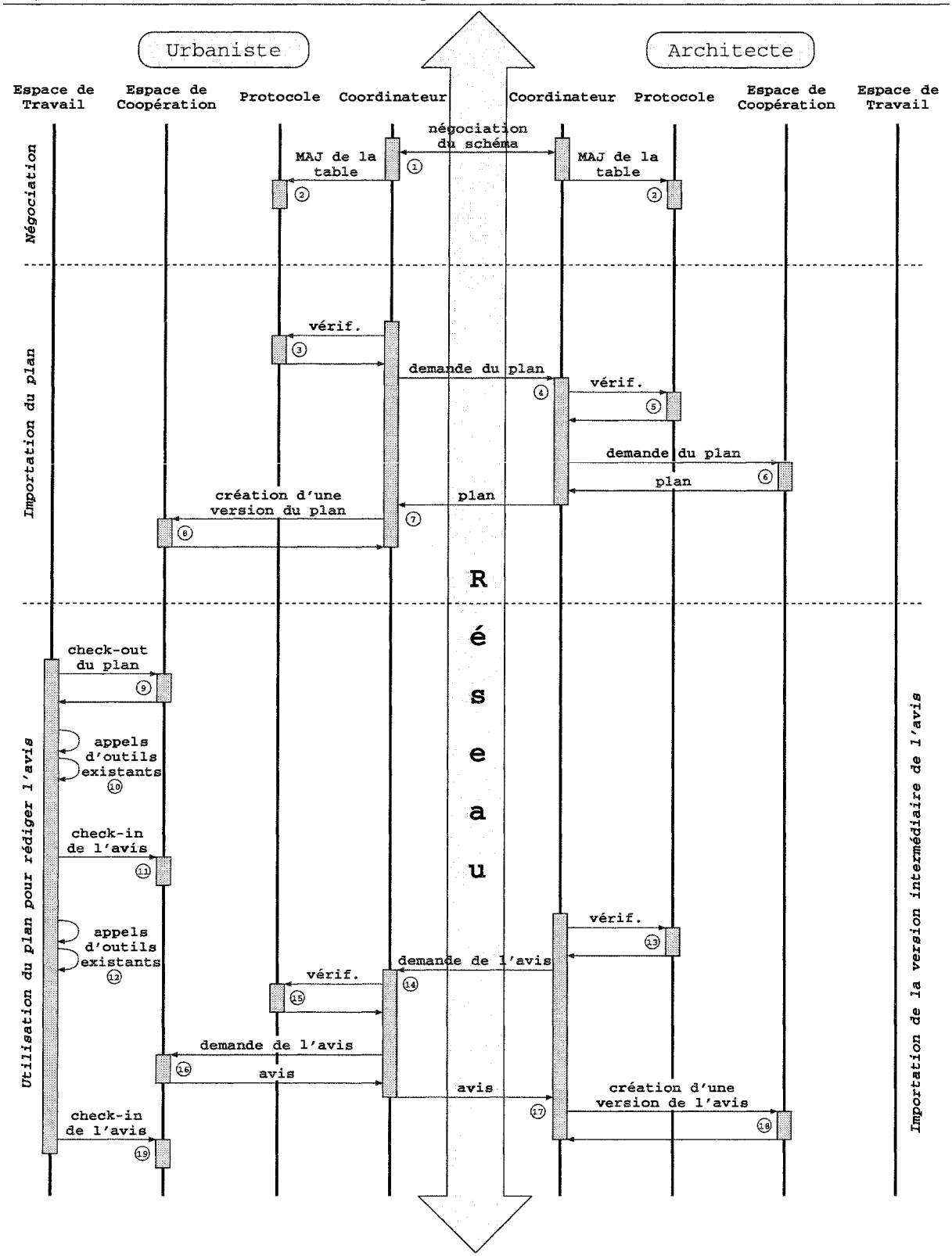
1. Contrôle de l'activité A_1: le coordinateur (chargé du contrôle des interactions) de cette activité vérifie auprès de son protocole s'il existe un contrat passé avec l'activité A_2 relatif à l'objet *ob* impliqué dans cet échange. Ce contrôle est réalisé en deux étapes correspondant aux deux membres de la conjonction de l'axiome $n°$ 2 de la définition 5.1, c'est-à-dire:
   - On vérifie d'abord qu'un événement $Contract[t_i, t_j, O, s]$ avec $t_i \in \{A_1, A_2\}$, $t_j \in \{A_1, A_2\} - t_i$ et $ob \in O$ a été journalisé dans l'histoire de la transaction A_1.
   - Si c'est le cas, le coordinateur demande au protocole de vérifier si cet échange est correct par rapport au schéma de coopération figurant sur le contrat, c'est-à-dire d'évaluer le prédicat $s(t_i, t_j, O)$ sur l'histoire locale de la transaction A_1.
2. Le coordinateur de l'activité A_1 transmet une requête au coordinateur de l'activité A_2 pour demander l'autorisation d'effectuer cet échange.
3. Contrôle de l'activité A_2: le coordinateur de cette activité vérifie auprès de son protocole si cet échange est correct par rapport au schéma de coopération figurant sur le contrat négocié avec l'activité A_1 (s'il existe).
4. Le coordinateur de l'activité A_2 transmet au coordinateur de l'activité A_1 sa réponse à la requête émise par A_1 ainsi qu'un "ticket" pour procéder à l'échange.
5. Le transfert de données peut avoir lieu entre les espaces de coopération. Celui-ci est effectué via les coordinateurs respectifs des deux activités qui vont vérifier que le contenu de cet échange est conforme au ticket émis.

Figure 5.7 Déroulement d'un Echange



Nous pouvons remarquer que ceci met bien en évidence le rôle primordial des coordinateurs des activités dans notre architecture. En effet, toutes les communications entre

Figure 5.8 Exemple d'Utilisation : Négociation, Importation & Espaces de Travail



• **Négociation du schéma de coopération:**

1. Les deux activités doivent tout d'abord négocier, par l'intermédiaire de leurs coordinateurs respectifs, un schéma de coopération.
2. Une fois le schéma choisi, chaque activité met à jour la table de coopération de son protocole.

• **Importation d'un document:** Afin de pouvoir commencer à travailler, l'urbaniste va demander à l'architecte de lui fournir une première version du plan:

3. Avant de transmettre la demande à l'architecte, le coordinateur de l'urbaniste vérifie que le protocole autorise l'urbaniste à importer le plan de l'architecte.
4. Le coordinateur de l'urbaniste transmet ensuite la demande d'importation du plan au coordinateur de l'architecte. Rappelons en effet que toutes les communications entre activités doivent obligatoirement passer par leurs coordinateurs respectifs.
5. Le coordinateur de l'architecte vérifie alors auprès de son propre protocole que cet échange est autorisé.
6. Les deux activités ayant donné leur accord, l'échange peut avoir lieu. Le coordinateur de l'architecte demande donc à son espace de coopération de lui fournir le plan (ou plus exactement une copie, i.e. l'original reste chez l'architecte).
7. Le coordinateur de l'architecte transmet le plan au coordinateur de l'urbaniste.
8. A réception du message, le coordinateur de l'urbaniste crée dans son propre espace de coopération une copie du plan.

• **Utilisation de l'espace de travail:** L'urbaniste va maintenant travailler sur le plan qui lui a été fourni par l'architecte pour rédiger son avis. Afin de pouvoir utiliser ses applications habituelles, il doit tout d'abord convertir le document `plan` qu'il possède dans son espace de coopération en fichier (ex: fichier `.DXF` pour Autocad) au sein de son espace de travail.

A l'inverse, une fois qu'il aura rédigé son avis (ex: fichier `.DOC` de Word) dans son espace de travail, il devra le convertir en document `avis` au sein de son espace de coopération.

9. Conversion du document `plan` en fichier.
10. Utilisation des applications existantes (ex: Word) pour rédiger l'avis (sous forme de fichier).
11. Afin que cet avis puisse être importé par d'autres activités, l'urbaniste doit le publier. Il doit pour cela le transférer dans son espace de coopération (conversion du fichier en document).
12. Après avoir publié une première version ("intermédiaire") de son avis, il peut continuer à le modifier.
13. A un moment donné, l'architecte demande à l'urbaniste de lui envoyer son avis. Avant de transmettre la demande à l'urbaniste, le coordinateur de l'architecte vérifie que le protocole autorise l'architecte à importer l'avis de l'urbaniste.
14. Le coordinateur de l'architecte transmet ensuite la demande d'importation de l'avis au coordinateur de l'urbaniste.
15. Le coordinateur de l'urbaniste vérifie alors auprès de son propre protocole que cet échange est autorisé.
16. Les deux activités ayant donné leur accord, l'échange peut avoir lieu. Le coordinateur de l'urbaniste demande donc à son espace de coopération de lui fournir l'avis (ou plus exactement une copie, l'original restant chez l'urbaniste).
17. Le coordinateur de l'urbaniste transmet l'avis au coordinateur de l'architecte.
18. A réception du message, le coordinateur de l'urbaniste crée dans son propre espace de coopération une copie de l'avis.
19. Pendant ce temps, l'urbaniste a continué à travailler sur son avis. Il en publie alors une nouvelle version.

deux activités devront obligatoirement passer par leurs coordinateurs respectifs. De cette façon, nous pouvons garantir, pour chaque activité, que tous ses échanges sont corrects par rapport à son protocole (i.e. les schémas de coopération figurant dans sa table de coopération). Il est par conséquent impossible pour une activité de "polluer" le référentiel local (i.e. l'espace de coopération) d'une autre activité en effectuant des opérations non contrôlées par cette activité.

Chaque activité est donc responsable de ses propres interactions avec les autres activités du système. Ainsi, l'architecture proposée est conforme au modèle de transactions coopératives distribués que nous avons défini dans la section 4 et à ses objectifs, à savoir:

- les objets partagés sont stockés localement sur les activités
- le contrôle des échanges est réalisé localement sur chaque activité
- les activités peuvent négocier les schémas de coopération

Pour conclure la présentation de notre architecture, un exemple d'échanges entre les activités `architecte` et `urbaniste` est fourni figure 5.8. Nous y retrouvons tout d'abord la phase de négociation d'un schéma de coopération (avec mise à jour des tables de coopération), les différentes étapes de l'importation effectuée par l'urbaniste du plan rédigé par l'architecte, puis un exemple d'utilisation de ce plan à l'aide d'applications existantes (relations entre espace de coopération et espace de travail).

Comme nous pouvons le constater sur cet exemple, toutes les communications entre activités se font effectivement par l'intermédiaire de leurs coordinateurs. Un coordinateur peut donc être considéré comme étant "l'interface" de l'activité associée vis-à-vis des autres activités du système.

Terminaison des Activités

Un des principaux problèmes relatifs à la coordination d'activités distribuées est celui de leur terminaison. En effet, une activité est censée produire des résultats finaux cohérents par rapport aux données du système. Pour cela, un certain nombre d'axiomes (les règles des schémas de coopération, représentant eux-mêmes des critères de correction) vont être vérifiés lors de la phase de terminaison d'une activité afin de s'assurer que les résultats produits sont cohérents. Cette vérification est à la charge du coordinateur (cf. axiome $n°$ 4 de la définition 5.1).

Dans un système centralisé cette tâche est facilitée par le fait qu'il est possible de "figer" le système pour le temps de la vérification et de définir un processus unique chargé de la coordination de toutes les activités. A l'inverse, un système distribué est susceptible de continuer à évoluer durant cette vérification. En outre, aucune activité n'a une vue globale du système et la coordination est donc réalisée "de proche en proche". Par conséquent, il est nécessaire de mettre en œuvre des mécanismes adaptés à la distribution.

Outre les problèmes de diffusion et d'ordonnancement inhérents aux communications par envois de messages, un point important concerne les groupes d'activités. Dans *COO* un groupe est défini comme étant un ensemble d'activités impliquées dans un même cycle de dépendances. Si l'on considère maintenant un schéma de coopération imposant un consensus sur les valeurs finales entre les activités d'un groupe (cas de la *DisCOO*-sérialisabilité par exemple), l'idée sous-jacente est que toutes les activités de ce groupe doivent être validées en même temps (cf. point 4 de la preuve du lemme 4.4 page 85

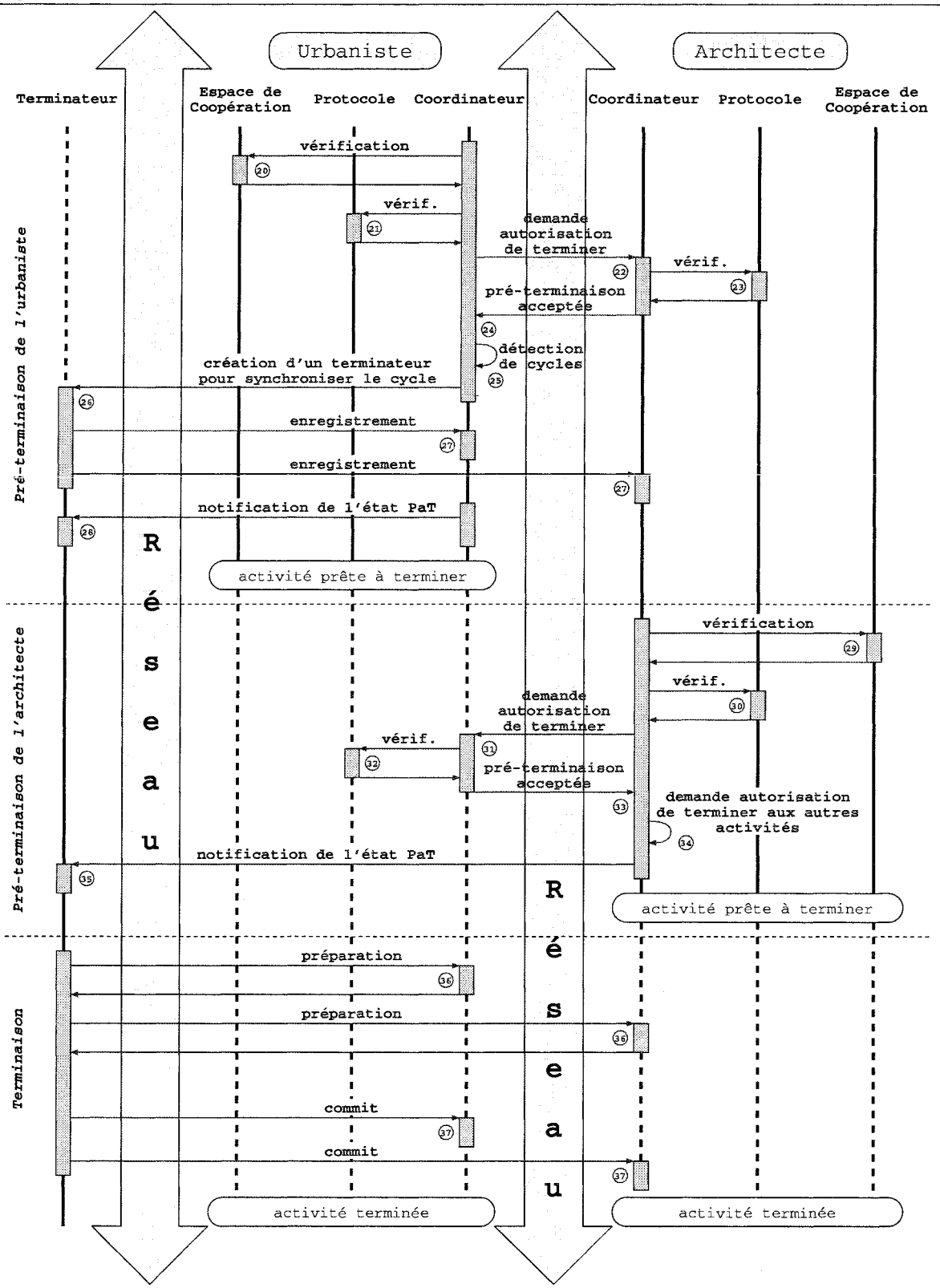
en ce qui concerne la *DisCOO*-sérialisabilité). Si, en termes de critères de correction, il suffit de dire que les "Commit" de chacune des transactions du groupe doivent apparaître simultanément dans les histoires locales de ces transactions, il est nécessaire, du point de vue de la mise en œuvre, d'indiquer de quelle façon cette synchronisation sera réalisée.

Comme nous l'avons indiqué, l'idée est que toutes les transactions d'un même groupe, au sens *COO*, soient validées en même temps, et non qu'elles s'attendent les unes les autres. Puisque nous sommes en milieu distribué, une solution consiste à effectuer une validation des transactions en deux phases (*"two phases commit"* ou 2PC). Une transaction désirant terminer vérifie qu'elle est *up_to_date* puis passe de l'état `active` à l'état `prête-à-terminer` (*"ready_to_commit"*). Idem pour les autres transactions du groupe. Lorsque toutes les transactions du groupe sont dans l'état `prêtes-à-terminer`, alors elles sont toutes validées en même temps. S'il arrivait qu'une transaction dans l'état `prête-à-terminer` ne soit plus *up_to_date* suite à des modifications effectuées par une transaction dont elle dépend, elle repasserait alors dans l'état `active`. Nous avons ainsi décomposé la phase de terminaison d'une activité en trois étapes:

1. Lorsqu'une activité (dans l'état `active`) désire terminer, elle passe tout d'abord dans un état `prête-à-terminer`. Cela suppose bien entendu qu'elle soit dans une configuration "cohérente", c'est-à-dire qu'à cet instant précis son exécution est correcte par rapport aux schémas de coopération qu'elle a négociés. Cela signifie que son coordinateur a vérifié avec succés l'axiome $n°$ 4 de la définition 5.1. Si, à un moment quelconque, elle n'est plus dans une configuration cohérente (suite à des opérations invoquées par d'autres activités par exemple), cette activité repasse alors dans l'état `active`.
   Lorsqu'une activité passe dans l'état `prête-à-terminer`, elle initie également une détection de cycles (effectuée de proche en proche). Cet algorithme ne s'intéresse en fait qu'aux dépendances entre les activités qui doivent atteindre un consensus sur les valeurs finales des données partagées. En effet, si dans un cycle de dépendances il existe des activités qui ne sont pas obligées de relire les valeurs finales des données utilisées, il est inutile de grouper les activités impliquées dans ce cycle puisqu'il ne sera pas nécessaire qu'elles terminent en même temps.
   Si notre activité ne détecte aucun cycle, elle peut alors terminer directement (`Commit`). Sinon, elle demande au système la création d'un **terminateur** qui sera chargé de la terminaison "simultanée" de toutes les activités impliquées dans le cycle détecté. Ce terminateur n'a strictement rien à voir avec les schémas de coopération et le contrôle des échanges: ceux-ci reste à la charge des activités elles-mêmes (et de leurs coordinateurs). Le rôle du terminateur est uniquement de garantir que toutes les activités dont il a la charge terminent en même temps, c'est-à-dire de mettre en œuvre un 2PC. A noter que ce terminateur a une existence propre: il est totalement autonome par rapport à l'activité qui a demandé sa création. En outre, si plusieurs activités détectent le même cycle de dépendances il ne sera créé qu'un seul terminateur pour ce cycle.

2. Lorsqu'un terminateur se rend compte que toutes les activités qu'il gère sont dans l'état `prêtes-à-terminer`, il peut tenter une terminaison. Il s'agit d'un processus de validation à deux phases (*"Two Phases Commit"*). Il demande donc aux activités

Figure 5.9 Exemple d'Utilisation : Terminaison des Activités



- **Terminaison des activités :**

20. Avant de transmettre sa demande à l'architecte, le coordinateur de l'urbaniste vérifie tout d'abord que l'urbaniste est bien dans un état lui permettant de terminer. Il interroge pour cela son espace de coopération pour être certain qu'il n'existe plus de document en cours de modification (i.e. dans l'espace de travail).
21. Le coordinateur de l'urbaniste demande ensuite à son propre protocole si l'urbaniste est autorisé à terminer (i.e. exécution conforme aux schémas de coopération négociés).
22. Du point de vue de l'urbaniste, l'opération de terminaison est acceptable. Le coordinateur de l'urbaniste transmet donc la demande à l'architecte.
23. A réception de la demande, le coordinateur de l'architecte interroge son protocole pour savoir si l'urbaniste est autorisé à terminer. Dans le cas présent il s'agit, entre autres, de vérifier que l'urbaniste à bien relu la dernière version en date du plan avant d'émettre son avis.
24. Le schéma rédacteur/relecteur utilisé entre les activités `urbaniste` et `architecte` impose à l'urbaniste de prendre en compte la dernière valeur du plan produite par l'architecte. Comme l'activité de ce dernier n'est pas encore terminée (`Commit`), l'activité `urbaniste` passe de l'état `active` à l'état `prête-à-terminer`.
25. Ceci fait, le coordinateur de l'urbaniste démarre la procédure de détection de cycle. L'urbaniste ayant importé le plan de l'architecte et l'architecte ayant importé l'avis de l'urbaniste, chacun devant prendre en compte la valeur finale du document partagé, un cycle est détecté entre ces deux activités.
26. Ces deux activités devront donc se terminer en même temps (consensus sur les valeurs finales). Le coordinateur de l'urbaniste crée pour cela un terminateur dont le rôle sera de s'assurer que toutes les activités du groupe `{urbaniste,architecte}` seront validées simultanément.
27. Ce terminateur informe immédiatement les activités dont il a la charge de sa présence.
28. Le coordinateur de l'urbaniste indique alors à ce terminateur que l'activité `urbaniste` est maintenant dans l'état `prête-à-terminer`.

Plus tard, l'architecte a achevé son travail et décide de terminer son activité. Il en informe alors son coordinateur.

29. Avant de transmettre sa demande à l'urbaniste, le coordinateur de l'architecte vérifie tout d'abord que l'architecte est bien dans un état lui permettant de terminer.
30. Le coordinateur de l'architecte demande ensuite à son propre protocole si l'architecte est autorisé à terminer. (i.e. exécution conforme aux schémas de coopération négociés).
31. Du point de vue de l'architecte, l'opération de terminaison est acceptable. Le coordinateur de l'architecte transmet donc la demande à l'urbaniste.
32. A réception de la demande, le coordinateur de l'urbaniste vérifie que l'architecte à bien relu la dernière version de l'avis.
33. Le schéma rédacteur/relecteur utilisé entre les activités `urbaniste` et `architecte` impose à l'architecte de prendre en compte la dernière valeur de l'avis rédigée par l'urbaniste. Comme l'activité de ce dernier n'est pas encore terminée (`Commit`), l'activité `architecte` devra donc passer de l'état `active` à l'état `prête-à-terminer`.
34. Avant de passer effectivement dans l'état `prête-à-terminer`, l'activité `architecte` doit également demander l'autorisation de terminer aux autres activités avec lesquelles elle a communiqué et qui ne sont pas encore terminées.
35. Une fois l'exécution de l'activité `architecte` jugée valide (par rapport aux schémas de coopération négociés), son coordinateur la fait passer dans l'état `prête-à-terminer` et en informe le terminateur.

Toutes les activités dont a la charge le terminateur sont maintenant dans l'état `prêtes-à-terminer`. Il peut donc initier la phase de terminaison effective. Il s'agit d'un *"Two Phases Commit"*:

36. Phase de préparation: chaque activité vérifie que les activités dont elle dépend et qui ne font pas partie du cycle sont bien terminées (`Commit`).
37. Phase de validation: une fois toutes les activités préparées, elles sont toutes validées "en même temps".

dont il a la charge de se "préparer". Chacune d'elles vérifie alors qu'elle est en position de terminer en supposant que toutes les activités de ce cycle sont déjà terminées. Une fois dans l'état `préparée`, une activité ne peut rien faire d'autre qu'attendre un ordre de terminaison effective (`Commit`) ou d'annulation de la préparation (retour à l'état `prête-à-terminer`) de la part du terminateur.

3. Lorsque toutes les activités d'un cycle sont dans l'état `préparées`, le cycle est en quelque sorte "figé". Le terminateur peut alors envoyer un ordre de terminaison effective (`Commit`) à chacune de ces activités.

L'utilisation d'un terminateur pour synchroniser la terminaison d'un groupe d'activités peut sembler contraire à nos objectifs de distribution et d'autonomie puisqu'il a un rôle "centralisateur". En fait, un tel terminateur n'est nécessaire que dans le cas d'un groupe d'activités inter-dépendantes et devant "converger" vers un état final accepté par chacune d'entre elles. Et c'est bien cette notion de convergence qui nous impose de valider toutes les activités du groupe de manière atomique. Si tel n'était pas le cas, les activités encore `actives` pourraient diverger de nouveau et ainsi remettre en cause des résultats produits par des activités validées (`Commit`), ce qui serait en contradiction avec la propriété de durabilité ! Cette "centralisation" de la phase de terminaison ne concerne toutefois que les activités impliquées dans le cyle. Les autres activités du système ne sont pas perturbées et peuvent ainsi continuer à travailler.

En outre, le fait de négocier un schéma de coopération imposant un accord des différents partenaires sur les valeurs finales signifie également que l'on accepte, au moment de la terminaison, d'avoir à se resynchroniser avec ses partenaires. Pour cela, une activité devra multiplier les communications avant de pouvoir terminer afin, justement, de prendre en compte les derniers résultats produits par ses partenaires. Par conséquent, le terminateur ne sera qu'un "partenaire" de plus, mais auquel on ne s'adressera qu'au moment du `Commit`.

Pour illustrer le déroulement de cette phase de terminaison des activités, les différentes étapes de la terminaison des activités `architecte` et `urbaniste` de notre exemple support sont représentées sur la figure 5.9. L'urbaniste est le premier à vouloir conclure. Etant donné qu'il a négocié un schéma de coopération "rédacteur/relecteur" avec l'architecte pour le partage du plan et de l'avis, il faut vérifier qu'il a bien pris en compte la dernière version du plan produite par l'architecte. Toutefois, même si c'est le cas à cet instant, l'activité `urbaniste` ne sera pas pour autant autorisée à terminer. En effet, l'activité `architecte` étant toujours `active`, le plan est susceptible d'être modifié. L'activité `urbaniste` ne peut donc que passer dans l'état `prête-à-terminer`[33] en attendant que l'activité `architecte` soit achevée (`Commit`). Ceci risque cependant de nous conduire à un interblocage en cas de cycle dans le graphe des dépendances entre activités. C'est pour cette raison que l'activité `urbaniste` initie un processus de détection de cycles (effectuée de proche en proche). Dans notre cas, l'urbaniste a importé le plan produit par l'architecte et l'architecte a importé l'avis rédigé par l'urbaniste. En outre, chacun d'eux est obligé (cf. définition du schéma "rédacteur/relecteur" au paragraphe 4.3.1) de prendre en compte

33. Si l'architecte venait à produire une nouvelle version de son plan, ceci aurait pour effet de faire repasser l'activité `urbaniste` dans l'état `active`.

la valeur finale des objets partagés avant de pouvoir terminer. Nous sommes donc en présence d'un cycle entre les activités `architecte` et `urbaniste`. La seule façon d'atteindre ce consensus est que les deux activités soient validées "en même temps". L'urbaniste crée à cet effet un terminateur dont le rôle sera d'orchestrer la terminaison des activités du groupe `{urbaniste,architecte}`.

Lorsque l'architecte désirera conclure, la démarche sera identique. Le cycle détecté par l'architecte étant le même que celui géré par le terminateur existant, il est inutile d'en créer un second. A cet instant, toutes les activités contrôlées par ce terminateur sont donc dans l'état `prêtes-à-terminer`. Il peut alors tenter une terminaison effective de ces activités. Il s'agit d'une validation en deux phases: préparation puis validation. Les activités doivent donc tout d'abord se préparer. Cela signifie qu'elles vont s'assurer que toutes les activités dont elles dépendent et qui ne font pas partie de ce cycle sont bien terminées (`Commit`). Si une des activités ne peut être "préparée", alors le 2PC est annulé et sera retenté ultérieurement par le terminateur. Sinon, le terminateur peut effectuer la deuxième phase du 2PC: la validation (`Commit`) des activités.

Bilan

Nous venons de présenter le fonctionnement des différents composants de notre architecture dans le cas idéal où il n'y a aucun conflit. Nous nous sommes servis de notre exemple support pour illustrer quatre cas d'utilisation de notre système:

- négociation d'un schéma de coopération entre deux activités
- importation d'une ressource (opération de transfert)
- utilisation d'une ressource à l'aide d'applications existantes
- terminaison des activités

La section suivante considère le cas où un échange de données entre deux activités est refusé car le coordinateur d'une de ces deux activités a détecté une violation du schéma de coopération négocié. Une telle erreur peut être due à un conflit entre deux schémas de coopération. Nous expliquons donc non seulement comment détecter un tel conflit mais également de quelle manière nous pouvons le résoudre, certaines des solutions proposées faisant toutefois partie des perspectives de nos travaux.

5.1.4 Conflits entre Schémas de Coopération

Comme nous l'avons mentionné dans la section 4.3.3, le fait d'utiliser plusieurs schémas de coopération simulanément dans un même système peut provoquer des conflits entre ces différents schémas. Par conflit nous entendons qu'une séquence d'opérations effectuées par une activité peut être correcte par rapport à un des schémas qu'elle utilise, mais incorrecte par rapport à un autre.

Nous expliquons dans cette section de quelle manière un tel conflit sera détecté par les coordinateurs des activités. Toutefois, certains conflits ne peuvent être détectés que tardivement, ce qui peut avoir pour conséquence de conduire les activités à une situation de blocage. Afin d'éviter de devoir abandonner une activité (un `Abort` de la transaction), nous proposons ensuite deux mécanismes permettant de résoudre un conflit: la re-négociation

du schéma de coopération et la possibilité, pour une activité, de revenir à une ancienne version d'un objet source de conflit.

Détection des Conflits

Un tel conflit peut se manifester de deux manières différentes selon la politique de contrôle des interactions adoptée par l'activité. En effet, comme nous l'avons présenté dans la section 4.3.3, une activité A peut, lors d'un échange de données avec une activité B_i, soit se conformer strictement au schéma de coopération s_i négocié entre A et B_i (politique **optimiste**), soit vérifier que cet échange est correct par rapport à tous les schémas $s_1..s_n$ qu'elle aura négociés avec les activités $B_1..B_n$ (politique **pessimiste**). Cette politique de contrôle des interactions est fixée au niveau du coordinateur de l'activité. Dans le cas de la définition 5.1 (axiomes de respect des schémas négociés), l'axiome $n°$ 2 correspond à une politique optimiste puisque, lors d'une opération de transfert entre les activités A et B_i, seul le schéma de coopération s_i négocié entre ces deux activités est vérifié.

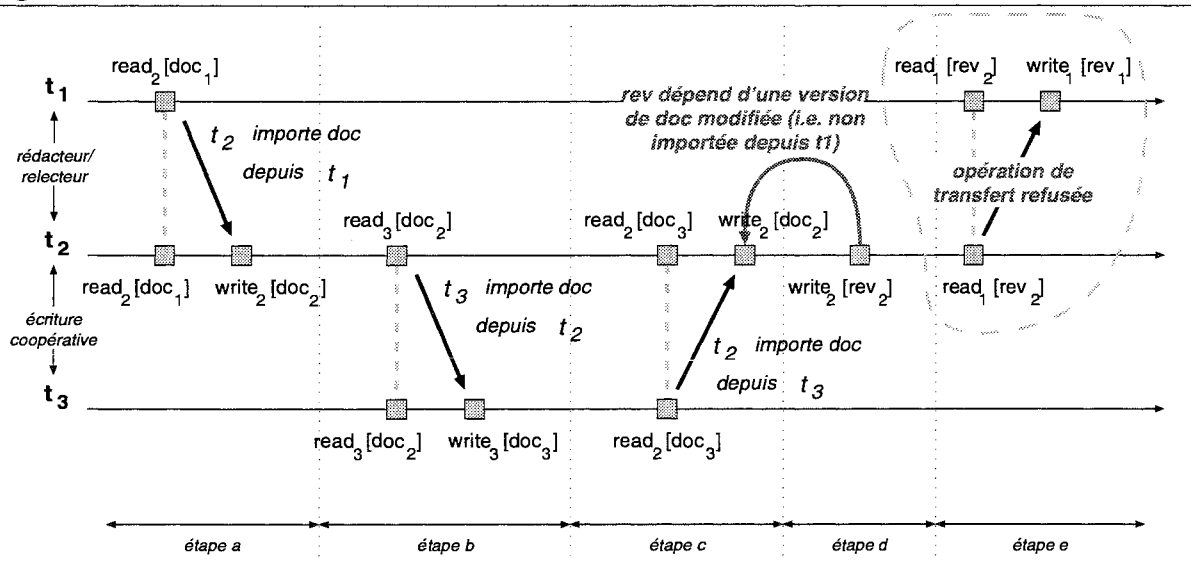
- Politique **optimiste** : Si cet échange entre les activités A et B_i est correct par rapport au schéma de coopération s_i mais incorrect par rapport à un schéma s_j négocié avec une autre activité B_j ($j \neq i$), ce "conflit" ne sera détecté que lors d'un prochain échange entre A et B_j.
- Politique **pessimiste** : L'idée est de détecter un tel conflit au plus tôt. Toutefois, il est ainsi possible que l'on refuse un échange entre les activités A et B_i bien qu'il soit conforme au schéma s_i qu'elles auraient négocié, ceci à cause d'un conflit avec un autre schéma s_j dont l'activité B_i n'a même pas connaissance !

Si une activité A utilise une politique pessimiste, elle s'assurera que chacun de ses échanges avec n'importe quelle autre activité B_i vérifie toutes les règles de coopération (utilisées en tant que préconditions aux opérations) définies par tous les schémas figurant dans sa table de coopération. Si un tel échange génère, en étant ajouté à l'histoire locale de l'activité A, une exécution incorrecte par rapport à un schéma utilisé entre les activités A et B_j, alors cet échange est refusé (même si $j \neq i$). En procédant de cette façon, le risque est d'aboutir à une situation où tous les échanges qui seront tentés avec l'activité A seront refusés par au moins un des schémas présents dans la table de coopération de A. De plus, nous avons vu à la section 4.3.3 que l'intégrité des données n'était pas obligatoirement toujours assurée car certains conflits entre schémas pouvaient n'être détectés que tardivement (cf. exemple de la figure 4.14 page 106).

Si au contraire une activité A utilise une politique optimiste, elle se contentera de vérifier que chaque échange avec une autre activité B_i est correct par rapport au schéma de coopération négocié avec B_i. Si un tel échange génère une exécution incorrecte par rapport à un schéma utilisé entre les activités A et B_j ($j \neq i$), cette "incohérence" ne sera détectée que lors d'un prochain échange entre les activités A et B_j, échange qui risque donc d'être refusé. Nous pouvons donc aboutir à une situation de blocage différente de la précédente dans le sens où les activités A et B_i pourront, vraisemblablement, toujours effectuer des échanges de données, mais où aucun de ces échanges ne pourra corriger cette incohérence. L'activité A sera donc dans l'impossibilité de conclure son travail avec l'activité B_j puisque cette incohérence persistera.

Pour illustrer cette notion de conflit entre schémas de coopération, reprenons le conflit détaillé sur la figure 4.14 (page 106) dans le cas de notre exemple support. Cet exemple de conflit est repris figure 5.10. L'activité `urbaniste` (transaction t_2) partage le document `plan` (objet *doc*) à la fois en tant que cliente dans une relation rédacteur/relecteur avec l'activité `architecte` (transaction t_1), et en écriture coopérative avec l'activité `pompier` (transaction t_3).

Figure 5.10 Exemple de Conflit entre Schémas de Coopération

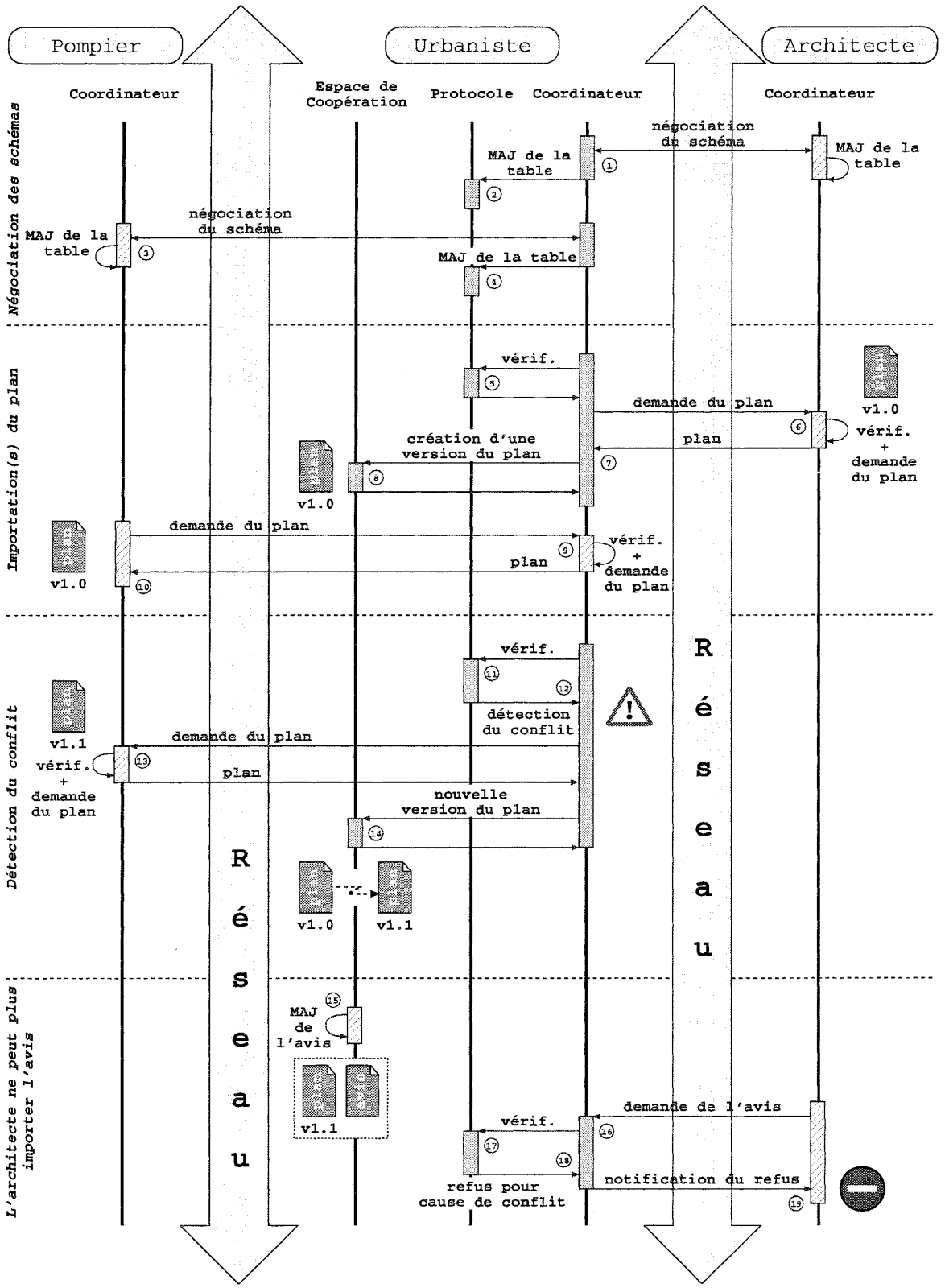


Cette situation est représentée en termes d'interactions entre les différents composants de notre architecture sur le diagramme de séquence de la figure 5.11. Les activités négocient tout d'abord les schémas de coopération qu'elles vont utiliser lors de leurs échanges (étapes 1 à 4). L'urbaniste importe ensuite le plan que lui fourni l'architecte (étapes 5 à 8), puis le transmet à son tour au pompier (étapes 9 et 10).

C'est au moment où l'urbaniste voudra importer les modifications au plan effectuées par le pompier (i.e. une nouvelle version de ce plan) qu'apparaîtront les problèmes d'interactions entre les deux schémas utilisés par l'urbaniste. Le coordinateur de l'activité `urbaniste` va demander à son protocole de vérifier si l'urbaniste a bien le droit d'importer le plan du pompier (étape 11). Bien qu'au sens du protocole d'écriture coopérative négocié entre l'urbaniste et le pompier cet échange soit autorisé, il serait tout de même souhaitable d'informer l'activité `urbaniste` (étape 12) que si elle rédige son avis "à partir" de cette nouvelle version du plan importée du pompier, elle ne pourra pas diffuser cet avis à l'architecte puisque cet avis "fera référence" à une version du plan dont ne dispose pas l'architecte. En outre, en tant que cliente (pour le plan) dans la relation rédacteur/relecteur, l'activité `urbaniste` ne pourra pas "propager" ces modifications du plan à l'activité `architecte`. L'utilisateur (i.e. l'urbaniste) peut alors soit annuler l'échange et éviter ainsi le conflit (attitude pessimiste), soit continuer et importer la nouvelle version du plan en repoussant à plus tard ce problème avec l'architecte (attitude optimiste).

Dans le cas où l'urbaniste décide de procéder à cette importation (étapes 13 et 14), il rédigera alors son avis (étape 15) à partir de la dernière version en date du plan, i.e.

Figure 5.11 Exemple d'Utilisation : Conflit entre Schémas de Coopération



celle qui lui a été fournie par le pompier. Si maintenant l'architecte demande à l'urbaniste de lui envoyer son avis[34] (étape 16), cet échange sera refusé par l'un ou l'autre des coordinateurs (étapes 17 à 19). En effet, une des règles de coopération 3 et 4 du schéma "rédacteur/relecteur" (défini au paragraphe 4.3.1 par le prédicat *writer_reviewer* rappelé ci-dessous) sera violée. Ces règles imposent qu'au moment où l'avis est transféré de l'urbaniste à l'architecte, la version courante du plan au niveau de l'urbaniste doit correspondre à une version du plan importée de l'architecte (et non modifiée depuis). Ce n'est bien évidemment pas le cas puisque l'urbaniste a pris en compte les modifications au plan effectuées par le pompier. Par conséquent, l'architecte ne pourra pas importer l'avis de l'urbaniste. Le schéma "rédacteur/relecteur" évite ainsi que cette opération n'introduise une incohérence dans la base de données locale de l'architecte (i.e. un avis faisant référence à une version du plan inconnue de l'architecte).

$$
\begin{aligned}
&writer\_reviewer(t_{wri}, t_{rev}, Doc \cup Rev) \equiv \\
&\left\{
\begin{array}{l}
client\_server(t_{rev}, t_{wri}, Doc) \\
\\
client\_server(t_{wri}, t_{rev}, Rev) \\
\\
\forall\, rev \in Rev \quad (op \in Read_{t_{wri}}[rev_{t_{rev}}] \cup Write_{t_{rev}}[rev_{t_{wri}}]) \wedge (op \in H) \Rightarrow \\
\quad \forall\, doc \in Doc \quad \nexists\, p \in Write^{(doc)} \\
\qquad (LastOcc(Read_{t_{rev}}[doc_{t_{wri}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \quad \wedge \\
\qquad (LastOcc(Write_{t_{wri}}[doc_{t_{rev}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \\
\\
\forall\, doc \in Doc \quad (op \in Read_{t_{rev}}[doc_{t_{wri}}] \cup Write_{t_{wri}}[doc_{t_{rev}}]) \wedge (op \in H) \Rightarrow \\
\quad \forall\, rev \in Rev \quad \nexists\, p \in Write^{(rev)} \\
\qquad (LastOcc(Read_{t_{wri}}[rev_{t_{rev}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \quad \wedge \\
\qquad (LastOcc(Write_{t_{rev}}[rev_{t_{wri}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op)
\end{array}
\right.
\end{aligned}
$$

Résolution des Conflits

Pour résoudre un tel conflit nous envisageons deux solutions: l'urbaniste peut soit tenter de re-négocier le schéma de coopération "rédacteur/relecteur" qu'il utilise avec l'architecte, soit modifier son avis en se basant cette fois sur la dernière version du plan qu'il a importée de l'architecte.

- **Re-négociation du schéma de coopération**: Si l'urbaniste et l'architecte s'accordent sur un nouveau schéma de coopération pour le partage du plan ("écriture coopérative" par exemple), ils modifient alors chacun leur table de coopération. Il faut toutefois être conscient que ce nouveau schéma peut à son tour générer de nouveaux conflits avec les autres schémas déjà présents dans leurs tables de coopération respectives.

  L'architecte peut alors réitérer sa demande d'importation de l'avis auprès par l'urbaniste. Leurs coordinateurs détectent de nouveau que cet avis a été rédigé à par-

34. Sous-entendu: l'avis concernant le plan de l'architecte.

tir d'une version du plan que ne possède pas l'architecte, mais du fait de la re-négociation, l'architecte peut maintenant importer cette nouvelle version du plan puis l'avis associé. Cette solution est représentée figure 5.12.

- **Retour à une ancienne version du plan**: Soit l'urbaniste décide de ne pas diffuser la nouvelle version du plan, soit l'architecte refuse de la prendre en compte. Dans un cas comme dans l'autre, l'urbaniste doit "revenir" à la dernière version du plan qu'il a importée de l'architecte. Cela peut éventuellement l'obliger à modifier son avis afin que ce dernier fasse bien référence à la bonne version du plan.

  L'architecte peut alors importer l'avis produit par l'urbaniste puisqu'il possède bien la version du plan à partir de laquelle a été rédigé cet avis. Cette solution est représentée figure 5.13.

Si la solution du retour à une ancienne version suppose simplement que les espaces de coopération fournissent des fonctionnalités de gestion des versions des objets partagés, la solution de la re-négociation du schéma de coopération nécessite de formaliser l'opération de re-négociation (schémas pouvant être re-négociés, contraintes à vérifier, ...). Nous ne détaillerons pas d'avantage ce concept de re-négociation qui fait partie des perspectives aux travaux présentés dans cette thèse.

5.1.5 Bilan

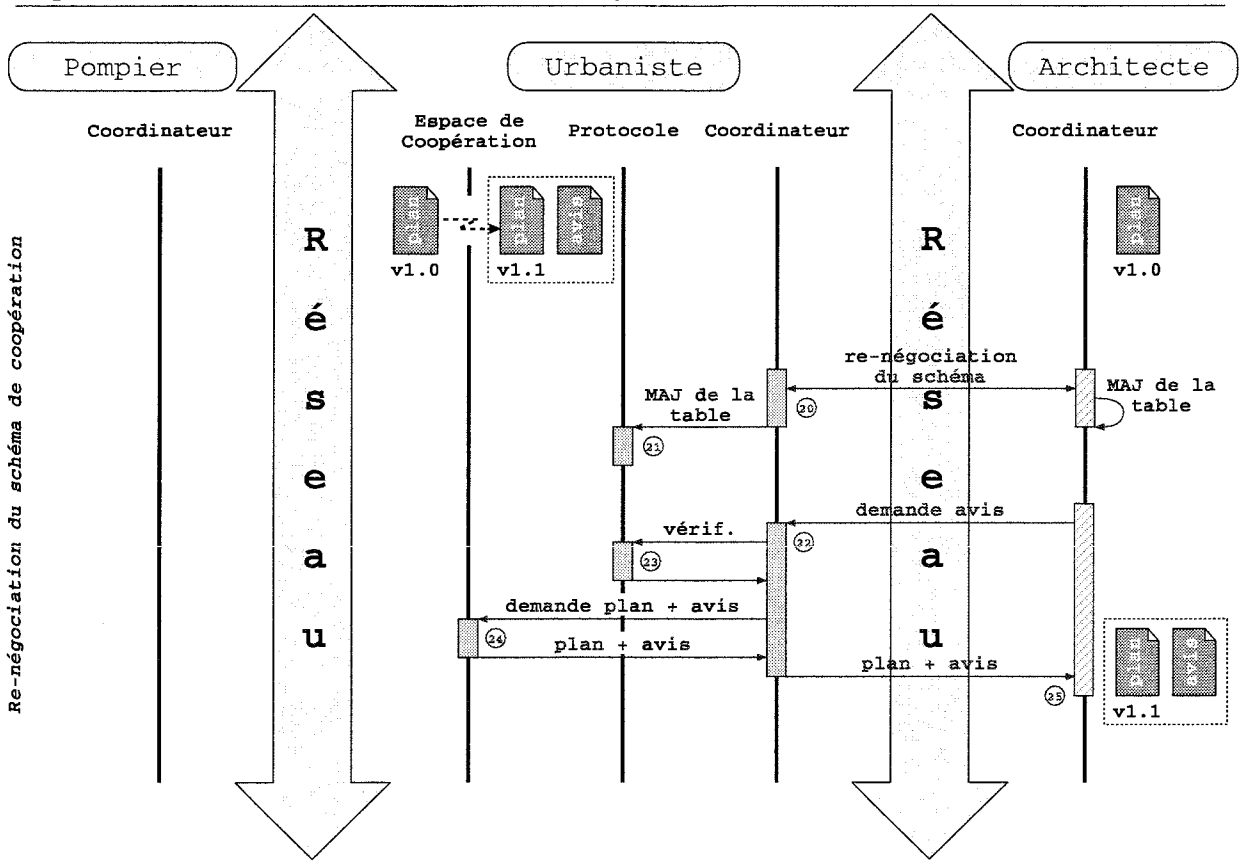
Dans cette section, nous venons de présenter l'architecture de notre système de coopération. Celle-ci permet aux activités de coopérer en s'échangeant des documents entre leurs **espaces de coopération** respectifs. La coordination de tels échanges entre deux activités est assurée par un schéma de coopération qu'elles auront préalablement négocié. Chaque activité possède ainsi une table de coopération qui lui permet de connaître, pour une activité et un document donnés, le schéma à vérifier (i.e. les règles de coopération à respecter) et son rôle le cas échéant (ex: rôles "client" et "serveur" du schéma "client/serveur"). Le **protocole** d'une activité (cf. figure 5.5 page 121) peut ainsi être considéré comme étant la "composition" de tous les "protocoles élémentaires" (les schémas de coopération) présents dans sa table de coopération. Ces schémas de coopération sont ensuite utilisés comme paramètres du contrôle des interactions réalisé par le **coordinateur**.

Il s'agit bien d'une architecture d'égal-à-égal dans laquelle chaque activité est responsable de ses propres interactions avec les autres activités du système. Cette architecture est donc conforme au modèle de transactions coopératives distribués que nous avons défini dans la section 4 et répond exactement à ses objectifs, à savoir:

- les objets partagés sont stockés localement sur les transactions
- le contrôle des échanges est réalisé localement sur chaque transaction
- les transactions peuvent négocier les schémas de coopération

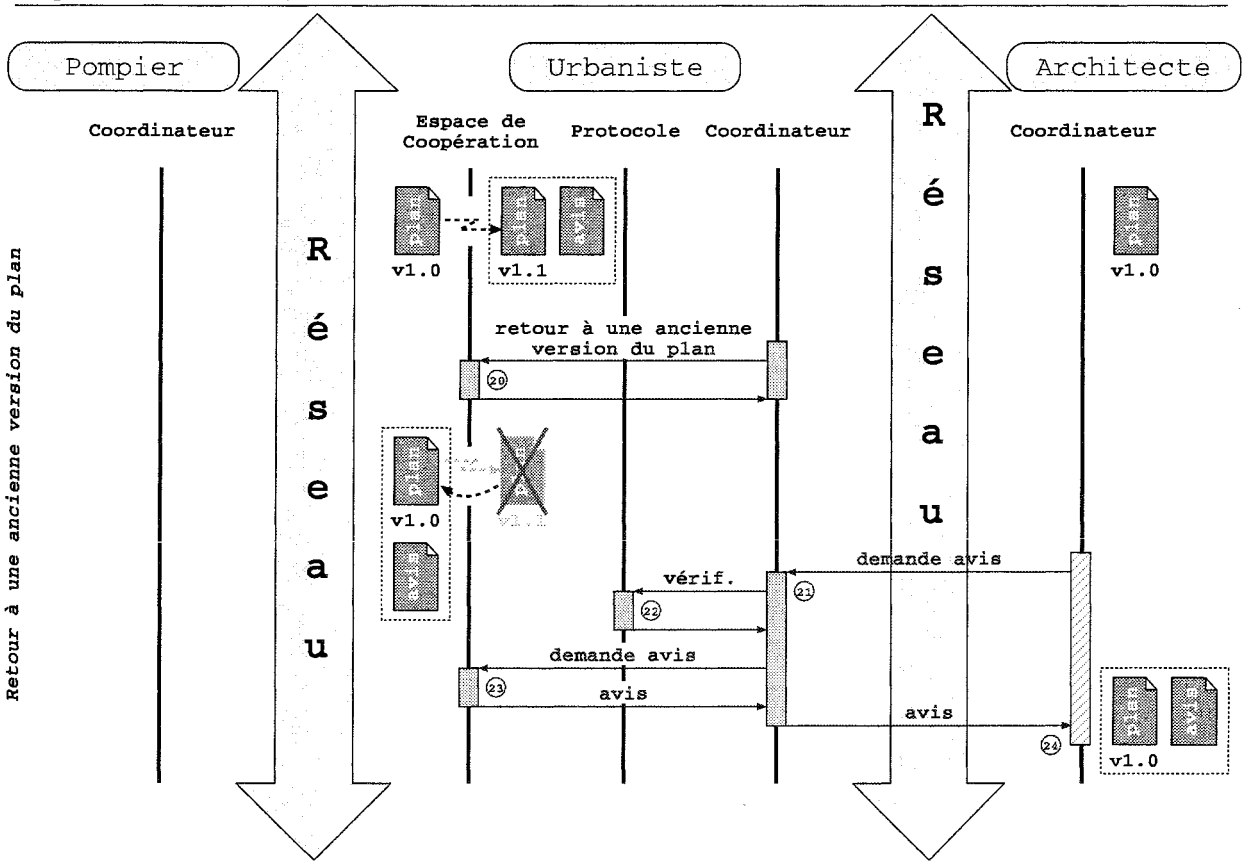
La section suivante est consacrée à la mise en œuvre de cette architecture: choix de la plate-forme logicielle (bus à objets répartis CORBA) puis réalisation du prototype *DisCOO* (définition des services de base de la coopération, implémentation des schémas de coopération, ...).

Figure 5.12 Exemple d'Utilisation : Re-Négociation du Schéma de Coopération



- **Re-négociation du schéma de coopération:**
  20. Les deux activités doivent tout d'abord re-négocier, par l'intermédiaire de leurs coordinateurs respectifs, un nouveau schéma de coopération.
  21. Une fois le schéma choisi, chaque activité met à jour sa table de coopération.
  22. Le coordinateur de l'architecte peut alors réitérer sa demande d'importation de l'avis auprès du coordinateur de l'urbaniste.
  23. Le coordinateur de l'urbaniste vérifie auprès de son propre protocole que cet échange est autorisé. Afin de présever la cohérence entre le plan et l'avis, si l'architecte veut importer l'avis, il devra également importer le plan mis à jour par l'urbaniste.
  24. Les deux activités ayant donné leur accord, l'échange peut avoir lieu. Le coordinateur de l'urbaniste demande donc à son espace de coopération de lui fournir à la fois l'avis et le plan.
  25. Le coordinateur de l'urbaniste transmet le plan puis l'avis au coordinateur de l'architecte qui se charge de mettre à jour son espace de coopération.

Figure 5.13 Exemple d'Utilisation : Retour à une Ancienne Version du Plan



- **Retour à une ancienne version du plan:**
  20. Le coordinateur de l'urbaniste demande à son espace de coopération (qui doit être capable de gérer l'historique des versions d'un document) de "revenir" (i.e. *"rollback"*) à la dernière version du plan qui a été importée de l'architecte. Cela peut également nécessiter quelques modifications de l'avis afin que les deux documents soient cohérents.
  21. Le coordinateur de l'architecte peut alors réitérer sa demande d'importation de l'avis auprès du coordinateur de l'urbaniste.
  22. Le coordinateur de l'urbaniste vérifie auprès de son propre protocole que cet échange est autorisé.
  23. Les deux activités ayant donné leur accord, l'échange peut avoir lieu. Le coordinateur de l'urbaniste demande donc à son espace de coopération de lui fournir l'avis.
  24. Le coordinateur de l'urbaniste transmet l'avis au coordinateur de l'architecte qui se charge de mettre à jour son espace de coopération.

5.2 Mise en Œuvre

Concernant la mise en œuvre des *DisCOO*-transactions et de l'architecture présentée ci-dessus, notre objectif est de ne pas développer "un système propriétaire supplémentaire". L'idée consiste plutôt à réaliser une infrastructure intégrant les mécanismes de base nécessaires à la coopération et identifiés dans la section précédente. Cette infrastructure servira ensuite de base pour la programmation d'applications coopératives distribuées.

Cette mise en œuvre est réalisée en deux étapes. La première consiste tout d'abord à choisir une plate-forme logicielle permettant le développement d'applications distribuées sur un réseau de stations. La seconde concerne la réalisation proprement dite du prototype *DisCOO*, c'est-à-dire la mise en œuvre des différents composants de notre architecture (cf. figure 5.5) sous la forme de services de coopération.

5.2.1 Plate-Forme Logicielle

Comme nous l'avons indiqué au chapitre 3, les environnements d'aide à la coopération existants tels que les gestionnaires de configurations, les outils de gestion des procédés, les systèmes transactionnels classiques, ou les outils d'aide au travail coopératif, ne sont pas adaptés à nos besoins de **coopération** (interactions entre les activités), de **distribution** (autonomie des activités, contrôle décentralisé) et d'**hétérogénéïté** (négociation de schémas de coopération). Pour mettre en œuvre les résultats de nos travaux nous avons donc choisi de développer notre propre infrastructure.

Afin de simplifier le développement d'applications distribuées, un certain nombre de plates-formes logicielles ont été proposées et évitent ainsi aux programmeurs de devoir accéder directement et manuellement aux couches basses du réseau. Ces plates-formes peuvent être classées en quatre catégories: appels de procédures à distance (RPC[35], Java RMI[36]), mémoires virtuelles partagées (PVM[37], PerDiS [Sha97]), systèmes orientés vers la persistance des objets (Shore, Arjuna), systèmes répartis à objets (modèle CORBA[38] de l'OMG[39], architecture ActiveX/DCOM de Microsoft).

Ces solutions sont fondamentalement différentes les unes des autres. Pour justifier les choix que nous avons effectués, nous rappelons ci-dessous nos motivations et nos besoins pour le développement de notre infrastructure:

- **organisation d'égal-à-égal**: Nous voulons éviter toute forme de dépendance des activités vis-à-vis d'un quelconque site central. Les partenaires doivent être autonomes, et lorsque deux d'entre eux décident de s'échanger des données, ils sont les seuls responsables du bon déroulement de cet échange.
- **autonomie**: Si les partenaires travaillent sur des sites géographiquement distincts et éventuellement déconnectés du réseau, cette autonomie permet de réduire les surcoûts de communication par rapport à la centralisation. En effet, toutes les données

35. *Remote Procedure Call*
36. *Remote Method Invocation*
37. *Parallel Virtual Machine*
38. *Common Object Request Broker Architecture* [OMG95b, Wei96]
39. Object Management Group

dont a besoin un partenaire pour accomplir sa tâche sont stockées localement: les objets partagés dans son espace de coopération, les informations de contrôle dans son protocole. Les performances, la fiabilité et la disponibilité du système en sont également accrues.

- **faible infrastructure informatique** des partenaires: Tous les partenaires ne disposent par forcément de stations de travail reliées par un réseau local. L'infrastructure proposée devra donc pouvoir être déployée tant sur des stations de travail Unix que sur des PC fonctionnant sous Windows 98/NT et éventuellement connectés à internet par l'intermédiaire d'un modem et d'un fournisseur d'accès.
- **intégration**: L'idée est de définir uniquement la "glue" permettant de faire coopérer les systèmes et les applications déjà utilisés par les partenaires. Notre objectif est donc de s'affranchir des problèmes liés à la distribution (connexions réseau, hétérogénéïté des environnements PC/stations Unix, ...) en utilisant les services fournis par la plate-forme logicielle et de proposer nos propres mécanismes de coopération sous la forme de services pouvant à leur tour être intégrés dans les systèmes existants pour leur ajouter des fonctionnalités dédiées à la coopération.
- **extensibilité**: L'ajout de nouvelles activités ou de nouvelles ressources partagées ne doit pas perturber le déroulement des activités déjà présentes dans le système. Cette contrainte est tout à fait légitime puisque chaque activité n'a qu'une vue locale du système. Par conséquent, si l'activité ou la ressource ajoutée ne concerne pas cette activité, ses performances ne doivent pas en être pénalisées.
- **évolutivité**: Notre infrastructure doit également permettre l'ajout de nouvelles formes de coopération telles que la gestion des procédés exécutés par exemple. D'où l'idée de définir non pas un système monobloc mais plutôt un ensemble de services de base.

Ces différents points constituent autant de raisons qui nous ont guidés vers le choix du système réparti à objets CORBA. Cette plate-forme logicielle nous permet ainsi d'être indépendants de l'environnement sous-jacent et de ses mécanismes de communication.

Présentation de CORBA

La norme CORBA (*Common Object Request Broker Architecture*) a été développée par l'OMG dans le but de simplifier le développement d'applications distribuées orientées objets en automatisant de nombreuses tâches de programmation réseau telles que la désignation, la localisation et l'activation des objets, le routage des requêtes (appels de méthode), le codage et le décodage des paramètres de ces requêtes, la gestion des exceptions, ...

Les spécifications de CORBA 1.1 publiées en 1992 (voir [OMG92]) ne définissaient que le langage de définition d'interfaces IDL[40], la traduction de spécifications IDL dans différents langages de programmation (C++, Java, Ada, ...), ainsi que les interfaces (API) de l'ORB (*Object Request Broker*). Il était ainsi possible d'écrire des programmes

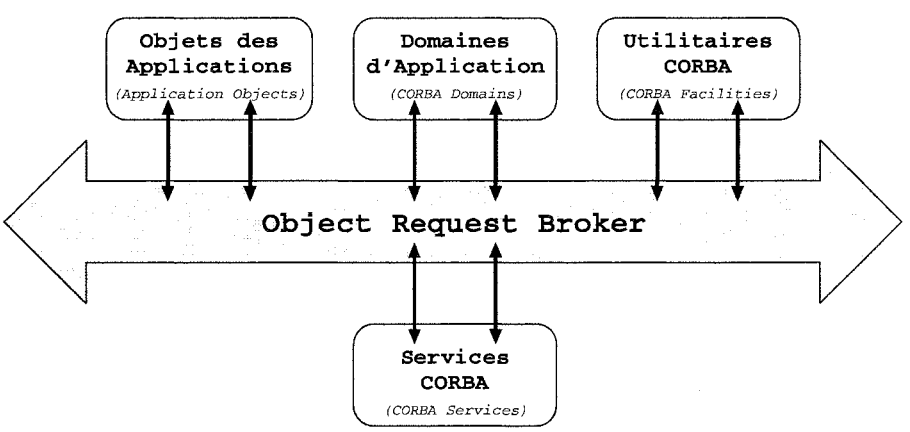
40. *Interface Definition Language*

portables pouvant fonctionner au-dessus de n'importe quel ORB du marché conforme aux spécifications CORBA. Dans sa version 2.0 [OMG95b], apparue en 1995, CORBA permet à des objets d'interopérer quel que soit leur langage d'implémentation (C++, Java, Ada, ...), leur environnement d'exécution (Windows 98/NT, Unix, MacOS, ...), leur réseau de communication ou l'ORB auquel ils sont connectés (via le protocole IIOP[41]). Les principes de CORBA sont:

- une séparation stricte entre l'interface et l'implémentation d'un objet
- la transparence de la localisation des objets
- la transparence de l'accès aux objets
- le typage des références objet par les interfaces
- l'héritage multiple d'interfaces
- l'organisation des fonctionnalités en services

La figure 5.14 représente les cinq catégories de composants du modèle de référence CORBA tel qu'il a été défini par l'OMG: l'ORB, les services CORBA, les utilitaires CORBA, les domaines d'application CORBA et les objets des applications.

Figure 5.14 Modèle de Référence CORBA



- L'**ORB** est le bus logiciel dont la fonction essentielle est d'offrir tous les mécanismes nécessaires pour, suite à une invocation d'opération provenant d'un objet client, trouver une implantation d'objet serveur capable de fournir ce service, préparer cette implantation d'objet à recevoir l'invocation d'opération, puis transmettre les paramètres, les résultats et les exceptions entre ces deux objets. Ceci est réalisé de manière transparente, c'est-à-dire que l'ORB fournit l'interopérabilité entre des objets pouvant se trouver sur différentes machines dans des environnements distribués hétérogènes, voire même sur des ORB différents grâce au protocole IIOP.
- Les **services CORBA** (*CORBA services*) introduisent des services complémentaires à ceux de l'ORB mais néanmoins fondamentaux pour développer des applications réparties. Les services CORBA sont indépendants d'une application spécifique.

41. *Internet Inter-ORB Protocol*

Les services CORBA actuellement standardisés couvrent de nombreuses fonctions telles que le nommage (désignation d'un objet dans un contexte défini), la notification d'événements (envois de messages entre objets producteurs et consommateurs), la gestion de propriétés (attributs associés dynamiquement aux objets), les transactions (gère des unités de travail atomiques impliquant plusieurs objets serveurs), ... Les spécifications des services CORBA standardisés sont détaillées dans le document [OMG95a].

- Les **utilitaires CORBA** (*CORBA facilities*) sont similaires aux services CORBA mais sont plus orientés vers l'utilisateur humain. Un des premiers utilitaires CORBA publiés par l'OMG est la gestion de documents composites distribués (*Distributed Document Component Facility*) basée sur le produit *OpenDoc*.
- Le rôle des **domaines d'application CORBA** (*CORBA domains*) est identique à celui des services et utilitaires CORBA. Il s'agit de services spécialisés pour certains secteurs industriels tels que la comptabilité, la sécurité, l'imagerie, ...
- Les **objets des applications** (*application objects*) représentent quant à eux les fonctions spécifiques à une application donnée qui ne sont pas couvertes par les autres composants du standard CORBA. S'il apparaît qu'un tel service est de plus en plus fréquemment utilisé, celui-ci peut devenir candidat à une future standardisation de l'OMG.

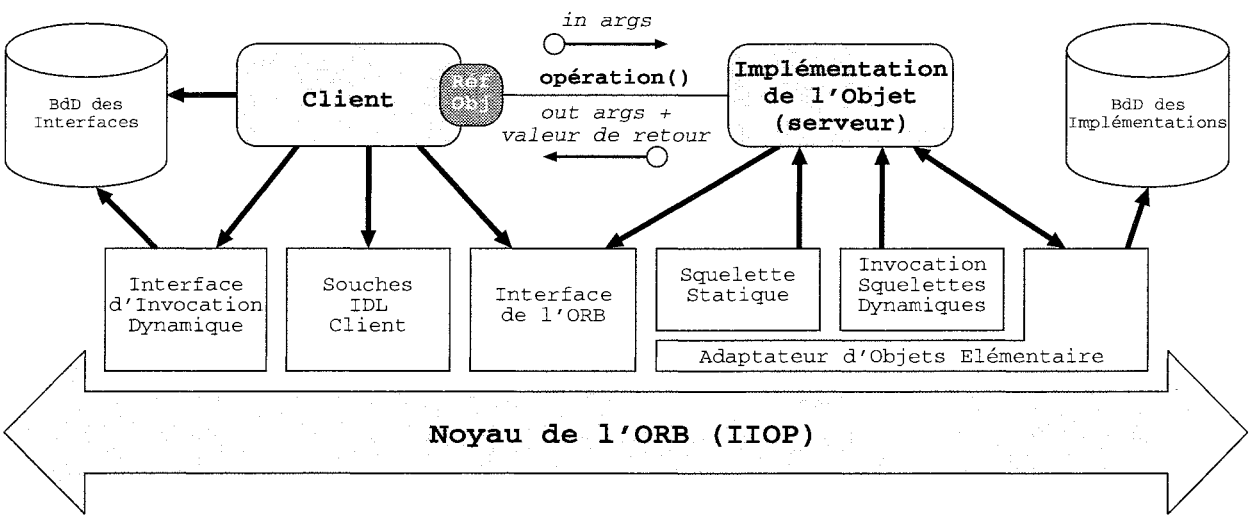
La figure 5.15 présente les différents éléments de l'architecture d'un ORB CORBA[42]. Pour illustrer le rôle de chacun de ces composants nous allons expliquer de quelle manière un objet client invoque une méthode d'un objet serveur. A noter toutefois que CORBA est une architecture d'égal-à-égal, c'est-à-dire que les termes "client" et "serveur" n'ont de signification que par rapport à un appel de méthode. L'objet "client" est celui qui invoque la méthode et l'objet "serveur" est celui qui implémente le service demandé. Au sein d'une telle architecture chaque objet est donc à la fois client et serveur.

1. La première étape pour l'objet client consiste tout d'abord à se connecter à l'ORB. L'objet client utilise pour cela les fonctions offertes par l'**interface de l'ORB** (*ORB interface*). L'ORB est en effet une entité logique pouvant être mise en œuvre de différentes manières (un ou plusieurs processus ou un ensemble de bibliothèques par exemple). Afin de découpler les applications de ces détails d'implémentation, les spécifications CORBA définissent une interface abstraite pour cet ORB. Cette interface fournit différentes fonctions telles la conversion de références d'objets en chaînes de caractères et vice-versa, la création de listes d'arguments pour les requêtes effectuées au travers de la DII décrite ci-dessous.
2. Afin de pouvoir effectuer une requête, l'objet client doit posséder une référence vers l'objet serveur[43], connaître l'interface de cet objet ainsi que le nom de la méthode

42. Les noms des différents composants ont été volontairement laissé en anglais car il nous semble plus pratique de parler du BOA (*Basic Object Adapter*) que de l'*adaptateur d'objet élémentaire.*

43. Pour obtenir cette référence vers l'objet serveur, l'objet client peut faire appel au service de nommage en désignant l'objet serveur par son nom. L'objet client peut également utiliser un service de requête (*Trading Service*) pour demander à l'ORB de lui indiquer un objet pouvant fournir un service donné.

Figure 5.15 Architecture d'un ORB CORBA



qu'il désire invoquer. En ce qui concerne l'inteface de l'objet serveur, l'objet client peut procéder de deux manières:

- Soit il connaît à priori cette interface. L'invocation de la méthode se fait alors au travers de la **souche** (*stub*) générée automatiquement par le compilateur IDL à partir de la spécification IDL de l'objet serveur.
- Soit il ne connaît pas cette interface et interroge alors l'**interface d'invocation dynamique** (*dynamic invocation interface*) qui stocke dans une base de données (*interface repository*) les interfaces des objets qui se sont fait enregistrer. Cette interface permet au client d'accéder aux mécanismes internes de l'ORB pour construire et invoquer dynamiquement des requêtes sur les objets. Au travers de la DII le client peut ainsi interroger l'ORB pour lui demander quelles sont les méthodes pouvant être invoquées sur tel objet, puis quels sont les paramètres nécessaires pour invoquer telle méthode, ...

Dans les deux cas, l'interface que voit le client est complètement indépendante de l'endroit où est situé l'objet serveur et du langage dans lequel cet objet est implémenté.

3. L'appel de méthode est ensuite transmis à l'**ORB** qui fournit les mécanismes permettant aux objets clients d'invoquer des méthodes sur des objets distants de manière transparente. Ceci doit être aussi simple que d'appeler une méthode sur un objet local, c'est-à-dire `obj->op(args)`. C'est l'ORB qui est chargé de trouver une implémentation de l'objet serveur, de l'activer automatiquement si nécessaire, de délivrer la requête à cet objet, puis de renvoyer la réponse (résultats et/ou exceptions) vers l'objet client.
4. La requête arrive ensuite au **BOA** (*basic object adapter*). Le BOA est une interface visant à supporter un large éventail d'implémentations objet. Il fournit les fonctions

suivantes:

- génération et interprétation des références objet
- authentification du principal effectuant l'appel
- activation et désactivation des objets individuels
- activation et désactivation de l'implémentation
- invocation des méthodes au travers de **skelettes**

Le BOA supporte des implémentations constituées d'un ou plusieurs objets pouvant répondre aux requêtes pour un même service. Les implémentations sont stockées dans une base de données (*implementation repository*). Le système peut ainsi installer, démarrer et arrêter des implémentations selon les besoins. Outre le BOA, les adaptateurs d'objets peuvent être classés en quatre catégories:

- serveur partagé: un processus pour tous les objets
- serveur non partagé: un processus par objet
- serveur par méthode: un processus par requête
- serveur persistant: l'OA fonctionne en serveur partagé mais ses requêtes sont traitées par un ordonnanceur

5. Le BOA peut transmettre la requête à l'objet serveur soit par l'intermédiaire d'un squelette statique soit au travers de la DSI (*dynamic skeleton interface*). Cette interface est à l'objet serveur ce que la DSI est à l'objet client. La DSI permet à l'ORB de délivrer des requêtes à une implémentation d'objet qui, au moment de la compilation, ne connaît pas le type de l'objet qu'elle implémente. Le fait que l'implémentation utilise les squelettes créés statiquement par le compilateur IDL ou des squelettes dynamiques est complètement transparent pour l'objet client qui a initié la requête.

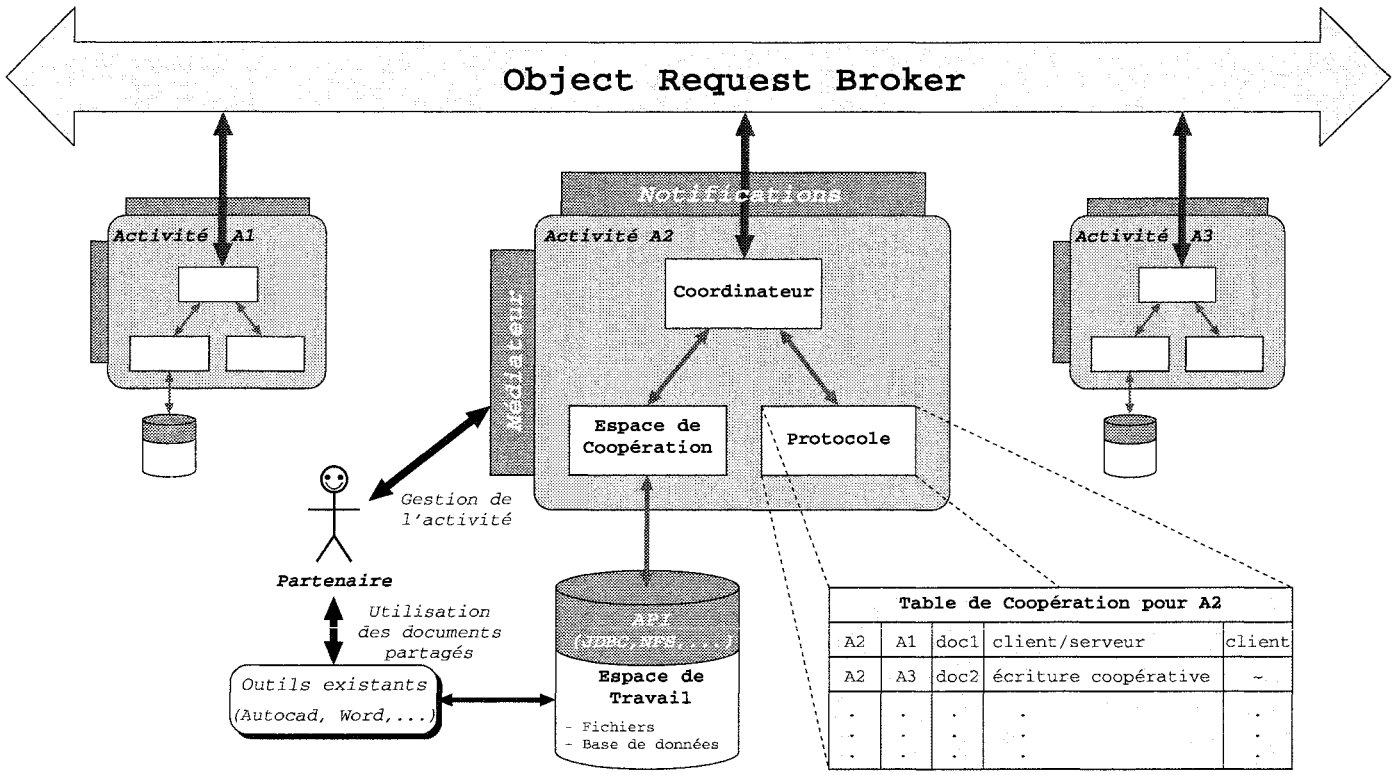
6. Finalement, la requête arrive enfin à l'objet serveur qui implémente le service demandé (défini par une interface IDL). Cette implémentation peut être codée en C++, Java, ADA, ..., des objets implémentés dans des langages différents pouvant bien entendu interagir sur un même ORB (voire entre différents ORB via le protocole IIOP).

CORBA constitue donc la plate-forme logicielle idéale pour nous permettre de mettre en œuvre les travaux présentés dans cette thèse. Les différents composants de notre architecture seront regroupés au sein d'un **service de coopération** faisant partie de la catégorie des **domaines d'application CORBA** (*CORBA domains*). Il s'agit donc réellement de définir un service fournissant les mécanismes de base nécessaires au développement d'applications coopératives distribuées (notre domaine).

5.2.2 Prototype *DisCOO*

Les interactions entre les différentes activités de notre système, ou plus précisément entre leurs coordinateurs respectifs, seront donc réalisées au travers d'un ORB (cf. figure 5.16). Le fait que ces activités soient distribuées leur sera ainsi complètement transparent.

Figure 5.16 Mise en Œuvre de notre Architecture au-dessus d'un ORB

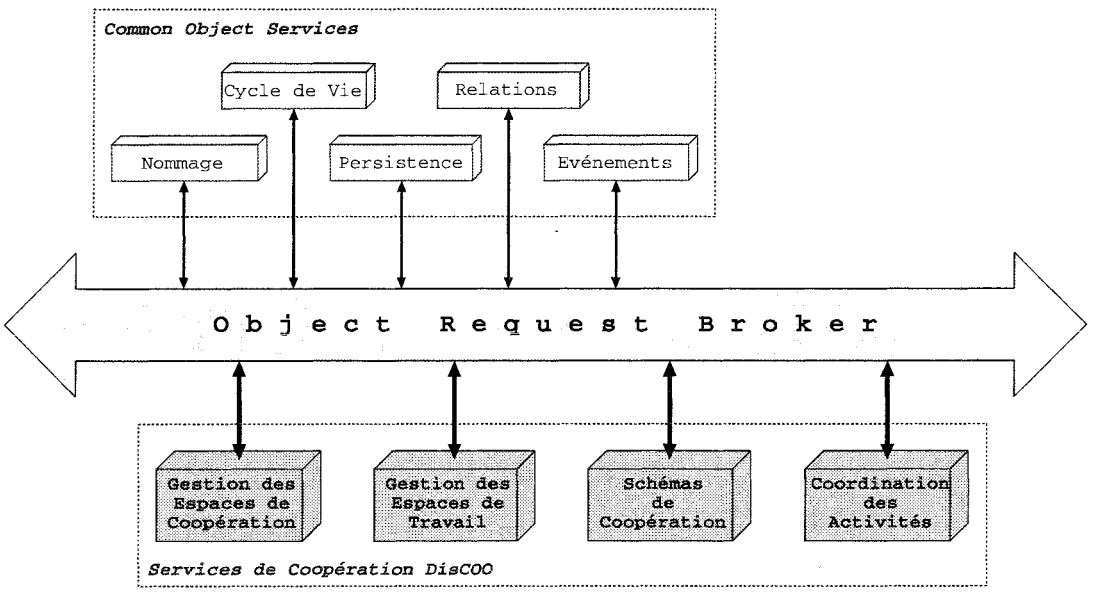


Comme nous l'avons indiqué au début de ce chapitre à la section 5.1, nous avons identifié quatre services essentiels pour la coopération: un service d'**espace de coopération**, un service d'**espace de travail**, un service de **coordination** et un service de **protocole**. Nous les avons développés en tant que services CORBA de manière à ce qu'ils puissent être à leur tour utilisés par d'autres services ou objets CORBA, au même titre que les services CORBA standard (cf. figure 5.17). En fait, tous les composants de notre architecture sont eux-même des objets CORBA. Les fonctions de chacun de ces services sont les suivantes:

- Service d'**espace de coopération**: Il définit les composants nécessaires à la gestion de la base de données locale d'une activité: espace de coopération (le référentiel local), ressource, identificateur d'une ressource, contenu d'une ressource (les données proprement dites), version d'une ressource.
- Service d'**espace de travail**: Il s'agit d'une interface permettant à un espace de coopération d'externaliser le contenu d'une ressource (les données) vers un espace de travail dans lequel l'utilisateur pourra faire appel à ses applications habituelles (Word, Autocad, emacs, gcc, ...) pour manipuler les données.
- Service de **coordination**: Ce service permet de garantir que toutes les requêtes transmises au service d'espace de coopération sont validées par le service de protocole. Le composant principal de ce service est le coordinateur. C'est dans celui-ci que sont codés les axiomes de respect des schémas négociés définis au chapitre 4 (définition 4.17).

- Service de **protocole** : Il est chargé, pour une activité donnée, du contrôle effectif des interactions de cette activités avec les autres activités du système, à savoir:
  - gestion des contrats négociés avec les autres activités (table de coopération)
  - gestion de l'histoire locale de cette activité (journalisation des événements)
  - vérification de tel ou tel schéma de coopération par rapport à l'histoire locale courante

Figure 5.17 Services de Coopération de *DisCOO*



La mise en œuvre des résultats de nos travaux présentés au chapitre 4 concerne donc essentiellement l'implémentation des services de coordination et de protocole. Au niveau du prototype *DisCOO*, l'implémentation des services d'espace de coopération et d'espace de travail a été simplifiée, notre objectif étant de fournir les fonctionnalités minimales pour développer un exemple d'application coopérative distribuée (cf. section 5.2.3) démontrant le bon fonctionnement des services de coordination et de protocole. Ainsi, dans un premier temps, un espace de coopération ne gère qu'une liste des versions successives d'une ressource donnée (et non un graphe) et les seules ressources pouvant être partagées sont des objets de type fichier que l'on peut enregistrer dans un espace de travail matérialisé par un répertoire sur le disque dur de la machine. L'utilisateur peut ainsi manipuler ses ressources (sous forme de fichiers) à l'aide de ses applications habituelles.

Service de Coordination

Le principal composant de ce service est le coordinateur. Son rôle au sein de notre architecture a été mis en évidence lorsque nous avons détaillé, pas à pas, les différentes étapes d'un échange de données entre deux activité (cf. figure 5.7 page 123). Le coordinateur d'une activité est chargé de "filtrer" ses échanges avec les autres activités du système, c'est-à-dire de refuser tout échange qui ne vérifierait pas le(s) schéma(s) de coopération négocié(s). Sinon la requête est transmise à l'espace de coopération de l'activité.

Dans la version actuelle du prototype *DisCOO*, cette vérification implémente strictement l'axiome $n°$ 2 de la définition 5.1. Cela signifie que pour une opération de transfert concernant l'objet *ob* réalisée entre les activités A et B, seul le schéma de coopération négocié entre A et B pour le partage de l'ensemble d'objets O (tel que $ob \in O$) est vérifié (politique optimiste).

En ce qui concerne l'opération de négociation, l'implémentation du coordinateur est conforme aux axiomes $n°$ 1 et $n°$ 3, c'est-à-dire que tout contrat négocié est journalisé simultanément par les deux partenaires et qu'il n'est pas possible de négocier un contrat concernant un objet figurant déjà sur un autre contrat. Cela signifie en particulier que cette implémentation du coordinateur n'autorise pas les activités à re-négocier un schéma de coopération.

Service de Protocole

Le cœur du prototype *DisCOO* est donc le service de protocole, et plus particulièrement les composants chargés de la gestion de l'histoire locale de cette activité et de la vérification d'un schéma de coopération donné par rapport à cette histoire locale.

Au paragraphe 4.3.1 nous avons définis un schéma de coopération comme étant un ensemble de règles de coopération. Ces règles sont elles-même des prédicats destinés à être évalués par rapport à l'histoire locale courante de l'activité. Par exemple, le schéma de coopération "écriture coopérative" est représenté par le prédicat *DisCOO* (qui utilise le prédicat *up_to_date*):

$$up\_to\_date(t_j, t_k, O) \equiv \forall ob \in O \quad \forall Q_{t_j}[ob_{t_k}]$$
$$(\exists t_i \ \exists p \ rvd(p_{t_i}[ob_{t_k}], LastOcc(Q_{t_j}[ob_{t_k}])) \ \wedge \ (p_{t_i}[ob_{t_k}] \rightarrow LastOcc(Q_{t_j}[ob_{t_k}]))) \Rightarrow$$
$$(\nexists p' \in Sequence(p_{t_i}[ob_{t_k}]) \quad (LastOcc(Q_{t_j}[ob_{t_k}]) \rightarrow p'_{t_i}[ob_{t_k}]))$$

$$DisCOO(t_i, t_j, O) \equiv \begin{cases} \forall t_1, t_2 \ (t_1 \in \{t_i, t_j\}, \ t_2 \in \{t_i, t_j\} - t_1) \\ \quad (\texttt{Commit}_{t_1} \in H_{t_2}) \ \Rightarrow \ up\_to\_date(t_1, t_2, O) \\ \\ \forall ob \in O \quad \forall t_1, t_2 \ (t_1 \in \{t_i, t_j\}, \ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k \ (t_k \neq t_1) \\ \quad (\texttt{Commit}_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \quad\quad rvd(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}]) \ \wedge \ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}]) \ \Rightarrow \\ \quad\quad (\texttt{Commit}_{t_k}[p_{t_k}[ob_{t_2}]] \in H_{t_2}) \\ \\ \forall ob \in O \quad \forall t_1, t_2 \ (t_1 \in \{t_i, t_j\}, \ t_2 \in \{t_i, t_j\} - t_1) \quad \forall t_k \ (t_k \neq t_1) \\ \quad (Abort_{t_1}[p_{t_1}[ob_{t_2}]] \in H_{t_2}) \ \Rightarrow \\ \quad\quad rvd(p_{t_1}[ob_{t_2}], q_{t_k}[ob_{t_2}]) \ \wedge \ (p_{t_1}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_k}[ob_{t_2}]) \Rightarrow \\ \quad\quad (Abort_{t_k}[q_{t_k}[ob_{t_2}]] \in H_{t_2}) \end{cases}$$

Plutôt que de reprogrammer des algorithmes représentant ces différentes règles de coopération **nous avons choisi d'implémenter directement ces règles sous la forme de prédicats Prolog**. Pour illustrer ceci, considérons l'exécution de la figure 5.18. Les activités `archi` (l'architecte) et `struct` (l'ingénieur-structure) s'échangent différentes versions de la ressource `plan`. Pour chacune de ces deux activités, son histoire locale est

représentée en Prolog par la liste des événements journalisés par cette activité dans l'ordre inverse de leur invocation, à savoir:

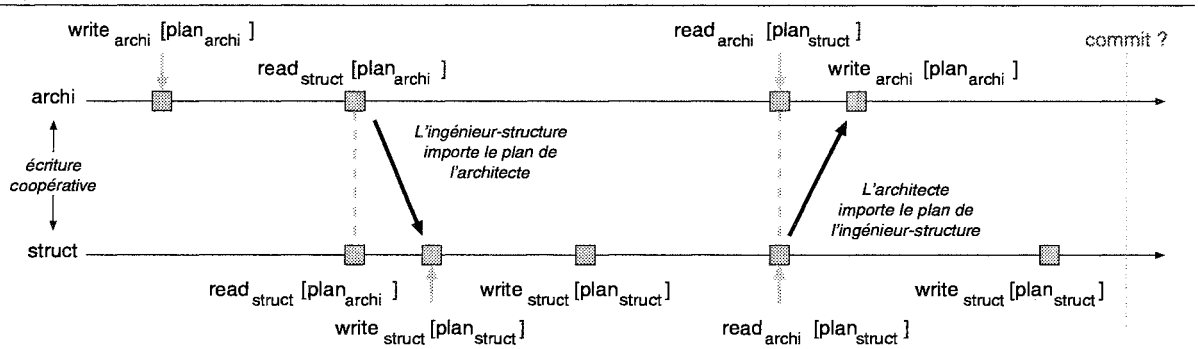
- Pour l'activité `archi` :

```
me(archi).
trace(archi,[write(archi,plan,archi),
             read(archi,plan,struct),
             read(struct,plan,archi),
             write(archi,plan,archi)]).
```

- Pour l'activité `struct` :

```
me(struct).
trace(struct,[write(struct,plan,struct),
              read(archi,plan,struct),
              write(struct,plan,struct),
              write(struct,plan,struct),
              read(struct,plan,archi)]).
```

Figure 5.18 Exemple d'Exécution



Comme indiqué ci-dessus, la première règle de coopération du prédicat *DisCOO* impose que l'activité `archi` soit "à jour" par rapport à l'activité `struct` en ce qui concerne la ressource `plan` avant de pouvoir être validée. Cette vérification sera effectuée à l'aide du programme Prolog suivant auquel il faut ajouter l'histoire locale de l'activité qui effectue le contrôle:

```
// Quelques prédicats bien utiles sur les listes

head([Tete|Queue], Tete).
tail([Tete|Queue], Queue).

find(Op, [Op|Queue]).
find(Op, [Tete|Queue]) :- find(Op, Queue).
find(Op, []) :- fail.

// On vérifie si, depuis le dernier write(Aj,Ob,Myself), il y a
// eu une opération qui aurait permis à Ai d'avoir connaissance
// de cette nouvelle valeur de Ob, i.e.:
```

```
//    - soit un read(Ai,Ob,Myself)
//    - soit un write(Myself,Ob,Ai)
//
// Par contre, si le write(Aj,Ob,Myself) est fait immédiatement
// suite à un read(Aj,Ob,Ai), cela signifie que ce write a été
// produit lorsque Aj a importé Ob depuis Ai. Donc ce write n'a
// pas à être pris en compte.
//
// Ex: On veut insérer un write(Ai,Ob,Myself). Celui-ci ne doit
// pas écraser le write(Aj,Ob,Myself) !!!

checkOverwrite([write(Aj,Ob,Myself)|Queue], List, Ai, Aj, Ob) :-
  me(Myself),
  not(eq(Ai,Aj)),
  not(head(Queue,read(Aj,Ob,Ai))),
  not(find(read(Ai,Ob,Myself), List)),
  not(find(write(Myself,Ob,Ai), List)).

checkOverwrite([Op|Queue], List, Ai, Aj, Ob) :-
  checkOverwrite(Queue, [Op|List], Ai, Aj, Ob).

// lostUpdate(A1,A2,X) indique que A1 n'a pas relu la dernière
// version de X produite par A2.

lostUpdate(A1, A2, X) :-
  me(Myself),
  trace(Myself, Tr),
  checkOverwrite(Tr, [], A1, A2, X),
  not(eq(A1, A2)).
```

Quand l'activité `archi` désirera être validée, elle sera amenée, entre autres, à demander l'autorisation à l'activité `struct`. Celle-ci vérifiera alors que l'activité `archi` a bien eu connaissance de la dernière version en date de la ressource `plan` dont elle dispose, cette version pouvant éventuellement avoir été importée depuis une autre activité. Elle enverra pour cela la requête suivante à son interpréteur Prolog:

```
lostUpdate(archi, Other, plan).
```

Si cette requête s'exécute avec succès, cela signifie qu'il existe effectivement un événement `write(Other,plan,struct)`, c'est-à-dire une opération de mise-à-jour de la ressource `plan` de l'activité `struct`, qui n'a pas été suivie d'un événement `read(archi,plan,struct)` qui aurait permis à l'activité `archi` de prendre connaissance de cette modification. L'activité `archi` a donc bel et bien manqué une mise-à-jour de la ressource `plan` de l'activité `struct`. C'est effectivement le cas sur l'exemple de la figure 5.18.

En procédant de cette manière, les schémas et règles de coopération (formalisés en ACTA) sont donc réellement implémentés sous la forme de propriétés (les prédicats Prolog) évaluées sur les histoires locales des activités. Du point de vue de la programmation cela

nous permet de créer et de modifier les schémas de coopération de façon "naturelle" par rapport à leur définition en ACTA. L'ajout de nouveaux schémas de coopération au prototype *DisCOO* en est ainsi grandement facilitée.

Programmation

Pour conclure la présentation du prototype *DisCOO* signalons également que tous les composants ont été programmés en Java au-dessus d'un ORB lui-même développé en Java (JacORB). Quant à l'interpréteur Prolog, il est lui aussi développé en Java. Le fait d'utiliser le langage Java nous permet ainsi de déployer *DisCOO* dans tout environnement (Windows 98/NT, Unix, MacOS, ...) pour lequel une machine virtuelle Java est disponible.

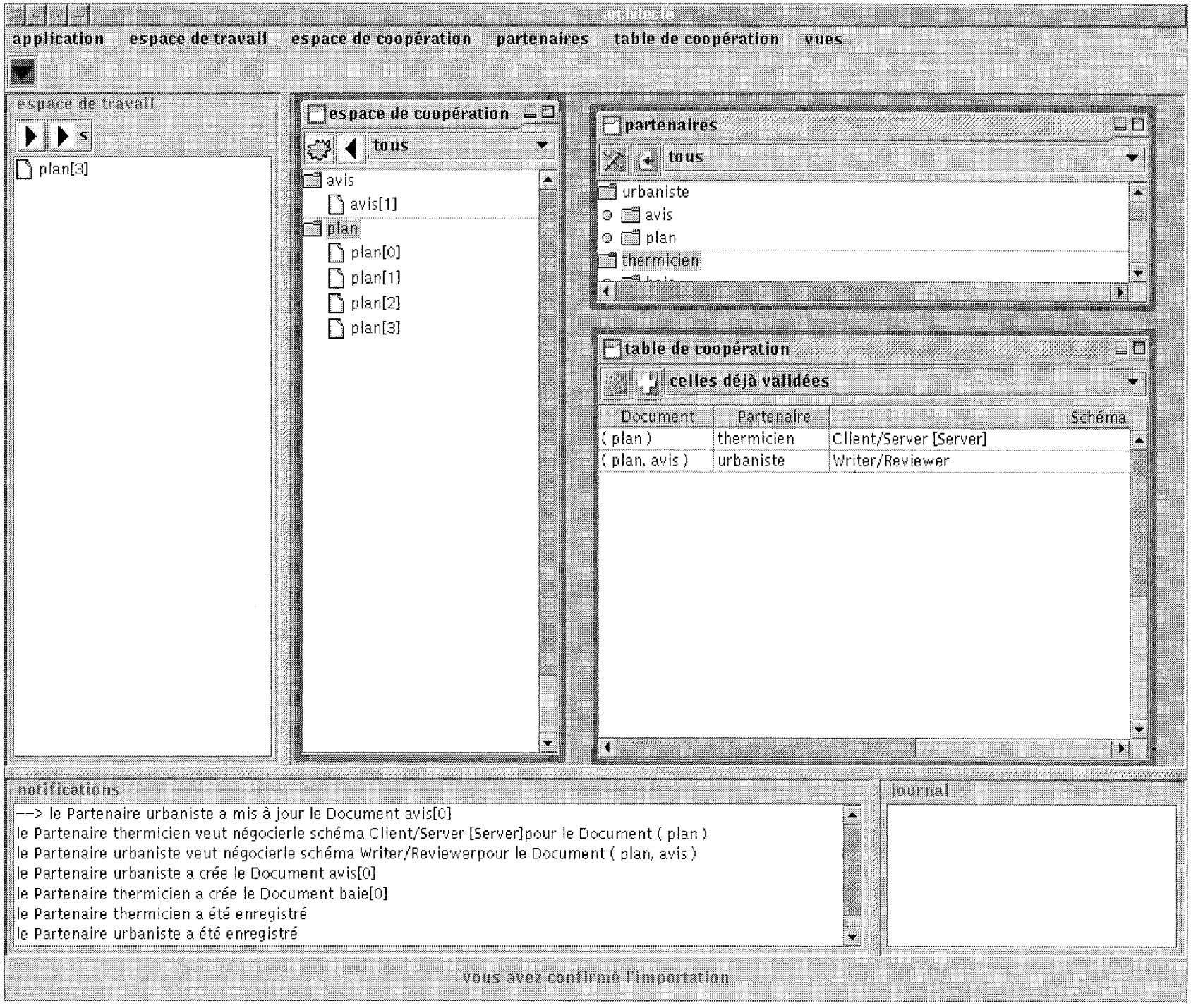
Le prototype *DisCOO* est donc complètement découplé de l'environnement d'exécution, tant du point de vue du système d'exploitation (bytecode Java) que de l'infrastructure de communication (CORBA). A titre d'exemple nous avons testé *DisCOO* sur un réseau local hétérogène composé à la fois de PC fonctionnant sous Windows NT, de stations de travail Solaris et de PC sous Linux.

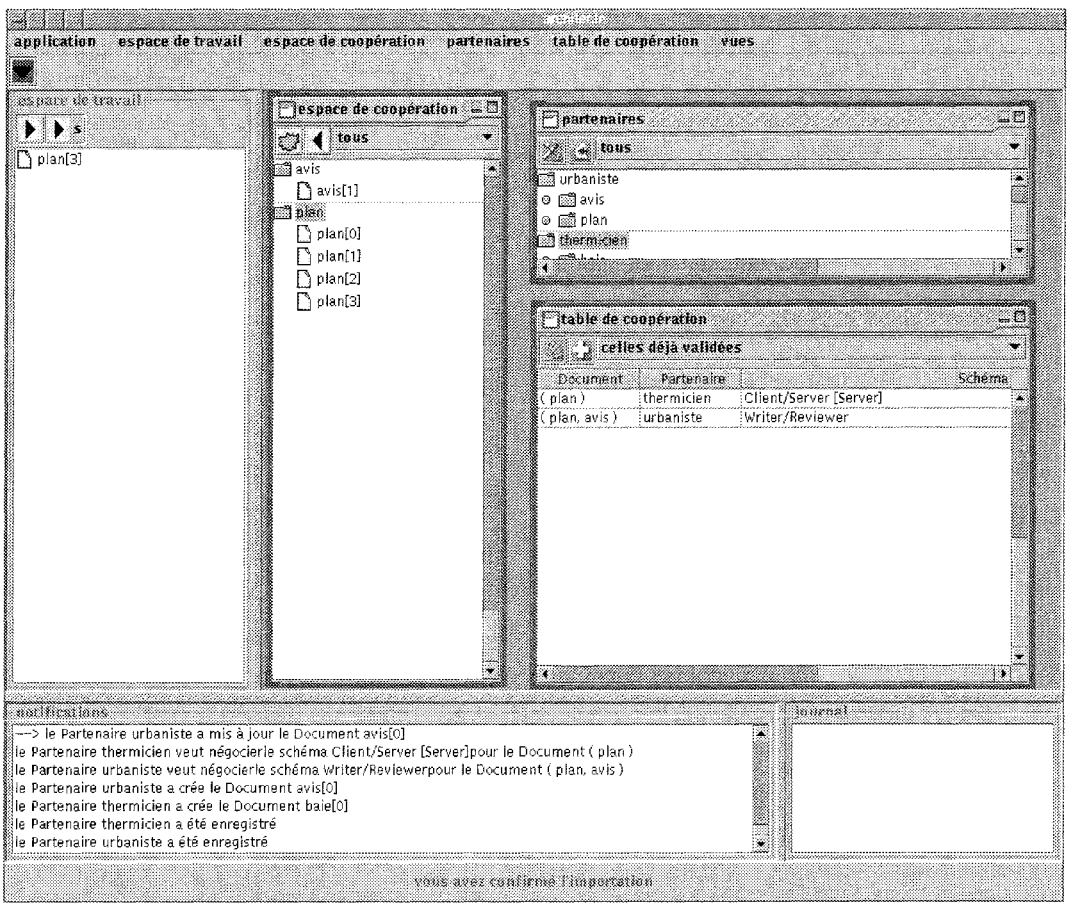
5.2.3 Exemple d'Application Basée sur *DisCOO*

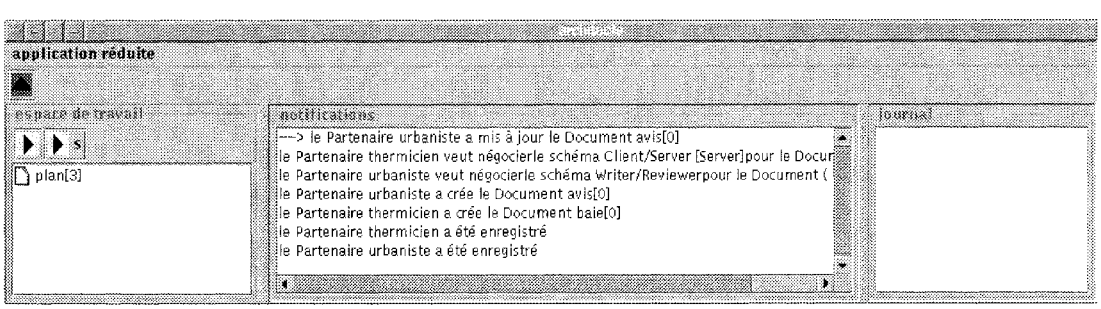
Afin d'illustrer les résultats de nos travaux (le modèle des *DisCOO*-transactions et les services de coopération proposés), nous avons développé une application permettant à plusieurs partenaires de coopérer via le partage de ressources de type fichier. Les principales fonctionnalités offertes sont les suivantes:

- créer une ressource de type fichier
- afficher la liste des partenaires
- afficher la liste des ressources d'un partenaire (son espace de coopération)
- afficher notre propre espace de coopération
- afficher notre table de coopération (contrats négociés)
- demander la négociation d'un schéma de coopération à un partenaire
- accepter/refuser une négociation demandée par un partenaire
- importer une ressource d'un partenaire
- externaliser une ressource vers notre espace de travail
- transférer une ressource de l'espace de travail vers notre espace de coopération
- afficher les ressources présentes dans notre espace de travail
- afficher le journal des événements (notre histoire locale)

Le principal objectif de cette application était de démontrer le fonctionnement de nos services de base de la coopération. Il s'agit d'un utilitaire indépendant des applicatifs existants et permettant à différents partenaires de coopérer tel que nous l'avons expliqué au


architecte
application
espace de travail
espace de coopération
partenaires
table de coopération
vues
espace de travail
plan[3]
espace de coopération
tous
avis
avis[1]
plan
plan[0]
plan[1]
plan[2]
plan[3]
partenaires
tous
urbaniste
avis
plan
thermicien
table de coopération
celles déjà validées
Document
Partenaire
Schéma
(plan)
thermicien
Client/Server [Server]
(plan, avis)
urbaniste
Writer/Reviewer
notifications
--> le Partenaire urbaniste a mis à jour le Document avis[0]
le Partenaire thermicien veut négocierle schéma Client/Server [Server]pour le Document (plan)
le Partenaire urbaniste veut négocierle schéma Writer/Reviewerpour le Document (plan, avis)
le Partenaire urbaniste a crée le Document avis[0]
le Partenaire thermicien a crée le Document baie[0]
le Partenaire thermicien a été enregistré
le Partenaire urbaniste a été enregistré
Journal
vous avez confirmé l'importation


application
espace de travail
espace de coopération
partenaires
table de coopération
vues
espace de travail
plan[3]
espace de coopération
tous
avis
avis[1]
plan
plan[0]
plan[1]
plan[2]
plan[3]
partenaires
tous
urbaniste
avis
plan
thermicien
table de coopération
celles déjà validées
Document
Partenaire
Schéma
(plan)
thermicien
Client/Server [Server]
(plan, avis)
urbaniste
Writer/Reviewer
notifications
journal
--> le Partenaire urbaniste a mis à jour le Document avis[0]
le Partenaire thermicien veut négocierle schéma Client/Server [Server]pour le Document (plan)
le Partenaire urbaniste veut négocierle schéma Writer/Reviewerpour le Document (plan, avis)
le Partenaire urbaniste a crée le Document avis[0]
le Partenaire thermicien a crée le Document baie[0]
le Partenaire thermicien a été enregistré
le Partenaire urbaniste a été enregistré
vous avez confirmé l'importation


application réduite
espace de travail
plan[3]
notifications
journal
--> le Partenaire urbaniste a mis à jour le Document avis[0]
le Partenaire thermicien veut négocierle schéma Client/Server [Server]pour le Docur
le Partenaire urbaniste veut négocierle schéma Writer/Reviewerpour le Document (
le Partenaire urbaniste a crée le Document avis[0]
le Partenaire thermicien a crée le Document baie[0]
le Partenaire thermicien a été enregistré
le Partenaire urbaniste a été enregistré

cours des chapitres précédents, à savoir: chaque partenaire dispose de son propre référentiel local pour stocker les ressources qu'il partage, les partenaires doivent négocier des schémas de coopération avant de pouvoir s'échanger des données entre leurs référentiels locaux respectifs, ...

Deux copies d'écran de cette application sont placées en encart dans le manuscrit. La première représente l'interface fournie à un utilisateur pour échanger des documents avec ses partenaires. Cette interface fait apparaître à l'utilisateur:

- son espace de coopération (son référentiel local) pour stocker les ressources qu'il accepte de partager avec ses partenaires,
- son espace de travail qui lui permet de manipuler les documents à l'aide de ses applications habituelles,
- la liste de ses partenaires et des ressources mises à disposition dans leurs espaces de coopération,
- sa table de coopération, c'est-à-dire la liste des contrats (schémas de coopération) qu'il a négociés avec ses partenaires,
- une fenêtre de notifications lui permettant d'être informé des opérations qui ont été invoquées par ses partenaires et qui le concernent,
- son journal, c'est-à-dire la liste des opérations qu'il a invoquées (son histoire locale).

Lorsque l'utilisateur désire travailler sur les documents de son espace de travail, cette interface peut être réduite pendant qu'il utilise ses applications habituelles (seconde copie d'écran). En effet, les seules informations qui sont alors utiles à l'utilisateur sont la liste des documents de son espace de travail, la fenêtre de notifications (*"group awareness"*) et son journal.

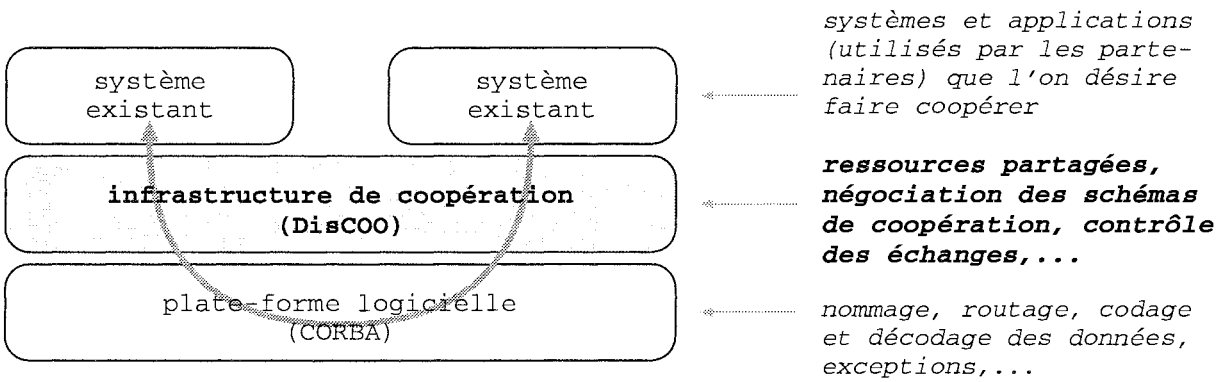
5.2.4 Bilan

Le prototype *DisCOO* offre donc différents services de base permettant le développement d'applications coopératives distribuées. Ces services prennent en compte, pour une activité donnée: la gestion des ressources que celle-ci accepte de partager (son espace de coopération), l'externalisation de ces ressources sous une forme utilisable par les applications habituelles (espace de travail de l'activité), la définition et la négociation de schémas de coopération (protocole), le contrôle des interactions (transferts entre espaces de coopération) de cette activité avec les autres activités du système (coordinateur).

Ces services de coopération constituent donc la "glue" entre les systèmes et applications existants d'une part, et l'infrastructure de communication (bus logiciel CORBA) d'autre part. Le fait de regrouper tous les mécanismes nécessaires à la coopération au niveau de cette couche logicielle (figure 5.19) nous évite ainsi de devoir les coder explicitement au sein de chaque application. Ceci diminue également le risque d'incohérence entre les contrôles réalisés par les différentes activités.

Finalement, d'un point de vue plus technique, le fait que tous les composants du prototype *DisCOO* soient programmés en Java nous permet d'avoir une seule et unique

Figure 5.19 *DisCOO* = Services de Coopération



implémentation de ces services pouvant fonctionner sur tout système d'exploitation disposant d'une machine virtuelle Java (Windows 98/NT, Solaris, Linux, ...). Le prototype *DisCOO* est donc complètement portable.

A noter également la présence de deux services supplémentaires sur la figure 5.16: un service de **notifications** et un **médiateur**. Nous ne les avons pas présentés car ils ne sont pas directement liés aux résultats de nos travaux. Il s'agit plutôt de services répondant à des besoins techniques. Les notifications sont des messages transmis entre activités pour signaler la publication d'une nouvelle version de telle ressource, la création d'un nouveau partenaire, ...Le service de notifications est donc orienté vers le *"group awareness"*. Le médiateur est quant à lui une interface de coopération générale. L'idée est de masquer aux applications les mécanismes internes mis en œuvre par nos quatre services de coopération. Il est ainsi possible de modifier certains de nos services sans perturber les applications qui utilisent déjà *DisCOO*.

Concernant l'utilisation de nos services de coopération pour le développement d'applications coopératives distribuées, nous avons fourni un exemple d'utilitaire indépendant dédié à la gestion de la coopération. Les utilisateurs se servent donc de leurs applications habituelles pour travailler puis, lorsqu'ils désirent interagir avec des partenaires, utilisent cet utilitaire pour coopérer. Une autre possibilité serait d'intégrer ces fonctionnalités à des applications existantes pour lesquelles nous disposons soit des sources (ex: emacs), soit d'une API permettant l'ajout de nouvelles fonctions dans les menus (ex: Word). Les utilisateurs pourraient ainsi avoir accès à ces mécanismes de coopération directement depuis leurs applications habituelles.

5.3 Conclusion

Nous venons de présenter dans ce chapitre la mise en œuvre des résultats établis au chapitre 4, c'est-à-dire le développement d'un prototype, *DisCOO*, conforme au modèle des *DisCOO*-transactions, à savoir:

- les objets partagés sont stockés localement sur les transactions
- le contrôle des échanges est réalisé localement sur chaque transaction

– les transactions peuvent négocier les schémas de coopération

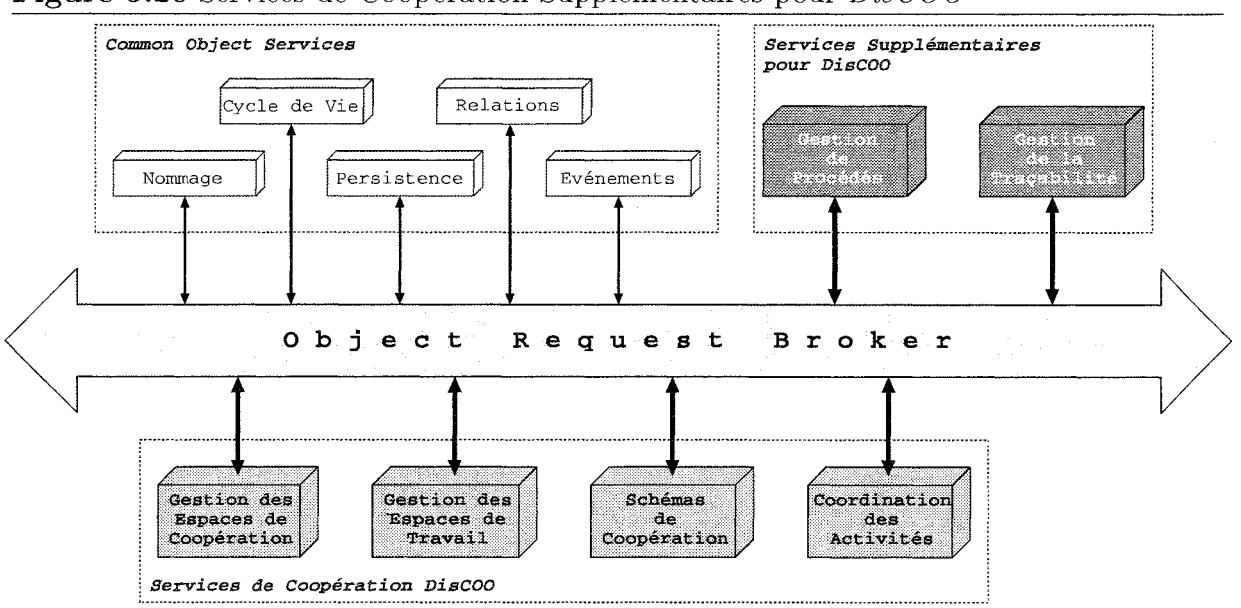
Nous nous sommes déchargés des problèmes technologiques de "bas niveau" liés à la distribution via l'utilisation d'un ORB conforme aux spécifications CORBA. Notre objectif était de ne pas développer "un système propriétaire supplémentaire". L'idée était plutôt de réaliser une infrastructure intégrant les mécanismes de base nécessaires à la coopération et à partir de laquelle il serait possible de programmer de telles applications coopératives distribuées. Nous avons donc implémenté ces mécanismes sous la forme de services CORBA qui peuvent être à leur tour utilisés par les objets de l'ORB au même titre que les services standard.

Un exemple d'application démontrant l'utilisation de nos services de coopération a ensuite été développé. Celle-ci permet à différents partenaires de coopérer tel que nous l'avons expliqué au cours des chapitres précédents.

Concernant la mise en œuvre des schémas de coopération et l'évaluation de leurs règles par rapport à l'histoire locale courante d'une activité, nous avons fait le choix d'implémenter ces règles directement sous la forme de prédicats Prolog. En procédant de cette manière, les schémas et règles de coopération (formalisés en ACTA) sont donc réellement codés sous la forme de propriétés (les prédicats Prolog) évaluées sur les histoires locales des activités. Du point de vue de la programmation cela nous permet de créer et de modifier les schémas de coopération de façon "naturelle" par rapport à leur définition en ACTA. L'ajout de nouveaux schémas de coopération au prototype *DisCOO* en est ainsi grandement facilitée.

En outre, le prototype *DisCOO* est complètement portable car développé entièrement en Java à partir d'un ORB et d'un interpréteur Prolog tous deux écrits en Java. *DisCOO* est donc totalement découplé de l'environnement d'exécution, tant du point de vue du système d'exploitation (bytecode Java) que de l'infrastructure de communication (CORBA).

Figure 5.20 Services de Coopération Supplémentaires pour *DisCOO*



Le développement du prototype *DisCOO* nous a donc permis de valider les résultats de nos travaux présentés au chapitre 4. Il s'agit toutefois d'un système ouvert, c'est-à-dire qu'il est tout à fait possible d'y intégrer de nouveaux services prenant en charge de nouvelles formes du contrôle de la coopération (cf. figure 5.20). Nous pensons en particulier à des services orientés vers la gestion des procédés exécutés par les activités (workflow distribué) ou de la traçabilité (i.e. savoir qui a modifié telle ressource, pour quelles raisons, quelles ont été les conséquences sur les autres ressources, ...). Ceci fera partie de nos perspectives décrites au chapitre 6.

Chapitre 6

Bilan et Perspectives

Comme nous l'avons indiqué en introduction, les travaux réalisés dans cette thèse visent à supporter la coopération entre les différentes activités d'une application distribuée. Cette coopération est vue en termes d'échanges d'informations via les documents partagés: c'est la coopération indirecte. Les activités ne sont pas isolées les unes des autres, ce qui signifie qu'elles peuvent s'échanger, au cours de leur exécution, des résultats intermédiaires. Afin d'éviter que de tels échanges n'introduisent des inconsistances au niveau des résultats finaux produits par les activités, il est nécessaire de contrôler ces interactions. D'un autre côté, la distribution de l'application sur un réseau tel qu'Internet nécessite également une certaine autonomie des activités, tant du point de vue de l'accès aux données partagées que du contrôle des échanges d'une activité avec ses partenaires.

La plupart des environnements actuels considèrent cependant la coopération comme étant un problème de concurrence d'accès à un référentiel commun. Par conséquent, bien que les différentes activités puissent être exécutées sur des sites géographiquement distribués, le contrôle de l'application globale reste centralisé sur le serveur (figure 6.1-a). Cette centralisation facilite la coordination et l'administration de ces activités puisque le serveur a ainsi connaissance de toutes les opérations invoquées par chacune des activités. Cela nécessite toutefois que les activités soient quasi-continuellement connectées au serveur, ce qui est en totale contradiction avec la notion d'autonomie des activités.

Cet aspect **"autonomie des activités"** a été notre fil conducteur tout au long des travaux présentés dans cette thèse, notre objectif étant d'aboutir à une solution dépourvue de tout site central dont dépendraient les activités. Nous avons ainsi décliné cette notion d'autonomie sous trois formes:

- autonomie pour l'**accès aux données**
- autonomie pour le **contrôle des échanges**
- autonomie pour le **choix des règles de coopération** à respecter

Afin de simplifier la programmation de telles applications nous avons choisi de cacher au maximum la complexité introduite par l'exécution concurrente des différentes activités en utilisant une **approche transactionnelle**. Ceci nous évite de devoir programmer explicitement toutes les interactions possibles entre les différentes activités. Le contrôle de la concurrence est ainsi réalisé par un critère de correction qui caractérise les exécutions considérées comme étant correctes.

Les travaux présentés dans cette thèse avaient donc pour objectif de définir un nouveau modèle de transactions avancé supportant à la fois la coopération entre les transactions, leur distribution et l'hétérogénéïté de leurs relations de coopération. Un certain nombre de travaux concernant le problème de la coopération avaient déjà été réalisés dans le cadre du projet *COO* et avaient donné lieu à la définition du modèle des *COO*-transactions et de la *COO*-sérialisabilité, un critère de correction supportant la coopération indirecte entre les transactions (échanges de résultats intermédiaires au cours de leur exécution). Toutefois, à la manière des systèmes transactionnels classiques, *COO* était orienté vers le contrôle des accès concurrents à un référentiel commun.

6.1 Résultats Obtenus

Les travaux réalisés dans le cadre de cette thèse ont donné lieu à deux résultats. D'un point de vue formel tout d'abord, il s'agit de la définition d'un nouveau modèle de transactions avancé, les ***DisCOO*-transactions**, ainsi que de **plusieurs critères de correction distribués**, la *D*-sérialisabilité et la *DisCOO*-sérialisabilité (avec trois schémas de coopération: "écriture coopérative", "client/serveur" et "rédacteur/relecteur"). Le second résultat est la définition d'une **architecture** nous permettant de construire de telles applications en "connectant", une fois qu'un schéma de coopération aura été négocié, les différentes activités distribuées. Cette architecture a ensuite été mise en œuvre sous forme de **services de base pour la coopération** au sein de notre prototype *DisCOO*.

6.1.1 Modèle de Transactions Coopératives Distribuées

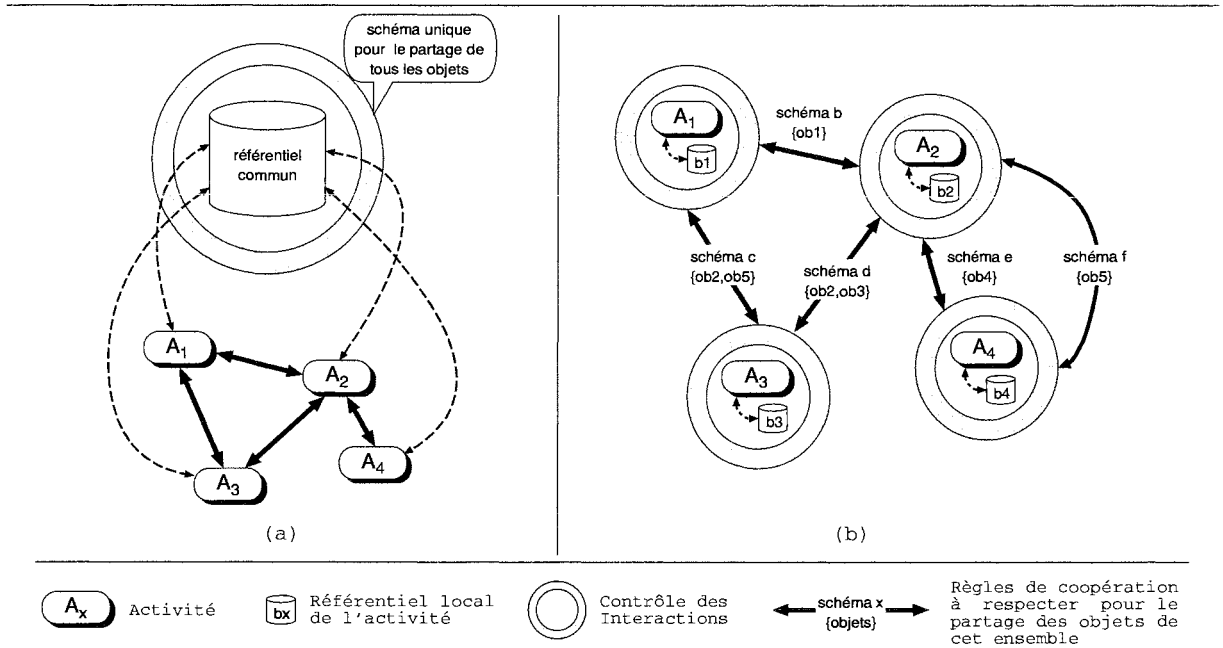
Partant des travaux réalisés dans *COO* en ce qui concerne la correction syntaxique des interactions coopératives, la première étape de notre travail consistait donc à passer d'un contrôle des accès concurrents (interactions implicites) à un contrôle des échanges de données entre transactions (interactions explicites). Nous avons pour cela défini les notions de **référentiel local** d'une transaction, d'**histoire locale** d'une transaction, et d'**opération de transfert** entre transactions. L'idée est que chaque transaction posséde sa propre copie des objets auxquels elle accède et coopère avec les autres transactions en échangeant des valeurs de leurs copies respectives. Ce sont ces opérations de transfert qui nous permettent de synchroniser les histoires locales des différentes transactions. C'était l'étape *"autonomie pour l'accès aux données"* (section 4.1).

La deuxième étape concernait la définition de nouveaux **critères de correction** qui soient eux-mêmes **distribués**. L'idée était en effet de permettre à chaque transaction du système de coordonner elle-même ses propres échanges de données avec les autres transactions. A l'inverse d'un critère de correction "classique" qui est défini sur l'histoire globale, un critère de correction dit "distribué" est destiné à être vérifié par chaque nœud du système (un nœud représentant l'exécution d'une transaction) en n'ayant accès qu'aux informations journalisées localement par ce nœud (l'histoire locale de la transaction). Nous avons ainsi défini de nouveaux critères de correction locaux (la ***D*-sérialisabilité** et la ***DisCOO*-sérialisabilité**) qui, lorsqu'ils sont assurés au niveau de chaque transaction, garantissent les mêmes propriétés au niveau du système complet que leurs homologues

"globaux" classiques (la sérialisabilité et la *COO*-sérialisabilité). C'était l'étape *"autonomie pour le contrôle des échanges"* (section 4.2).

Dans le cadre des applications coopératives distribuées, toutes les transactions ne sont pas forcément guidées par les mêmes règles pour coopérer avec leurs partenaires. Etant donné que les transactions de notre modèle n'interagissent plus via des accès concurrents à un référentiel commun mais par l'intermédiaire d'échanges de données explicites avec les autres transactions, nous avons choisi de leur permettre de **négocier** les **schémas de coopération** qu'elles utilisent, c'est-à-dire de choisir elles-même les règles de coopération qui contrôleront leurs échanges. Cette étape était ainsi dédiée à *"l'autonomie pour le choix des règles de coopération à respecter"* (section 4.3).

Figure 6.1 Architecture Centralisée vs Architecture Distribuée



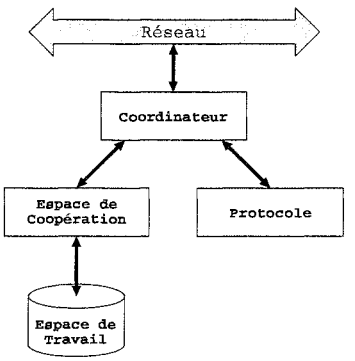
Les transactions de notre modèle sont ainsi organisées en réseau, les nœuds réprésentant les transactions et les arcs réprésentant les schémas de coopération négociés entre les transactions. Il s'agit d'une architecture d'égal-à-égal dans laquelle chaque transaction est elle-même responsbable du contrôle de ses propres interactions avec les autres transactions du système (figure 6.1-b). Comme nous pouvons le constater, il n'existe donc plus **un seul** critère de correction contrôlant tous les échanges mais **plusieurs** critères de correction distribués qui doivent cohabiter au sein du même système.

6.1.2 Services de Coopération

Concernant la mise en œuvre de ce modèle, notre objectif était de ne pas développer "un système propriétaire supplémentaire". L'idée était plutôt de réaliser une infrastructure intégrant les mécanismes de base nécessaires à la coopération et à partir de laquelle il serait possible de programmer de telles applications coopératives distribuées.

Lors de la présentation de l'architecture retenue pour notre système de coopération (section 5.1), nous avons indiqué qu'une activité était constituée de quatre composants principaux, définissant ainsi les quatre **services de coopération de base**:

- Un **espace de coopération** pour stocker les objets partagés.
- Un **espace de travail** pour manipuler ces objets à l'aide des outils existants.
- Un **protocole** pour contrôler les échanges de l'activité avec ses partenaires.
- Un **coordinateur** qui joue le rôle d'interface de l'activité.



6.2 Perspectives

6.2.1 Opérations de Négociation et de Re-Négociation

Dans la section 4.3 nous ne nous sommes en fait intéressés qu'au résultat de la négociation d'un schéma de coopération entre deux transactions, c'est-à-dire à l'événement $Contract[t_i, t_j, O, s]$ qui est journalisé dans l'histoire locale de chaque transacion. Nous avons de plus indiqué qu'en cas de conflit entre les différents schémas de coopération négociés par une même transaction, une solution pouvait être de re-négocier certains de ces schémas (révision dynamique des contrats négociés). Cela nécessite toutefois de formaliser cette opération de re-négociation, à la fois du point de vue de la correction des exécutions (contraintes à vérifier avant de re-négocier, impact d'une telle opération sur les techniques de recouvrement, ...) et du point de vue des schémas de coopération pouvant effectivement être re-négociés (accord des deux partenaires).

L'idée consiste en fait à considérer les opérations de négociation et de re-négociation comme étant des opérations classiques (au même titre que les opérations *read* et *write* par exemple) pouvant être invoquées sur des "objets de négociation" qui seraient porteurs d'informations spécifiques: liste des schémas proposés par chaque partenaire, liste des schémas convenant aux deux partenaires, proposition d'un autre schéma en cas de refus de celui demandé par le partenaire, ... En d'autres termes, nous nous intéressons à la phase de négociation d'un contrat comme partie intégrante de l'exécution des transactions.

L'objectif est en fait de permettre aux différentes transactions de coopérer, selon un schéma prédéfini, pour définir le schéma de coopération qu'elles utiliseront pour contrôler leurs échanges concernant tel ou tel objet.

6.2.2 Nouveaux Schémas de Coopération

Les schémas de coopération proposés dans la section 4.3 sont tous issus soit de la sérialisabilité (schéma *DSeria*) soit de la *COO*-sérialisabilité (schémas *DisCOO*, *client_server* et *writer_reviewer*). Ces quatre schémas ne couvrent pas toutes les relations de coopération pouvant exister entre les différentes activités d'une application coopérative distribuée.

Une perspective à nos travaux consiste à analyser les usages courants de coopération dans différents domaines tels que l'ingénierie du bâtiment, la construction automobile (coopération entre un constructeur et ses équipementiers), la télémédecine, ... De cette analyse nous pensons pouvoir en déduire de nouveaux schémas de coopération qu'il nous faudra formaliser de manière à les mettre en œuvre au sein de notre infrastructure.

Un premier travail dans ce sens a déjà été réalisé dans [Can98b] et a donné lieu à une taxonomie de schémas de coopération selon les paramètres suivants: dépendances lecture/écriture entre les transactions, vues synchrones ou asynchrones, consensus sur les valeurs finales.

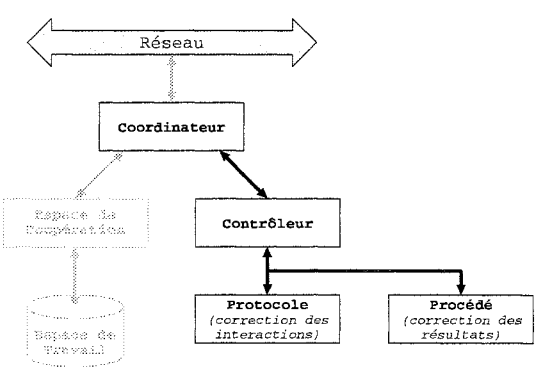
Un second aspect de ce travail est l'étude de nouveaux schémas de coopération multi-partites, c'est-a-dire la négociation d'un contrat par plus de deux activités. L'objectif d'un tel schéma de coopération serait de coordonner, explicitement, les échanges au sein d'un groupe d'activités tout en ne se basant que sur des contrôles réalisés sur les histoires locales de ces activités.

6.2.3 Intégration d'Autres Composants de Contrôle

Dans l'architecture présentée au chapitre 5, le seul contrôle réalisé par le coordinateur sur les échanges de données entre les espaces de coopération concerne la correction des interactions assurée par le protocole. Nous avons en effet fait l'hypothèse que les activités sont cohérentes, c'est-à-dire que les utilisateurs travaillent correctement. Par exemple, lorsqu'un rélecteur importe une nouvelle version du document produit par le rédacteur, il modifie sa revue en conséquence avant de l'envoyer au rédacteur.

Nous pensons toutefois que cette hypothèse est trop forte dans le cas d'applications coopératives distribuées à large échelle car il est tout à fait possible qu'un utilisateur se trompe ou oublie un détail sur un projet de grande envergure.

Outre la correction des exécutions, il nous faut donc également garantir la correction des résultats produits par les activités. Une solution serait de modifier notre architecture de manière à pouvoir intégrer plusieurs composants de contrôle. Par exemple, ce pourrait être la gestion du procédé exécuté par l'activité, le contrôle de l'enchaînement des différentes activités (workflow distribué), ...



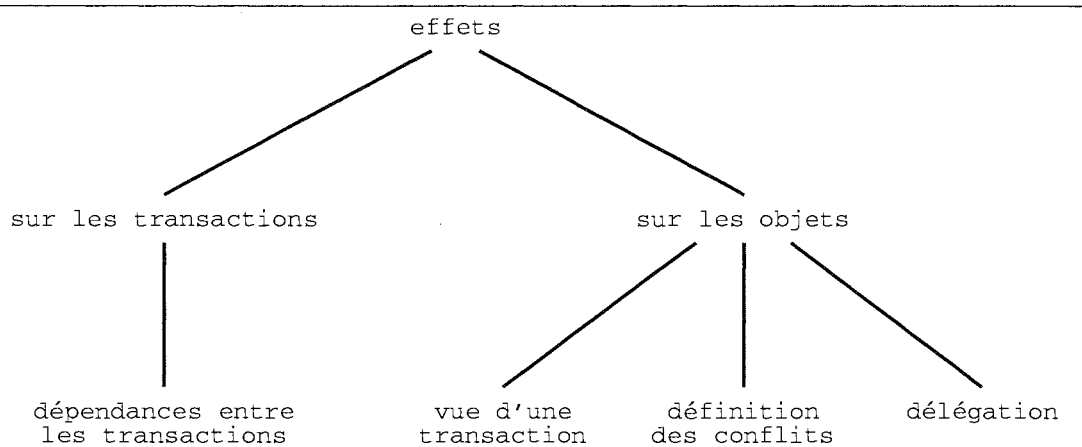
Comme nous pouvons le constater, l'architecture que nous avons proposée est suffisamment flexible pour permettre l'intégration de nouveaux composants de contrôle. Ceux-ci doivent cependant respecter la philosophie de notre architecture, à savoir qu'il doit s'agir d'un contrôle local réalisé uniquement à partir des informations journalisées par chaque activité.

Annexe A

Le Formalisme ACTA

La diversité des modèles de transactions étendus, leur relative complexité et, pour certains d'entre eux, leur manque de formalisation, rendent la caractérisation et la comparaison de ces modèles délicates. Devant cet état de fait, ACTA a été proposé comme un outil de spécification formelle permettant de caractériser le comportement et les propriétés des modèles de transactions [Chr90, Chr92, Ram93, Ram96, Chr94]. Le formalisme ACTA est basé sur la logique du premier ordre. Les règles ACTA permettent de caractériser les effets que les transactions produisent sur les autres transactions ainsi que sur les objets qu'elles manipulent.

Figure A.2 Classification des Effets d'une Transaction



Dans un sytème, les transactions produisent des effets sur des objets en déclenchant des événements objets (ex: opérations lire, écrire) et des effets sur d'autres transactions en déclenchant des événements significatifs (ex: début, validation, abandon). Les effets sur les objets mettent en jeu les concepts de visibilité, de conflits et de délégation d'effets, tandis que les effets sur les transactions mettent en jeu des dépendances entre les transaction. La figure A.2 illustre la classification de ces effets dans l'environnement ACTA.

A.1 Préliminaires

ACTA considère l'appel d'une opération sur un objet comme un événement objet. L'événement $p_t[ob]$ correspond à l'invocation de l'opération p par la transaction t sur l'objet ob. L'effet d'une opération sur un objet n'est pas rendu permanent au moment de l'exécution de l'opération. L'opération doit être explicitement validée ou annulée pour que l'effet devienne permanent.

Le formalisme ACTA repose su la définition de prédicats portant sur l'occurence d'événements objets et d'événements significatifs dans l'historique d'exécution (noté H) constitué de tous les événements produits par l'ensemble des transactions du système et indiquant l'ordre partiel dans lequel ces événements ont été produits[44]. H_{ct} représente l'histoire courante, c'est à dire l'histoire des événements à un moment donné.

Un modèle transactionnel est défini en ACTA à l'aide de prédicats et de règles ACTA caractérisant l'ensemble des histoires que peut produire ce modèle. Cela peut se représenter comme des invariants définis sur les histoires ou comme des pré et post conditions sur les événements objets et les événements significatifs. Les propriétés de correction apportées par les différents modèles sont alors exprimées en terme de propriétés sur les histoires.

A.2 Dépendances Inter-Transactions

A la base les dépendances sont des contraintes sur les histoires, produites par l'exécution concurrente de transactions interdépendantes. Par exemple une dépendance de terminaison entre deux transactions peut s'exprimer de la manière suivante: $t_j\ \mathcal{CD}\ t_i$ et signifie que si t_i et t_j terminent alors la terminaison de t_i précède celle de t_j, i.e.:

$$Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H(Commit_{t_i} \rightarrow Commit_{t_j})$$

De même un dépendance d'annulation peut s'exprimer de la manière suivante: $(t_j\ \mathcal{AD}\ t_i)$ signifie que si t_j est annulée alors t_j doit également être annulée.

$$Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H$$

Les dépendances entre transactions peuvent être structurelles ou développées dynamiquement au cours de l'exécution:

Dépendance de structure Par exemple, dans les transactions emboîtées, la création d'une sous-transaction crée une dépendance de terminaison entre le père et le fils.

$$Spawn_{t_p}[t_c] \in H \Rightarrow (t_c\ \mathcal{WD}\ t_p) \wedge (t_p\ \mathcal{CD}\ t_c)$$

Dépendances de comportement Les dépendances peuvent aussi être produite à l'exécution par des interactions entre transactions sur des objets partagés. Par exemple:

$$(p_{t_i}[ob] \rightarrow q_{t_j}[ob] \Rightarrow (t_j\ \mathcal{D}\ t_i)$$

indique que si t_i invoque l'opération p sur ob et qu'ensuite t_j invoque l'opération q sur le même objet, alors t_j développe une dépendance de type $\mathcal{D}$ vers t_i.

44. Le prédicat $e \rightarrow e'$ est vrai si l'événement e précède l'événement e' dans l'histoire H. Il est faux dans le cas contraire.

A.3 Conflits entre Opérations

Une histoire $H^{(ob)}$ d'opérations sur un même objet *ob* est représentée comme la composition fonctionnelle des différentes opérations ordonnées dans le temps: $H^{(ob)} = p_1 \circ p_2 \circ \ldots \circ p_n$ avec $p_i \rightarrow p_{i+1}$. Chacune de ces opérations retourne une valeur et produit un état. Si s est l'état d'un objet, $return(s, p)$ renvoie le résultat produit par l'opération p sur cet objet. $state(s, p)$ retourne l'état de cet objet produit après l'exécution de l'opération p. L'état s d'un objet *ob* produit par une séquence d'opérations à partir d'un état initial s_0 est donc égal à $state(s_0, H^{(ob)})$.

DÉFINITION A.1
Deux opérations p et q sont dites *conflictuelles* pour un état $H^{(ob)}$ (ce qui sera noté $conflict(H^{(ob)}, p, q)$ ou plus simplement $conflict(p[ob], q[ob])$) ssi

$$\begin{aligned}(state(H^{(ob)} \circ p,\ q) &\neq state(H^{(ob)} \circ q,\ p)) \vee \\ (return(H^{(ob)},\ q) &\neq return(H^{(ob)} \circ p,\ q)) \vee \\ (return(H^{(ob)},\ p) &\neq return(H^{(ob)} \circ q,\ p))\end{aligned}$$

Deux opérations qui ne sont pas en conflit sont dites *compatibles*.

En cas de conflit entre deux opérations, c'est-à-dire si le prédicat $conflict(H^{(ob)}, p, q)$ vaut vrai, $return\_value\_independent(H^{(ob)}, p, q)$ est vrai si la valeur de retour de q est independante du fait que p précède q ou non, i.e., $return(H^{(ob)} \circ p,\ q) = return(H^{(ob)},\ q)$; sinon q est *"return-value dependent"* de p (noté $return\_value\_dependent(H^{(ob)}, p, q)$).

A.4 Visibilité et Ensemble des Conflits

La visibilité d'une transaction caractérise l'ensemble des effets sur les objets qui sont visibles pour cette transaction. Elle exprime l'accessibilité de la transaction aux mises à jour courantes subies par les objets.

La vue d'une transaction, notée $View_t$, spécifie les objets et l'état de ces objets visibles par une transaction t à un instant donné.

$View_t$ est en fait une projection de l'histoire courante H_{ct} selon un prédicat. Plus formellement:

$$View_t = Projection(H_{ct}, \mathcal{P}redicate)$$

L'ensemble des conflits d'une transaction t, noté $ConflictSet_t$, contient les opérations de l'histoire courante avec lesquelles les opérations invoquées par t peuvent entrer en conflit.

Cet ensemble est en fait un sous-ensemble des événements présents dans H_{ct} satisfaisant un prédicat:

$$ConflictSet_t = \{p_{t_i}[ob] \mid \mathcal{P}redicate\}$$

A.5 Conclusion

Un modèle transactionnel formalisé en ACTA est donc composé des éléments suivants:

- La liste des événements significatifs permettant de créer, de terminer ou de valider une transaction,
- La définition de la vue d'une transaction: $View_t$,
- La spécification de l'ensemble des conflits d'une transaction: $ConflictSet_t$,
- La liste des prédicats et de règles ACTA (les axiomes) caractérisant les histoires correctes.

De nombreux modèles transactionnels ont déjà été spécifiés selon ce formalisme: les transactions emboîtées, les "split&join transactions", les transactions enchaînées, les sagas, ...Nous avons choisi d'utiliser ACTA pour exprimer notre modèle avec rigueur mais également pour permettre de le comparer plus facilement avec les autres modèles.

Annexe B

Définition Axiomatique des Transactions Coopératives Distribuées

B.1 Préliminaires

Lorsqu'une transaction t exécutera une opération op sur un objet ob (ce qui est noté $op_t[ob]$ dans le formalisme ACTA, cf. section A.1), il sera nécessaire de préciser sur quelle instance de cet objet ob sera effectuée cette opération op. Nous utiliserons la notation $op_t[ob_{t'}]$ pour représenter le fait que l'opération $op_t[ob]$ est invoquée sur l'objet ob de la transaction t'. Ce que nous appelons la **base locale** d'une transaction t est l'ensemble des objets ob_t.

B.2 Conflits entre Opérations

DÉFINITION B.2 (OPÉRATION DE TRANSFERT)
Soit t_l une transaction.
Soient $ob_{t_{k_i}}$ et $ob_{t_{k_j}}$ deux instances d'un même objet logique ob.
Soit $Read^{(ob)}$ l'ensemble[a] des opérations q telles que $State(H^{(ob)} \circ q) = State(H^{(ob)})$.
Soit $Write^{(ob)}$ l'ensemble[b] des opérations p telles que $State(H^{(ob)} \circ p) \neq State(H^{(ob)})$.

Une opération de transfert est alors définie comme étant un couple d'opérations, i.e. $transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}] = (q_{t_l}[ob_{t_{k_i}}], p_{t_l}[ob_{t_{k_i}}])$, telles que $(q \in Read^{(ob)}) \wedge (p \in Write^{(ob)}) \wedge (q_{t_l}[ob] \rightarrow p_{t_l}[ob])$.

[a] $Read^{(ob)}$ représente l'ensemble des opérations permettant d'accéder, sans la modifier, à la valeur de l'objet ob.

[b] $Write^{(ob)}$ représente l'ensemble des opérations permettant de modifier la valeur de l'objet ob.

Nous pouvons remarquer que pour tout objet ob et pour toutes opérations $q \in Read^{(ob)}$ et $p \in Write^{(ob)}$, nous avons $return(H^{(ob)}, q) \neq return(H^{(ob)} \circ p, q)$, et par conséquent $conflict(q_{t_l}[ob], p_{t_l}[ob])$.

Une opération de transfert (définition B.2) est une **opération virtuelle** qui est en fait

la succession de deux opérations invoquées par une même transaction t_l, l'une pour accéder à la valeur de l'objet $ob_{t_{k_i}}$ (ex: $read_{t_l}[ob_{t_{k_i}}]$), l'autre pour modifier la valeur de l'objet $ob_{t_{k_j}}$ (ex: $write_{t_l}[ob_{t_{k_i}}]$), de manière à transférer la valeur de l'instance $ob_{t_{k_i}}$ de l'objet ob vers une autre de ses instances $ob_{t_{k_j}}$. Une telle opération sera notée $transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}]$.

DÉFINITION B.3 (RELATION DE DÉPENDANCE SÉMANTIQUE $\xrightarrow{dep}$)
La relation de dépendance sémantique $\xrightarrow{dep}$ entre les différentes instances d'un même objet logique est définie de la manière suivante:

- **Réflexivité:** $\forall ob\ \forall t_k \quad (ob_{t_k} \xrightarrow{dep} ob_{t_k})$
- **Opération de transfert:** $\forall ob\ \forall t_l, t_{k_i}, t_{k_j} \quad transfer_{t_l}[ob_{t_{k_i}}, ob_{t_{k_j}}] \Rightarrow (ob_{t_{k_i}} \xrightarrow{dep} ob_{t_{k_j}})$
- **Transitivité:** $\forall ob\ \forall t_{k_1}, t_{k_2}, t_{k_3} \quad (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_2}}) \wedge (ob_{t_{k_2}} \xrightarrow{dep} ob_{t_{k_3}}) \Rightarrow (ob_{t_{k_1}} \xrightarrow{dep} ob_{t_{k_3}})$

La relation de dépendance sémantique $\xrightarrow{dep}$ entre objets (définition B.3) nous permet de savoir, pour un objet ob donné, que la valeur de telle instance de cet objet a été produite à partir de la valeur de telle autre instance.

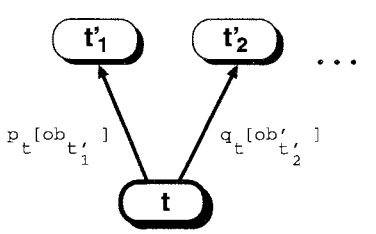
DÉFINITION B.4 (CONFLIT ENTRE OPÉRATIONS)
Soient t_i et t_j deux transactions.
Soient p et q deux opérations.
Soient $ob_{t_{k_i}}$ et $ob_{t_{k_j}}$ deux instances d'un même objet logique ob.

$$conflict(p_{t_i}[ob_{t_{k_i}}], q_{t_j}[ob_{t_{k_j}}]) \Leftrightarrow conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (ob_{t_{k_i}} \xrightarrow{dep} ob_{t_{k_j}})$$

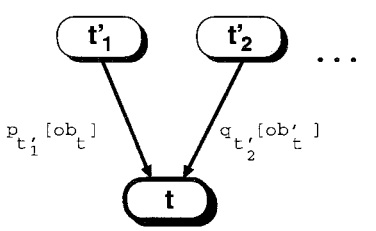
B.3 Visibilité et Ensemble des Conflits

Les opérations dont les effets sont visibles par une transaction t sont:

- les opérations invoquées par la transaction t elle-même, quels que soient les objets sur lesquels ces opérations ont été invoquées: $\{p_t[ob_{t'}] \in H_{ct}\}$



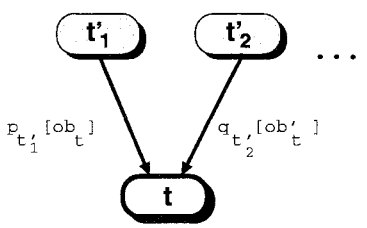
- les opérations invoquées par des transactions t' (différentes de t) sur des objets se trouvant dans la base locale de la transaction t: $\{p_{t'}[ob_t] \in H_{ct}\}$



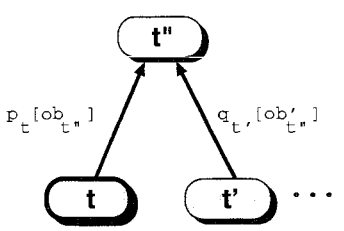
Par conséquent, la vue d'une transaction t de notre modèle est définie de la manière suivante: $View_t = \{p_t[ob_{t'}] \in H_{ct}\} \cup \{p_{t'}[ob_t] \in H_{ct}\}$. Ceci determine quelles sont, parmi toutes les opérations de l'histoire globale, celles dont la transaction t a connaissance, et donc ainsi son **histoire locale** que nous noterons $H_{ct/t}$.

En ce qui concerne notre modèle de transactions, les opérations pouvant être en conflit avec des opérations invoquées par une transaction t sont définies ci-dessous. Le prédicat $Inprogress(p)$ représente le fait que l'opération p est en cours d'exécution, i.e. qu'elle n'a pas encore été ni validée ni annulée.

- les opérations effectuées par des transactions t' $(t' \neq t)$ sur des objets ob_t de la transaction t: $\{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\}$



- les opérations exécutées par des transactions t' $(t' \neq t)$ sur des objets d'une tierce transaction t'' $(t'' \neq t)$ sur lesquels la transaction t a précédemment invoqué des opérations:



$\{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\ q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\}$

Par conséquent, pour une transaction t de notre modèle, l'ensemble des opérations pouvant être en conflit avec une des opérations de t est défini de la manière suivante:

$$\begin{aligned} ConflictSet_t = \quad & \{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\} \\ \cup\ & \{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \wedge (\exists q\ q_t[ob_{t''}] \in H_{ct}) \wedge Inprogress(p_{t'}[ob_{t''}])\} \end{aligned}$$

B.4 Schémas de Coopération

La négociation d'un schéma de coopération doit respecter certaines règles et induit elle-même certains contrôles sur les échanges de données utltérieurs (définition B.5). Tout d'abord, le contrat doit être accepté par les deux transactions, i.e. l'événement *Contract* doit apparaître dans les histoires locales des deux parties (axiome 1). Ensuite, il est nécessaire de garantir que deux transactions ont négocié un schéma de coopération avant toute invocation d'une opération d'échange de données et que les règles de coopération définies par ce schéma sont bien vérifiées (axiome 2). L'axiome 3 limite la négociation à un seul schéma de coopération par objet et par couple de transactions. Cela signifie qu'il ne sera pas possible, dans un premier temps, de renégocier un schéma de coopération. Finalement, une transaction ne pourra être terminée (`Commit`, `Abort`, ...) que si toutes les règles de coopération de tous les schémas qu'elle aura négociés le lui permettent (axiome 4).

DÉFINITION B.5 (AXIOMES DE RESPECT DES SCHÉMAS NÉGOCIÉS)

1. Un contrat est ratifié simultanément par les deux partenaires, c'est-à-dire que l'événement *Contract* est journalisé en même temps par les deux transactions:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)\quad \forall O\quad \forall s$$
$$(Contract[t_i, t_j, O, s] \in H_{t_1}) \Rightarrow (Contract[t_i, t_j, O, s] \in H_{t_2})$$

2. Une interaction entre deux transactions ne peut avoir lieu que si ces deux transactions ont préalablement négocié un schéma de coopération et si les règles de ce schéma sont vérifiées:

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad (p_{t_i}[ob_{t_j}] \in H_t) \Rightarrow$$
$$\exists t_1 \in \{t_i, t_j\}\quad \exists t_2 \in \{t_i, t_j\} - t_1\quad \exists O\ (ob \in O)\quad \exists s$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} p_{t_i}[ob_{t_j}]) \wedge s(t_1, t_2, O)$$

3. Le partage d'un objet donné est contrôlé par un et un seul schéma de coopération (i.e. pas de renégociation possible pour le moment):

$$\forall t_i, t_j\ (t_i \neq t_j)\quad \forall t \in \{t_i, t_j\}\quad \forall ob\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)$$
$$(Contract[t_1, t_2, O, s] \in H_t) \wedge (ob \in O) \Rightarrow \nexists s'$$
$$(Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_1, t_2, O', s'] \wedge (ob \in O'))$$
$$\vee\ (Contract[t_1, t_2, O, s] \rightarrow_{H_t} Contract[t_2, t_1, O', s'] \wedge (ob \in O'))$$

4. Une transaction ne peut être terminée (`Commit`, `Abort`, ...) que si tous les schémas de coopération qu'elle a négociés avec d'autres transactions sont vérifiés:

$$\forall \alpha \in TE_{t_i}\quad (\alpha \in H_{t_i}) \Rightarrow \forall t_j\quad \forall ob\quad \forall s\quad \forall t \in \{t_i, t_j\}$$
$$(Contract[t_i, t_j, O, s] \in H_t) \wedge (ob \in O) \Rightarrow s(t_i, t_j, O)$$
$$\vee\ (Contract[t_j, t_i, O, s] \in H_t) \wedge (ob \in O) \Rightarrow s(t_j, t_i, O)$$

B.4.1 *DisCOO* (Ecriture Coopérative)

DÉFINITION B.6 (SÉQUENCE D'OCCURENCES D'UNE OPÉRATION)
Une même opération $p_{t_i}[ob_{t_j}]$ peut être invoquée plusieurs fois au cours de l'exécution. Bien que notées de manière identique dans l'histoire, il s'agit d'occurences différentes. L'ensemble des occurences de cette opération $p_{t_i}[ob_{t_j}]$ (ordonné selon la relation de précédence $\rightarrow$) est appelé **séquence** de l'opération $p_{t_i}[ob_{t_j}]$. Cette séquence sera également notée $P_{t_i}[ob_{t_j}]$.

La fonction *Sequence* nous renvoie donc l'ensemble des occurences d'une opération donnée à partir de l'une de ses occurences, i.e.

$$Sequence(occ) = \{p_{t_i}[ob_{t_j}] \in H \mid occ \equiv p_{t_i}[ob_{t_j}]\}$$

Une transaction ne peut être validée (`Commit`) que si *"elle est à jour"*, i.e. que les résultats lus sont bien les derniers résultats en date produits par leurs transactions respectives, et que ces résultats sont "stabilisés" (i.e. que les opérations qui les ont produits ont été validées). En d'autres termes, si une opération q est dépendante d'une opération

DÉFINITION B.7 (DERNIÈRE OCCURENCE D'UNE OPÉRATION)
La fonction *LastOcc* nous renvoie la dernière occurence d'une séquence d'opérations, i.e.

$$LastOcc(S) = occ \in S \quad \text{telle que} \quad \nexists\, occ' \in S\ (occ' \neq occ) \quad occ \rightarrow occ'$$

p (i.e. $return\_value\_dep(p, q)$), alors la dernière occurence de q doit dépendre de la dernière occurence de p. Ceci sera représenté par le prédicat *up_to_date* défini ci-dessous. Il représente le fait que la transaction t_j est "à jour" par rapport à la transaction t_k en ce qui concerne les objets de l'ensemble O. Cette définition utilise les fonctions *LastOcc* et *Sequence* (définitions B.7 et B.6) qui nous fournissent respectivement la dernière occurence d'une opération dans une séquence d'opérations donnée et la séquence d'opérations à laquelle appartient l'occurence d'une opération donnée.

$$\begin{aligned} &up\_to\_date(t_j, t_k, O) \equiv \forall ob \in O \quad \forall Q_{t_j}[ob_{t_k}] \\ &\quad (\ \exists t_i\ \exists p\ rvd(p_{t_i}[ob_{t_k}], LastOcc(Q_{t_j}[ob_{t_k}]))\ \wedge\ (p_{t_i}[ob_{t_k}] \rightarrow LastOcc(Q_{t_j}[ob_{t_k}]))\ ) \Rightarrow \\ &\quad (\ \nexists p' \in Sequence(p_{t_i}[ob_{t_k}])\quad (LastOcc(Q_{t_j}[ob_{t_k}]) \rightarrow p'_{t_i}[ob_{t_k}])\ ) \end{aligned}$$

$$DisCOO(t_i, t_j, O) \equiv \begin{cases} \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1) \\ \quad (\texttt{Commit}_{t_1} \in H_{t_2})\ \Rightarrow\ up\_to\_date(t_1, t_2, O) \\ \\ \forall ob \in O\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)\quad \forall t_k\ (t_k \neq t_1) \\ \quad (\texttt{Commit}_{t_1}[q_{t_1}[ob_{t_2}]] \in H_{t_2})\ \Rightarrow \\ \quad\quad rvd(p_{t_k}[ob_{t_2}], q_{t_1}[ob_{t_2}])\ \wedge\ (p_{t_k}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_1}[ob_{t_2}])\ \Rightarrow \\ \quad\quad (\texttt{Commit}_{t_k}[p_{t_k}[ob_{t_2}]] \in H_{t_2}) \\ \\ \forall ob \in O\quad \forall t_1, t_2\ (t_1 \in \{t_i, t_j\},\ t_2 \in \{t_i, t_j\} - t_1)\quad \forall t_k\ (t_k \neq t_1) \\ \quad (Abort_{t_1}[p_{t_1}[ob_{t_2}]] \in H_{t_2})\ \Rightarrow \\ \quad\quad rvd(p_{t_1}[ob_{t_2}], q_{t_k}[ob_{t_2}])\ \wedge\ (p_{t_1}[ob_{t_2}] \rightarrow_{H_{t_2}} q_{t_k}[ob_{t_2}]) \Rightarrow \\ \quad\quad (Abort_{t_k}[q_{t_k}[ob_{t_2}]] \in H_{t_2}) \end{cases}$$

Le schéma de coopération *DisCOO* (écriture coopérative) est représenté par le prédicat du même nom qui est en fait la conjonction de trois règles de coopération dont voici la signification:

1. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Si la transaction t_1 désire être validée (`Commit`), elle doit être à jour par rapport à la transaction t_2. Cela signifie que les résultats de t_2 dont la transaction t_1 a eu connaissance sont bien les derniers résultats en date au niveau de la transaction t_2.
2. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Une opération de la transaction t_1 ne peut être validée (`Commit`) qu'à condition que toutes les opérations invoquées par des transactions t_k $(t_k \neq t_1)$ sur des objets $ob \in O$ de la transaction t_2 et dont elle est dépendante soient elles-mêmes validées.

3. Soient t_1 et t_2 deux transactions distinctes de $\{t_i, t_j\}$. Si une opération de la transaction t_1 est annulée (`Abort`), alors toutes les opérations, au niveau de la transaction t_2, qui en dépendent doivent également être annulées.

B.4.2 Client/Serveur

Dans le cas du schéma client/serveur, les échanges de données ne peuvent donc avoir lieu que du serveur vers le client. Supposons que deux transactions t_s (le serveur) et t_c (le client) décident d'utiliser ce schéma pour se partager les objets de l'ensemble O. La transaction cliente t_c ne sera pas autorisée à effectuer d'opérations qui modifieraient la valeur d'un objet $ob \in O$ possédé par la transaction serveur t_s (deuxième règle du prédicat *client_server*). De son côté, la transaction serveur t_s ne pourra lire la valeur d'un objet $ob \in O$ possédé par la transaction cliente t_c (troisième règle du prédicat *client_server*). Les échanges de données concernant les objets de l'ensemble O sont ainsi orientés du serveur (transactions t_s) vers le client (transactions t_c).

$$client\_server(t_c, t_s, O) \equiv \begin{cases} DisCOO(t_c, t_s, O) \\ \forall ob \in O \quad \nexists\, p \in Write^{(ob)} \quad \text{tel que} \quad p_{t_c}[ob_{t_s}] \in H \\ \forall ob \in O \quad \nexists\, q \in Read^{(ob)} \quad \text{tel que} \quad q_{t_s}[ob_{t_c}] \in H \end{cases}$$

B.4.3 Rédacteur/Relecteur

Un point important du schéma rédacteur/relecteur est que la dernière valeur de chaque revue doit être produite à partir de la dernière valeur des documents à relire et, qu'à son tour, la dernière valeur de chaque document doit être produite à partir de la dernière valeur des revues. Ceci est assuré par le prédicat *DisCOO* des deux prédicats *client_server*. En effet, étant donné que nous avons introduit un cycle de dépendances entre ces deux transactions, elles devront chacune être *up_to_date* l'une par rapport à l'autre avant de pouvoir être validées. Cela nous assure que le relecteur (la transaction t_{rev}) aura bien relu la dernière version de chaque document (objets $doc \in Doc$) et que le rédacteur (la transaction t_{wri}) aura bien eu connaissance de la dernière version de chaque revue (objets $rev \in Rev$).

Un second aspect du schéma rédacteur/relecteur est qu'il induit une inter-dépendance entre les documents et les revues. Cela signifie qu'il sera nécessaire de vérifier qu'une revue du relecteur qui sera lue par le rédacteur aura bien été produite "à partir" de documents fournis par le rédacteur, et non d'une version modifiée d'un de ces documents. En effet, le schéma client/serveur utilisé entre le rédacteur et le relecteur pour partager les documents n'impose des contraintes que sur les échanges entre ces deux activités et non sur la manière dont elles doivent utiliser les données partagées. En d'autres termes, le relecteur sera tout à fait libre de modifier (dans sa base locale) un des documents qu'il aura importé du rédacteur, mais ne pourra en aucun cas "propager" ces modifications au rédacteur.

$$writer\_reviewer(t_{wri}, t_{rev}, Doc \cup Rev) \equiv$$

$$\begin{cases} client\_server(t_{rev}, t_{wri}, Doc) \\ \\ client\_server(t_{wri}, t_{rev}, Rev) \\ \\ \forall\, rev \in Rev \quad (op \in Read_{t_{wri}}[rev_{t_{rev}}] \cup Write_{t_{rev}}[rev_{t_{wri}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\, doc \in Doc \quad \nexists\, p \in Write^{(doc)} \\ \qquad (LastOcc(Read_{t_{rev}}[doc_{t_{wri}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{wri}}[doc_{t_{rev}}]) \rightarrow p_t[doc_{t_{rev}}] \rightarrow op) \\ \\ \forall\, doc \in Doc \quad (op \in Read_{t_{rev}}[doc_{t_{wri}}] \cup Write_{t_{wri}}[doc_{t_{rev}}]) \wedge (op \in H) \Rightarrow \\ \quad \forall\, rev \in Rev \quad \nexists\, p \in Write^{(rev)} \\ \qquad (LastOcc(Read_{t_{wri}}[rev_{t_{rev}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \quad \wedge \\ \qquad (LastOcc(Write_{t_{rev}}[rev_{t_{wri}}]) \rightarrow p_t[rev_{t_{wri}}] \rightarrow op) \end{cases}$$

La troisième règle de coopération du prédicat *writer_reviewer* signifie que si l'on désire "propager" la valeur d'un objet $rev \in Rev$ du relecteur vers le rédacteur, pour chaque objet $doc \in Doc$, la "version courante" de l'objet *doc* au niveau du relecteur doit provenir du rédacteur, i.e. soit le relecteur a lu cette version chez le rédacteur (opération $read_{t_{rev}}[doc_{t_{wri}}]$), soit le rédacteur a écrit cette version chez le relecteur (opération $write_{t_{wri}}[doc_{t_{rev}}]$), et que l'objet $doc_{t_{rev}}$ n'a pas été modifié depuis. S'il y a eu une telle modification, cela signifie que la version de la revue a été produite à partir d'une version d'un document dont le rédacteur n'a pas connaissance. Par conséquent, il serait incohérent de lui envoyer cette revue.

Bien entendu, le problème est symétrique et se pose également au moment de propager, du rédacteur vers le relecteur, la valeur d'un document produit par le rédacteur et basé sur les revues précédemment rédigées par le relecteur. Ceci est représenté par la quatrième règle de coopération du prédicat *writer_reviewer*.

B.5 Définition Axiomatique de notre Modèle

Axiomes Fondamentaux des Transactions

Les invocations d'événements significatifs sont soumises à certaines règles, notamment en ce qui concerne les événements d'initialisation et de terminaison. Il s'agit des **axiomes fondamentaux des transactions** de la définition B.8. L'axiome I interdit qu'une transaction puisse être démarrée par deux événements différents. L'axiome II établit qu'une transaction terminée doit avoir été demarrée. L'axiome III interdit qu'une transaction puisse être terminée par deux événements de terminaison différents. L'axiome IV signifie que seules les transactions en cours d'exécution peuvent invoquer des opérations sur des objets. Finalement, l'axiome V représente le fait que les invocations d'opérations ne peuvent être effectuées que sur des objets de transactions ayant été initialisées. Cet axiome autorise cependant l'invocation d'opérations sur les objets d'une transaction terminée.

DÉFINITION B.8 (AXIOMES FONDAMENTAUX DES TRANSACTIONS)
Soit t une transaction. Soit H^t la projection de l'histoire H par rapport à t. Soient α, β, γ et δ des événements significatifs.

$p_t[ob_{t'}]$ représente l'événement correspondant à l'invocation effectuée par la transaction t de l'opération p sur l'objet ob de la transaction t'.

$$
\begin{array}{ll}
(I) & \forall \alpha \in IE_t \ (\alpha \in H^t) \ \Rightarrow \ \nexists \beta \in IE_t \ (\alpha \rightarrow \beta) \\
(II) & \forall \delta \in TE_t \ \exists \alpha \in IE_t \ (\delta \in H^t) \ \Rightarrow \ (\alpha \rightarrow \delta) \\
(III) & \forall \gamma \in TE_t \ (\gamma \in H^t) \ \Rightarrow \ \nexists \delta \in TE_t \ (\gamma \rightarrow \delta) \\
(IV) & \forall ob_{t'} \ \forall p \ (p_t[ob_{t'}] \in H) \ \Rightarrow \ ((\exists \alpha \in IE_t \ (\alpha \rightarrow p_t[ob_{t'}])) \wedge (\exists \gamma \in TE_t \ (p_t[ob_{t'}] \rightarrow \gamma))) \\
(V) & \forall ob_{t'} \ \forall p \ (p_t[ob_{t'}] \in H) \ \Rightarrow \ (\exists \alpha \in IE_{t'} \ (\alpha \rightarrow p_t[ob_{t'}]))
\end{array}
$$

Modèle des Transactions Coopératives Distribuées

DÉFINITION B.9 (TRANSACTIONS DISTRIBUÉES)
Soit t une transaction.

(B.1) $SE_t = \{\texttt{Begin}, \texttt{Commit}, \texttt{Abort}\}$

(B.2) $IE_t = \{\texttt{Begin}\}$

(B.3) $TE_t = \{\texttt{Commit}, \texttt{Abort}\}$

(B.4) t satisfait les axiomes fondamentaux I à V

(B.5) $View_t = \{p_t[ob_{t'}] \in H_{ct}\} \ \cup \ \{p_{t'}[ob_t] \in H_{ct}\}$

(B.6) $ConflictSet_t = \{p_{t'}[ob_t] \in H_{ct} \mid Inprogress(p_{t'}[ob_t])\}$
$\cup \ \{p_{t'}[ob_{t''}] \in H_{ct} \mid (t' \neq t) \ \wedge \ (\exists q \ q_t[ob_{t''}] \in H_{ct}) \ \wedge \ Inprogress(p_{t'}[ob_{t''}])\}$

(B.7) $\exists ob_{t_k} \ \exists p \ (\texttt{Commit}_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \Rightarrow (\texttt{Commit}_{t_j} \in H)$

(B.8) $(\texttt{Commit}_{t_j} \in H) \Rightarrow \forall ob_{t_k} \ \forall p \ (\ (p_{t_j}[ob_{t_k}] \in H) \Rightarrow (Commit_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \ )$

(B.9) $\exists ob_{t_k} \ \exists p \ (Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \Rightarrow (\texttt{Abort}_{t_j} \in H)$

(B.10) $(\texttt{Abort}_{t_j} \in H) \Rightarrow \forall ob_{t_k} \ \forall p \ (\ (p_{t_j}[ob_{t_k}] \in H) \Rightarrow (Abort_{t_j}[p_{t_j}[ob_{t_k}]] \in H) \ )$

(B.11) t satisfait les axiomes de respect des schémas négociés

L'axiome B.1 définit trois événements significatifs associés aux transactions de notre modèle: `Begin`, `Commit` et `Abort`. L'axiome B.2 précise que `Begin` est l'événement de début de ces transactions. L'axiome B.3 indique que `Commit` et `Abort` sont les deux événements de

terminaison associés aux transactions. En outre, les transactions de notre modèle doivent également respecter les axiomes fondamentaux énoncés précédemment (axiome B.4).

L'axiome B.5 définit la vue d'une transaction (i.e. son histoire locale) et l'axiome B.6 indique, pour une transaction t, les opérations (effectuées par d'autres transactions) avec lesquelles des conflits doivent être envisagés lorsque t invoque une nouvelle opération.

Les quatre axiomes B.7 à B.10 garantissent qu'une transaction est *"failure atomic"*, c'est-à-dire que pour une transaction donnée, soit toutes ses opérations sont validées soit aucune ne l'est.

Finalement, l'axiome B.11 impose le respect des axiomes de la définition B.5 en ce qui concerne les schémas de coopération négociés par les transactions.

Bibliographie

[Agr90] D. Agrawal and A.El. Abbadi. Transaction Management in Database Systems. In A.K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kauffman, 1990.

[All95] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Jacky Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, October 1995.

[Alo96] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert Journal*, 1996.

[Atr94] Atria Software Inc. ClearCase product summary. Technical report, Atria Software Inc., 24 Prime park Way, Natick, Massachusetts 01760, 1994.

[Bab93] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*. Addison-Wesley, 2nd edition, 1993.

[Bae93] Ronald M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work*. Morgan Kaufmann Publishers, 1993.

[Bar92a] N. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, 1992. Technical Report CUCS-001-92.

[Bar92b] N. Barghouti. Supporting Cooperation in the MARVEL Process-Centered SDE. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, 17(5):21–31, December 1992.

[Bee88a] D. Beech. A Foundation for Evolution from Relational to Object Databases. In *Proceedings of the International Conference on Extending Database Technology*, pages 251–267, venise, March 1988.

[Bee88b] C. Beeri, H-J. Scheck, and G. Weikum. Multilevel transaction management: Theorical or practical need? In LNCS 303, editor, *Advanced Database Technology Conference*, pages 134–154, march 1988.

[Bel94] N. Belkhatir and J. Estublier. ADELE–TEMPO : An Environment to Support Process Modelling and Enaction. In J. Kramer A. Finkelstein and B. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Study Press, 1994.

[Ben95] R. Bentley, T. Horstmann, K. Sikkel, and J. Trevor. Supporting Collaborative Information Sharing with the World Wide Web: The BSCW Shared Workspace System. In *Proceedings of the 4th International World Wide Web Conference*, pages 63–74, Boston, Massachusetts, 12-14 December 1995. O'Reilly & Associates.

[Ben97a] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel. Basic Support for Cooperative Work on the World Wide Web. *International Journal of Human-Computer Studies: Special issue on Innovative Applications of the World Wide Web*, 46(6):827–846, June 1997.

[Ben97b] R. Bentley, T. Horstmann, and J. Trevor. The World Wide Web as enabling technology for CSCW: The case of BSCW. In *Computer-Supported Cooperative Work: Special issue on CSCW and the Web*, volume 6. Academic Press, 1997.

[Ben98a] K. Benali, G. Canals, C. Godart, and S. Tata. An Approach for Developing Cooperation in Project-Enterprises. In *3 th International Conference on the Design of Cooperative Systems*, Cannes, France, may 1998.

[Ben98b] K. Benali, M. Munier, and C. Godart. Cooperation models in co-design. In *International Conference on Agile Manufacturing (ICAM'98)*, Minneapolis, USA, June 1998.

[Ben99] K. Benali, M. Munier, and C. Godart. Cooperation models in co-design. *International Journal on Agile Manufacturing (IJAM)*, 2(2), 1999.

[Ber81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing surveys*, 13(2):186–221, 6 1981.

[Ber87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Ber97] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, 1997.

[Bes97] J. Besancenot, M. Cart, J. Ferrié, R. Guerraoui, P. Pucheral, and B. Traverson. *Les Systèmes Transactionnels: Concepts, Normes et Produits.* Hermes, 1997.

[Big98] J.C. Bignon, G. Halin, K. Benali, and C. Godart. Cooperation models in co-design: application to architectural design. In *4 th International Conference on Design and Decision Support Sys tems in Architecture and Urban planning*, Maastrich, July 1998.

[Bro97] Gerald Brose. A Java Object Request Broker. Technical Report B 97-2, Université de Berlin, 1997.

[Can96] G. Canals, P. Molli, and C. Godart. Concurrency control for cooperating software processes. In *Proceedings of the 1996 Workshop on Advanced Transaction Models and Architecture (ATMA'96)*, Goa, India, 1996.

[Can98a] G. Canals, C. Godart, P. Molli, and M. Munier. A Criterion to Enforce Correctness of Indirectly Cooperating Applications. *Information Sciences*, 110/3-4:279–302, September 1998.

[Can98b] G. Canals, C. Godart, M. Munier, and S. Tata. Supporting cooperation in project-enterprises. In *Proceedings of the 1998 International Conference on*

Enterprise Networking and Computing (ENCOM'98), Atlanta, Georgia, June 1998.

[Can98c] G. Canals, Claude Godart, F. Charoy, P. Molli, and H. Skaf. COO approach to support cooperation in software developments. *IEE Procedings on Software Engineering*, 145(2-3):79–84, 1998.

[Car89] M. Cart, J. Ferrié, and H. Richy. Contrôle de l'exécution de transactions concurrentes. *Techniques et Sciences Informatiques*, (16):225–240, March 1989.

[Car90a] M. Cart and J. Ferrié. Integrating Concurrency Control into an Object-Oriented Database System. In *Proceedings of Advances in Database Technology - EDBT'90, LNCS-416*, pages 363–377, march 1990.

[Car90b] M. Cart, J. Ferrié, and JF. Pons. Objects Modeling when Using a Multi-Level Transaction Model. In *Proceedings of Workshop on Transaction and Objects. OOPSLA/ECOOP90*, october 1990.

[Chr90] P.K. Chrysanthis and K. Ramamritham. A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proccedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[Chr92] P.K. Chrysanthis and K. Ramamritham. ACTA: The SAGA Continues. In A.K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kauffman, 1992.

[Chr94] P.K. Chrysanthis and K.Ramamritham. Synthesis of Extended Transaction Models. *ACM Transactions on Database Systems*, 19(3):451–491, September 1994.

[Cla96] A. Clarke. A Theoretical Model of Cooperation. In *Proc. 2nd International Conference on the Design of Cooperative Systems (COOP'96)*, Juan-les-Pins, France, june 1996.

[Coi98] C. Cointe and N. Matta. Multi-Agents System to Support Decision Making in Concurrent Engineering. In *TMCE'98*, Manchester, April 1998.

[Dar97] Françoise Darses. L'ingénierie concourante: un modèle en meilleure adéquation avec les processus cognitifs de conception. In C. Chanchevrier P. Bossard and P. Leclair, editors, *Ingénierie Concourante: de la technique au social*, pages 39–55. Economica, 1997.

[Elm92a] A.K. Elmagarmid, editor. *Database transaction models for advanced applications*. Morgan Kauffman, 1992.

[Elm92b] A.K. Elmagarmid, Y. Leu, J.G. Mullen, and O. Bukhres. Introduction to Advanced Transaction Models. In A.K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kauffman, 1992.

[Gar90] G. Gardarin and P. Valduriez. *Bases de données objets, déductives, réparties*. Eyrolles, 1990.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GM87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 12th Annual ACM Conference on the Managemant of Data*, pages 249–259, San Francisco, California, May 1987.

[God96] C. Godart, G. Canals, F. Charoy, P. Molli, and H. Skaf. Designing and Implementing *COO*: Design Process, Architectural Style, Lessons Learned. In *International Conference on Software Engineering (ICSE18)*, 1996. IEEE Press.

[Gra78] J. Gray. Notes on Database Operating Systems. Technical Report RJ2188, IBM Research Report, Febrary 1978.

[Gra93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Gue97] R. Guerraoui. *Transactions réparties: Algorithmes, Systèmes et Langages.* PhD thesis, Habilitation à diriger des recherche, Université Joseph Fournier, 1997.

[Har97] M. Hardwick and R. Bolton. The industrial Virtual Enterprise. *Communications of the ACM*, 40(9), September 1997.

[Hoo95] Frederic Hoogstoel. *Une approche organisationelle du travail coopératif assisté par ordinateur.* PhD thesis, Université des Sciences et Technologies de Lille, 1995.

[Jaj97] S. Jajodia and L. Kershberg, editors. *Advanced Transaction Models and Applications.* Morgan Kauffman, 1997.

[JMA96] Steve Freeman Jean-Marc Andreoli and Remo Pareschi. The coordination language facility: coordination of distributed objects. *Journal of Theory and Practice of Object Systems (TAPOS)*, 2(2):77–94, 1996.

[Kra97] Sarit Kraus. Negotiation and cooperation in multi-agents environments. *Artificial Intelligence*, 94:79–97, 1997.

[LB96] Fabienne Boyer Luc Bellissard, Slim Ben Atallah and Michel Riveill. Distributed application configuration. In *Proceedings of the 16th Internationnal Conference on Distributed Computing Systems (ICDCS 96)*, Hong Kong, 1996. IEEE Computer Society.

[LD97] M.P. Atkinson L. Daynès and P. Valduriez. Customizable concurrency control for persistent java. In *Advanced TRansaction Models and Architecture.* Kluwer Academic Publisher, 1997.

[Mar92] B.E. Martin and C. Pedersen. Long-Lived Concurrent Activities. In Özsu, Dayal, and Valduries, editors, *Distributed Object Management.* Morgan Kauffman, 1992.

[Mat96] N. Matta and O. Corby. Modèles génériques de gestion de conflit dans la conception concourante. Technical Report 3071, INRIA, 1996.

[Mol96] Pascal Molli. *Environnements de Développement Coopératifs.* Thèse en informatique, Université de Nancy I – Centre de Recherche en Informatique de Nancy, 1996.

[Mol97] P. Molli. COO-Transaction: Enhancing Long Transaction Model with Cooperation. In *7th Software Configuration Management Workshop (SCM7), LNCS*, Boston, USA, May 1997.

[Mos81] J. Elliot Moss. *Nested Transactions: An Approach to Reliable Distributed Computing,*. PhD thesis, MIT, 1981.

[MR96] M. Ahamad M. Raynal, G. Thia-Kime. From serializable to Causal Transactions for Collaborative Applications. Rapport de Recherche 2802, IRISA, 1996.

[Mun98] M. Munier and C. Godart. Cooperation services for widely distributed applications. In *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco Bay, USA, 1998.

[Nod92] M.H. Nodine, S. Ramaswamy, and S.B. Zdonik. A Cooperative Transaction Model for Design Databases. In A.K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kauffman, 1992.

[OMG92] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 91.12.1 rev 1.1, Object Management Group, 1992.

[OMG95a] OMG. Common Object Services Specification. Technical Report 95.3.31, Object Management Group, 1995.

[OMG95b] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 2.0, Object Management Group, 1995.

[Orf97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. Wiley, 1997.

[Orf98] Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA, 2nd Edition*. Wiley, 1998.

[Pac94] F. Pacull, A. Sandoz, and A. Schiper. Duplex: A distibuted collaborative editing environment in large scale. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 165–173, Chapel Hill, NC, USA, October 1994. ACM, ACM Press.

[Pu91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 16th Annual ACM Conference on the Management of Data*, pages 377–386, Denver, May 1991.

[Ram93] K. Ramamritham and P.K. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In Özsu, Dayal, and Valduriez, editors, *Distributed Object Management*. Morgan Kauffman, 1993.

[Ram96] K. Ramamritham and P.K. Chrysanthis. A taxonomy of correctness criteria in database applications. *The VLDB Journal*, 5(5):85–97, 1996.

[Ray93] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed objects memories. In *Proceedings of the 4th workshop on Future Trends of Computing Systems*, 1993.

[Ray96] M. Raynal and G. Thia-Kim. Weakening transactions to fit the need of cooperative applications. In *Proc. of the Int. Conf. on Information, System, Analysis and Synthesis (ISAS96)*, number IIIS, pages 531–538, Orlando, juin 1996.

[Rib98] M. Ribière and N. Matta. Virtual Enterprise and Corporate Memory. In *Building, maintaining and using organizational memories workshop of ECAI'98*, Brighton, August 1998.

[Rob98] William N. Robinson and Vecheslav Volkov. Supporting the negotiation life cycle. *Communications of the ACM*, 41(5):95–102, May 1998.

[Ros96] M.A. Rosenman and J.S. Gero. Modelling multiple views of design objects in a collaborative CAD environment. *Computer-Aided Design*, 28(3):193–205, 1996.

[Saa96] M. Saad and M.L. Maher. Shared understanding in computer-supported collaborative design. *Computer-Aided Design*, 28(3):183–192, 1996.

[Sal96] Hélène Saliou. *Conception d'un environnement pour le support de téléactivités coopératives.* PhD thesis, Université de Rennes 1, 1996.

[Sha97] Shapiro, Kloosterman, and Riccardi. PerDiS — a Persistent Distributed Store for Cooperative Application. In *Proceedings of the 3rd Cabernet Plenary Workshop*, IRISA, Rennes, France, April 1997.

[Shi96] André Shiper and M. Raynal. From group communication to transaction in distributed systems. *Communications of the ACM*, 39(4):85–87, 1996.

[Ska89] Andrea H. Skarra. Concurrency Control for Cooperating transactions in an Object Oriented Database. *ACM SIGPLAN Notices*, 24(4), April 1989.

[Ska97] Hala Skaf. *Une approche hybride pour gérer la cohérence dans les environnements de développement coopératif.* Thèse en informatique, Université de Nancy I – Centre de Recherche en Informatique de Nancy, 1997.

[Ske81] D. Skeen. NonBlocking Commit Protocols. In *Proceedings 1981 of the ACM-SIGMOD International Conference on Management of Data*, pages 133–142, 1981.

[Tic89] Tichy and F. Walter. RCS — A system for version control. *Software — Practice and Experience*, 15(7):637–654, July 1989.

[TK97] Gérard Thia-Kime. *Critères de cohérence pour données partagées à support réparti.* PhD thesis, Université de Rennes 1, 1997.

[vdH96] A. van der Hoek, D. Heimbinger, and A. Wolf. A generic, peer-to-peer repository for distributed configuration management. In IEEE Press, editor, *ICSE (International Conference on Software Engineering)*, pages 308–317, Berlin, 1996.

[Wac92] H. Wachter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–258. Morgan Kauffmann, 1992.

[Wei84] W.E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA, March 1984.

[Wei89] W.E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[Wei96] Michael Weiss, Andy Jhonson, and Joe Kiniry. *Distributed Computing: Java, CORBA, and DCE.* Open Software Foundation Version 2.1, February 1996.

[WFM97] WFMC. Terminology & Glossary. Technical Report WFMC-TC-1011, Issue 2.0, Workflow Management Coalition, June 1997.

Monsieur Manuel MUNIER

DOCTORAT DE L'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, AP PROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 25 JAN. 2000

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY 1 — LE PRÉSIDENT

J.P. FINANCE

UNIVERSITÉ HENRI POINCARÉ · NANCY 1

3C-F9

24-30, rue Lionnois B.P. 3069 54013 Nancy cedex · Tél. 03 83 85 48 00 · Fax 03 83 85 48 48

Adresse électronique : _____ @uhp.u-nancy.fr

Résumé

Cette thèse présente un nouveau modèle de transactions avancé permettant non seulement la coopération entre activités par le biais d'échanges d'informations en cours d'exécution, mais également leur distribution et l'hétérogénéïté de leurs relations de coopération. Notre objectif est de décentraliser le contrôle de l'application dans les activités la composant, c'est-à-dire:

- chaque activité est responsable de ses interactions avec les autres activités,
- les contrôles réalisés localement assurent, implicitement, la synchronisation globale du système.

Pour cela, nous avons développé deux critères de correction distribués, à savoir la *D*-sérialisabilité et la *DisCOO*-sérialisabilité. Ils assurent des propriétés globales équivalentes à celles des critères de correction classiques (la sérialisabilité et la *COO*-sérialisabilité) mais en ne se basant que sur des contrôles locaux effectués par chacune des transactions du système.

Outre la décentralisation du contrôle des interactions, nous proposons également des mécanismes permettant aux transactions de négocier les règles (schémas de coopération) à respecter lors de leurs échanges.

Mots-clés: coopération, distribution, transaction, schéma de coopération, négociation.

Abstract

In this thesis we detail a new advanced transaction model that not only supports cooperation between activities through intermediate results exchanges while executing, but also allows them to be distributed and autonomous and to use possibly different cooperation schema to share their data. Our main objective was to decentralize the control of interactions towards the activities themselves, ie:

- each activity coordinates its own interactions with other activities,
- these controls performed locally implicitly ensure the synchronization of the whole system.

With that in mind, we defined two distributed corectness criteria: the *D*-serializability and the the *DisCOO*-serializability. They provide the same global properties than classical corectness criteria (serializability and *COO*-serializability) but they only rely on local controls performed by each transaction.

Besides the decentralization of the control of interactions, our second objective was to define mechanisms to let transactions negotiate the rules (cooperation schemas) they want to verify on their data exchanges.

Keywords cooperation, distribution, transaction, cooperation schema, negotiation.