Service transactionnel souple pour systèmes répartis à objets persistants

Adriana Danes

▶ **To cite this version:**

Adriana Danes. Service transactionnel souple pour systèmes répartis à objets persistants. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 1996. Français. NNT : . tel-00004984

HAL Id: tel-00004984
https://theses.hal.science/tel-00004984

Submitted on 23 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE PAR

DANES ADRIANA

Pour obtenir le titre de Docteur
de l'Université Joseph Fourier − Grenoble 1

(Arrets ministériels du 5 Juillet 1984 et
du 30 Mars 1992)
Spécialité Informatique

Service transactionnel souple pour systèmes répartis à objets persistants

Date de soutenance : 22 Octobre 1996

Composition du jury :	Roland Balter	Président
	Jean Ferrié	Rapporteur
	Daniel Herman	Rapporteur
	Xavier Rousset de Pina	Examinateur
	Sacha Krakowiak	Directeur

Thèse préparée au sein du laboratoire Bull−IMAG et de l'INRIA Rhône−Alpes

Je tiens à remercier :

Sacha Krakowiak, Professeur à l'Université Grenoble I, pour la confiance qu'il m'a accordé pendant ces quatre années et pour avoir accepté la responsabilité de diriger mon travail.

Xavier Rousset de Pina, Professeur à l'Institut Polytechnique de Grenoble, pour son aide et son esprit critique. Les discutions enrichissantes que nous avons eues ont beaucoup contribué à l'aboutissement de se travail.

Jean Ferrié, Professeur à l'Université de Montpellier, qui a accepté d'être rapporteur de ce travail. Les remarques qu'il a apporté m'ont aidé a beaucoup améliorer ce document.

Daniel Herman, Professeur à l'Université de Rennes, pour l'intérêt qu'il a porté à ce travail en qualité de rapporteur, et pour l'évaluation qu'il en a fait.

Roland Balter, Professeur à l'Université Grenoble I, et Directeur du laboratoire "Unité Mixte Bull−IMAG", pour avoir accepté de participer au jury, et surtout, pour m'avoir accueilli au sein de son équipe dans la période 1991−1996.

Jacques Mossière, Professeur à l'Institut Polytechnique de Grenoble, pour son soutien et son aide.

Les membres de l'équipe "Eliott", et en particulier à *Daniel Hagimont*, pour leur aide, leurs précieux conseils et leur encouragements.

Je tiens à remercier aussi les personnes dont l'amitié et le soutien moral ont été aussi importants que le soutien scientifique de ceux que je viens de citer. Ne pouvant tous les énumérer, je ne pas m'empêcher de citer *Claudia Roncancio*, *Hervé Jamrozik*, *Elizabeth Perez Cortes*.

Pour finir, j'ai une pensé tendre pour *Samer Haj Houssain* et *François Exertier*. C'est le fait d'avoir travaillé avec eux durant mon DEA que ma donné le courage de commencer cette thèse.

Introduction

1 Contexte

L'évolution des systèmes informatiques pendant les dernières années est caractérisée par une tendance très forte vers la décentralisation, les configurations réparties étant de plus en plus répandues. Cette évolution est due aussi bien aux facteurs économiques, comme la baisse des prix des ordinateurs personnels, qu'à des facteurs technologiques, dont l'amélioration des performances, tant des machines que des réseaux.

La tendance à la répartition a entraîné une évolution de l'architecture des systèmes d'exploitation et des outils pour la conception des applications réparties.

Le rôle des systèmes d'exploitation répartis est de gérer un ensemble de machines reliées par un réseau d'interconnexion, de façon à donner l'illusion aux utilisateurs de se trouver en face d'une seule machine. Un système d'exploitation réparti doit tirer le meilleur parti des ressources physiques disponibles (processeurs, disques, etc.) et assurer la disponibilité des données qui se trouvent sur les différentes machines.

Dans ce contexte, la conception des applications devient très difficile si les programmeurs doivent tenir compte de la répartition des données, des accès concurrents aux données ainsi que de l'apparition de nouvelles catégories de pannes. Par conséquent, il devient essentiel de fournir aux programmeurs des outils pour faciliter l'écriture des applications réparties. Ces outils doivent permettre :

1. la structuration des applications. Le modèle à objets et son intégration dans des langages de programmation apportent une réponse à ce problème. La notion d'*objet*, entité qui encapsule des données et des opérations à travers lesquelles les données peuvent être manipulées, renforce la modularité des applications.
2. de cacher, ou de rendre transparente, la *répartition des données* aux programmeurs. La *désignation* permet d'accéder à un objet à l'aide d'un nom symbolique, sans savoir où l'objet se trouve physiquement.
3. d'assurer la persistance des objets. Il s'agit de pouvoir créer des objets qui survivent à l'activité qui les a créés (ils existent jusqu'à ce qu'ils soient détruits explicitement). De cette manière, autres activités peuvent les

utiliser par la suite, ce qui facilite la coopération entre activités. Il ne faut pas confondre *persistance* et *permanence* ; cette dernière est une propriété plus forte, qui permet de conserver les objets même en cas de pannes ou d'arrêt des machines.

4. d'assurer la *sécurité* et la *protection* des objets. Il s'agit de contrôler l'accès aux ressources, de manière à ce que les objets soient utilisables seulement par ceux qui en ont le droit et exclussivement dans un mode d'utilisation conforme à leurs droits.
5. d'assurer la *cohérence* des objets lorsqu'ils sont accessibles par plusieurs activités concurrentes. Il s'agit d'assurer la synchronisation des activités au niveau d'un objet donné ou d'un groupe d'objets impliqués dans la réalisation d'une tâche.
6. de cacher, ou de rendre transparentes les *pannes* qui pourraient empêcher le bon déroulement des applications. Les programmeurs ne doivent pas être préoccupés par les effets partiels des applications interrompues par une panne. En outre, les pannes ne doivent pas affecter les résultats des traitements qui sont terminés.

Le travail présenté dans cette thèse a été effectué dans le cadre du projet de recherche Guide (Grenoble Universities Integrated Distributed Environement), mené à l'Unité mixte Bull−IMAG. La version actuelle du système, appelée Guide−2 est réalisée au dessus du micro−noyau Mach 3.0 [1].

L'objectif du projet Guide a été de réaliser un environnement pour le développement et l'exécution des applications réparties. Guide fournit un modèle de programmation à base d'objets qui est intégré dans le langage de programmation du même nom. Le système assure la persistance des objets, permettant ainsi la communication basée sur le partage d'objets persistants. Les objets sont identifiés par des références universelles, et leur localisation est transparente (on accède aux objets locaux de la même manière qu'à ceux situés à distance).

Le système Guide répond aux quatre premiers besoins listés ci−dessus. Ceux−ci correspondent à des thèmes qui ont fait l'objet d'un intérêt particulier (voir [30], [7], [8] et [22]). En revanche, les problèmes liés au maintien de la cohérence et à la tolérance aux pannes n'ont été traitées que partiellement :

− La synchronisation des accès concurrents a été traitée uniquement au niveau des objets individuels (le langage Guide fournit dans ce but des moyens de définir des conditions de synchronisation pour un objet). Cependant, ce mécanisme ne permet pas de coordonner la synchronisation de plusieurs objets (i.e. la synchronisation des activités concurrentes qui partagent plusieurs objets).

− Par ailleurs, la tolérance aux défaillances n'a pas été considérée comme un objectif principal dans la conception initiale de Guide. Des mécanismes de base ont été implantés pour assurer, par exemple, la sauvegarde atomique d'objets ; le système n'est cependant pas capable d'assurer l'exécution atomique des applications en présence de pannes.

Ces lacunes nous ont conduit à définir un *service transactionnel* permettant la construction d'applications fiables dans l'environnement Guide. Dans la suite de l'introduction seront présentés les objectifs sous−jacents au service transactionnel, ainsi que la démarche suivie dans sa conception et sa mise en œuvre ; enfin nous présenterons le plan de la thèse.

2 Motivation et objectifs

L'objectif global de la thèse est de remédier aux lacunes du système Guide en ce qui concerne le maintien de la cohérence des objets face aux accès concurrents incontrôlés et en présence de pannes.

Les *transactions* (ou actions atomiques) sont des unités d'exécution indivisibles. Nous nous sommes orientés, dès le début, vers l'utilisation de ce concept à cause du fait que les transactions offrent un modèle uniforme pour la protection des données contre les accès concurrents et les pannes. À l'origine, les transactions ont été utilisées pour maintenir la cohérence des bases de données.

Des travaux de recherche menés dans le domaine des systèmes répartis ont montré que l'utilisation des transactions permet non seulement de maintenir la cohérence du système, mais aussi de simplifier la conception des applications réparties.

L'apparition des *transactions emboîtées* constitue une évolution dans le domaine des transactions ; une transaction peut être composée par des sous−transactions peuvant s'exécuter en séquence ou en parallèle, le résultat de l'exécution étant le même dans les deux cas. Nous allons montrer dans la suite qu'en dehors du fait qu'elles permettent d'augmenter le parallélisme dans le système, et donc d'améliorer potentiellement les performances, les transactions emboîtées offrent une meilleure résistance aux pannes.

Les *objets atomiques* constituent un autre thème de recherche autour de la conception des applications réparties. Il s'agit d'objets qui offrent, en plus des

opérations permettant leur manipulation, la possibilité de synchroniser les opérations qui leurs sont appliquées et de permettre l'exécution atomique de ces opérations. Des projets de recherche tel qu'Argus [33] et Arjuna [41] ont exploré l'utilisation des objets atomiques pour la mise en œuvre des transactions. Dans ce contexte, l'atomicité des transactions est basée sur les propriétés des objets atomiques et les objets atomiques sont utilisés uniquement au sein de transactions.

L'utilisation des objets atomiques permet d'adapter les algorithmes de synchronisation à la sémantique des applications. L'objectif est d'augmenter la concurrence par rapport à celle permise par les algorithmes classiques basés uniquement sur des opérations de type lecture / écriture.

Compte tenu de ces résultats, il nous a semblé un objectif réaliste de fournir aux concepteurs d'applications réparties dans l'environnement Guide les moyens d'utiliser des transactions emboîtées et des objets atomiques. Nous avons opté pour l'approche "service" ou "outils". Cette approche, si on la compare avec celle d'intégration dans le langage de programmation, facilite la mise en œuvre des mécanismes requis et leur évolution.

Nous avons souhaité fournir un service souple, qui permette de dissocier les concepts de transaction et d'objet atomique et de minimiser les restrictions liées à leur utilisation. La souplesse a pour but de permettre le choix entre plusieurs degrés de cohérence. Cette facilité vise à trouver un compromis entre les besoins de cohérence et ceux de performance (plus la cohérence souhaitée est forte, plus elle nécessite des mécanismes coûteux).

3 La démarche suivie

La mise en œuvre de mécanismes souples nécessite la définition d'un modèle de concurrence moins strict que celui imposé par les transactions classiques. Par conséquent, la première étape de notre travail a été l'étude des modèles transactionnels et la proposition d'un modèle étendu, les extensions correspondant aux besoins concrets liés à l'utilisation du service.

Dans un deuxième temps, nous avons défini une architecture permettant l'intégration du service transactionnel dans le système Guide. L'architecture présente deux couches. La couche haute est composée d'un ensemble de classes spécialisées qui d'une part constituent l'interface du service, et d'autre part mettent en œuvre une partie des mécanismes requis.

Les classes spécialisées sont des outils permettant l'utilisation des transactions et des objets atomiques grâce à des mécanismes spécifiques à l'approche de programmation à base d'objets : instanciation des classes, définition de nouvelles classes par héritage avec surcharge de méthodes héritées, appels de méthode sur les instances.

Les classes spécialisées s'appuient sur des mécanismes implantés au niveau du noyau du système Guide, qui constituent la couche basse du service transactionnel. Certains de ces mécanismes ont été rajoutés par nous, d'autres ont été modifiés pour la mise en œuvre des transactions et des objets atomiques.

Les outils fournis permettent la programmation d'applications Guide ayant les caractéristiques suivantes :

- Elles peuvent être soit transactionnelles, soit avoir des parties qui s'exécutent de manière transactionnelle, le reste des traitements étant non−transactionnels.
- Les applications peuvent définir et utiliser des objets atomiques. Le système garantit l'atomicité *(2 aspets : atomicité face à la concurrence et face aux pannes)* des opérations appliquées aux objets atomiques, qu'ils soient utilisés à l'intérieur d'une transaction ou bien en dehors des transactions. Le service transactionnel assure la cohérence globale des objets atomiques utilisés au sein d'une transaction.
- Le modèle de synchronisation permet l'exécution d'activités concurrentes pour le compte d'une même transaction, et pas seulement l'exécution de sous−transactions concurrentes.

  De plus, le système assure la synchronisation entre transactions et méthodes non−transactionnelles, ainsi que la synchronisation des méthodes non−transactionnelles concurrentes.
- Le service transactionnel fournit des outils pour la programmation des transactions emboîtées qui permettent un certain degré de coopération entre les composantes transactionnelles. Par exemple, une transaction mère peut donner à ses descendants des droits d'accès à certains des objets qu'elle a modifiés.
- Le service transactionnel assure l'atomicité des transactions lors de l'abandon programmé des transactions ainsi que lors des pannes d'application (interruptions dues à des erreurs de programmation) ou des pannes de matériel. Actuellement, les mécanismes qui assurent la résistance aux pannes matérielles sont effectifs seulement pour les transactions non−réparties (lorsque les objets utilisés par une transaction sont couplés

sur le site où la transaction est créée). Les mécanismes nécessaires pour la résistance aux pannes en réparti sont spécifiés mais pas encore implantés.

4 Plan de la thèse

La première partie de la thèse apporte les éléments nécessaires à la compréhension du travail que nous avons effectué. Il s'agit d'une vue globale sur les concepts utilisés dans le cadre de la thèse et sur leur mise en œuvre dans plusieurs systèmes (chapitre I), ainsi qu'une présentation succincte du système Guide (chapitre II). La deuxième partie est consacrée au service transactionnel. Après une description générale dans le chapitre III nous aborderons les deux aspects fondamentaux du service : la synchronisation, (qui assure la cohérence des données en présence des accès concurrents), dans le chapitre IV et l'atomicité, (qui assure la cohérence des données, dans la présence des pannes), dans les chapitres V et VI. Une évaluation du service est proposée dans le chapitre VII.

Le chapitre I est consacré à la définition des concepts fondamentaux utilisés dans la thèse tels que *transactions* et *objets atomiques*. La présentation des propriétés des transactions et des objets atomiques sera suivie par une vue synthétique sur les modèles transactionnels. Nous présenterons ensuite quelques systèmes répartis qui utilisent les transactions et les objets atomiques pour assurer la fiabilité des applications.

Le chapitre II présente l'environnement Guide : le modèle de programmation et le langage Guide d'une part, le modèle d'exécution d'autre part. Les principales composantes du système sont passées en revue afin de permettre par la suite, une meilleure compréhension de l'architecture du service transactionnel.

Les caractéristiques générales du Service Transactionnel Guide (STG) sont présentées dans le chapitre III. Nous y décrivons les principales classes spécialisées, en les considérant du point de vue de leur utilisation lors de l'écriture des programmes Guide. L'identification concrète des besoins en termes d'utilisation du service permet de cerner les caractéristiques que le modèle transactionnel doit présenter. Ces résultats sont utilisés dans les chapitres suivants pour définir le modèle de concurrence et d'atomicité.

Le chapitre IV est consacré au modèle de concurrence du STG. Ce modèle est une extension du modèle de concurrence défini par Moss pour les transactions emboîtées [35]. Pour développer un tel modèle nous avons suivi la démarche suivante :

− identification des éléments qui ont permis à Moss d'étendre le modèle de concurrence des transactions plates, et analyse des besoins auxquels répondent ces extensions.

− étudie des différentes propositions d'extension du modèle de Moss présentes dans la littérature.

Sur la base de ces éléments, nous avons enrichi le modèle de Moss de manière progressive, de façon à arriver à un modèle qui prend en compte tous les besoins définis dans le chapitre III.

Le modèle obtenu est concrétisé par un ensemble de règles de verrouillage et de marquage qui définissent les conditions d'utilisation des objets atomiques au sein des transactions et hors transaction. Nous présentons ensuite l'implantation de ces règles au niveau des classes spécialisées, ainsi que le support système correspondant.

Les chapitres V et VI sont consacrés au maintient de l'atomicité des transactions et des objets atomiques. À l'opposé de l'approche descendante, utilisée pour la définition du modèle de concurrence, nous avons dû adopter une approche ascendante pour la définition du modèle d'atomicité, de manière à prendre en compte les contraintes architecturales imposées par le système Guide. L'analyse de ces contraintes et le modèle d'architecture qui en découle sont présentés au début du chapitre V. La suite de ce chapitre est consacrée à l'atomicité des transactions non−réparties et à l'atomicité des opérations appliquées aux objets atomiques. Les problèmes spécifiques qui découlent de la répartition sont traités dans le chapitre VI.

La conception des mécanismes requis pour le maintien de l'atomicité a été effectuée selon une démarche progressive. Nous avons d'abord mis en œuvre, au niveau des classes spécialisées, le support pour l'abandon programmé des transactions. Les mécanismes pour la récupération après une panne d'application ont nécessité l'extension de l'architecture avec des entités actives, appelés *gérants transactionnels* et s'exécutant sur chaque site Guide. Le fonctionnement des gérants transactionnels a été finalement étendu pour la prise en compte de la répartition des transactions.

Une évaluation qualitative du STG fait l'objet du chapitre VII. Nous proposons une analyse comparative avec d'autres systèmes étudiés, sur des aspects liés à l'approche à base d'objets et aux modèles de concurrence et d'atomicité adoptés.

La conclusion dresse un bilan du travail réalisé et évoque un certain nombre de perspectives.

Chapitre I

Transactions et objets atomiques : utilisation et mise en œuvre

Ce chapitre présente les propriétés des transactions et des objets atomiques, afin d'expliquer pourquoi ces abstractions facilitent la mise en œuvre d'applications réparties fiables. Un deuxième objectif de ce chapitre est l'identification des principaux mécanismes nécessaires à l'implantation de ces concepts dans le cadre d'un système réparti.

L'exécution d'une transaction est conforme à un modèle bien défini. L'analyse du modèle canonique des transactions plates nous permet d'identifier les besoins qui ont conduit aux différentes extensions. Parmi ces extensions, nous allons étudier le modèle des transactions emboîtées.

Dans la dernière partie de ce chapitre, nous présentons quelques systèmes répartis qui utilisent des transactions et des objets atomiques pour la mise en œuvre d'applications réparties fiables.

I.1 Les transactions

Du point de vue du programmeur d'applications, une *transaction* est constituée par un ensemble d'opérations, précédé par une opération qui marque le début de la transaction et terminé par une opération qui marque sa fin (ces opérations sont représentées par des crochets dans la Fig. 1.1). L'exécution d'une transaction est caractérisée par le fait qu'une et une seule des assertions suivantes est vraie :

- toutes les opérations de l'ensemble sont effectuées (ce cas correspond à la *validation* de la transaction);
- aucune des opérations n'est effectuée (ce qui correspond au *rejet*, ou à l'*abandon* de la transaction).

Quelle que soit la cause du rejet, les effets partiels de la transaction sont annulés. Ce comportement est garanti par la propriété d'*atomicité* des transactions

(appelée aussi *tout−ou−rien*). Cette propriété est très importante dans le cas des transactions réparties(1), car elle représente un élément essentiel de la tolérance aux pannes de ces systèmes.

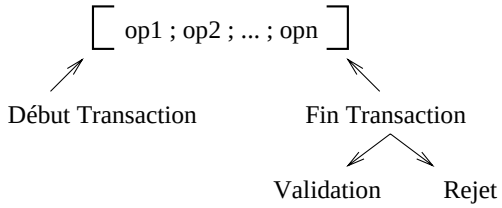

Fig. 1.1 : Représentation schématique d'une transaction

Les opérations d'une transaction sont normalement exécutées en séquence (il s'agit dans ce cas d'un modèle classique), mais il peut y avoir du parallélisme entre l'exécution des opérations dans le cas des modèles transactionnels avancés, tels que le modèle des transactions emboîtées ou multi−niveaux (voir I.3).

Les opérations sont appliquées à des objets qui peuvent être partagés (plusieurs applications utilisant ces objets peuvent s'exécuter de manière concurrente). De plus, les objets peuvent être modifiés. Par conséquent, les objets partagés doivent être protégés contre les incohérences engendrées par des accès concurrents incontrôlés(2). Les transactions ont cette capacité de protection grâce à leur propriété d'*isolation*. Cette propriété garantit que, si plusieurs transactions s'exécutent de manière concurrente, alors tout se passe comme si chacune s'exécutait seule dans le système.

Lorsque les transactions sont utilisées dans les bases de données, elles sont caractérisées par une troisième propriété : la *cohérence*. Cette propriété garantit que, si une transaction qui s'exécute seule dans le système modifie des données dans une base et si la base se trouve dans un état cohérent avant l'exécution de la transaction, alors celle−ci passe dans un état cohérent après l'exécution de la transaction. Ce type de cohérence, liée à la sémantique des opérations effectuées par une transaction, est laissée à la charge des programmeurs et réalisée par des mécanismes de plusieurs sortes (déclencheurs, règles actives, etc.), dont l'étude ne fait par partie de la problématique de cette thèse.

---

(1) Les opérations peuvent être appliquées à des objets qui se trouvent éparpillés sur plusieurs sites d'un système réparti.

(2) Des exemples de telles incohérences sont les lectures successives d'une même donnée qui donnent des valeurs différentes, ou les pertes de mise à jour (lors de la lecture d'une donnée, qui est précédée par une écriture sur la même donnée, la valeur obtenue est différente de celle ayant été écrite).

La dernière propriété des transactions est la *permanence* ou la *durabilité*. Celle−ci garantit que les effets d'une transaction validée ne peuvent pas disparaître suite à des pannes logicielles ou matérielles(3).

Les quatre propriétés décrites ci−dessus sont désignées dans la littérature par le sigle ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*).

Dans la suite de cette section, nous décrivons les mécanismes qui garantissent la propriété d'*isolation* ainsi que les propriétés d'*atomicité* (ou *tout−ou−rien*) et de *permanence* (ou *durabilité*).

I.1.1 Isolation

Un moyen simple pour assurer l'isolation des transactions est de les exécuter en séquence. Ceci n'est cependant pas acceptable, car les performances du système seraient fortement réduites. Une autre solution est de permettre les exécutions concurrentes, mais de garantir que ces exécutions sont équivalentes (ayant le même effet) à une exécution en série des mêmes transactions. De telles exécutions sont dites *sérialisables*.

Exemple. Supposons qu'une transaction T1 modifie la valeur de plusieurs objets, alors que T2 lit la valeur de ces objets. La sérialisation garantit que T2 voit tous ces objets soit dans leur état initial, soit dans l'état d'après les modifications effectuées par T1, et ceci quel que soit l'ordonnancement des opérations qui composent les deux transactions. Dans le premier cas T2 est sérialisée avant T1, alors que dans le deuxième cas l'ordre de sérialisation est : T1, T2.

Dans la suite nous présentons les techniques de base utilisées pour la sérialisation des transactions.

La sérialisation des transactions est assurée par un mécanisme de contrôle de concurrence qui met en œuvre un *protocole de synchronisation*. Le rôle d'un protocole de synchronisation est de détecter et de résoudre les *conflits* entre les transactions. Deux transactions sont en conflit si l'une des transactions peut observer les effets de l'autre, ou si l'une des transactions invalide les effets de l'autre. Plus précisément, en tenant compte de la propriété de *commutativité*(4) des opérations,

---

(3) En général, on prend en compte les pannes non catastrophiques. La résistance aux pannes catastrophiques, tel que la destruction des disques, nécessite des techniques spécifiques.

(4) Deux opérations op1 et op2 appelées sur un objet x par deux transactions différentes sont commutatives si elles produisent le même effet sur l'objet x et sur les transactions, indépendamment de l'ordre dans lequel elles sont exécutées.

deux transactions sont en conflit si elles appellent des opérations non−commutatives sur un même objet.

Il existe deux catégories principales de protocoles de synchronisation : les protocoles dits *pessimistes* et ceux dits *optimistes* (voir [6] pour une revue des protocoles de synchronisation). Dans le cas des protocoles pessimistes, la détection des conflits est faite de manière continue (au cours de l'exécution des transactions), alors que dans le cas des protocoles optimistes, le contrôle de la séralialisabilité est effectué lors de la validation des transactions.

Les protocoles optimistes sont le plus souvent basés sur une méthode de certification [35]. Cette méthode impose un ordre de sérialisation des transactions qui est identique à leur ordre de validation. Cependant, le protocole optimiste proposé en [7], basé sur l'utilisation d'intervalles d'estampillage, permet l'obtention d'un ordre de sérialisation différent de celui de l'ordre de validation, ce qui permet de diminuer le nombre des transactions rejetées.

Les protocoles pessimistes sont souvent basés sur la technique de verrouillage [20], [34] : tout objet doit être verrouillé avant de pouvoir être utilisé par une transaction. Si le verrouillage n'est pas possible à cause d'un conflit, la transaction est suspendue jusqu'à la disparition du conflit. Les attentes consécutives aux conflits peuvent créer des inter−blocages (i.e. plusieurs transactions forment un cycle d'attentes) ; différentes techniques de traitement des inter−blocages sont présentées en [5].

Le protocole de verrouillage est à *deux phases* si la transaction ne verrouille plus d'objets après avoir commencé à déverrouiller des objets. Si les verrous en écriture sont gardés jusqu'à la fin de la transaction, le protocole est dit strict. Cette technique diminue le potentiel de concurrence, mais évite les *rejets en cascade*(5). L'ordre de sérialisation est défini de manière dynamique, par l'ordre dans lequel les transactions atteignent ce qu'on appelle le *point de verrouillage maximale* (l'instant où tous les objets utilisés par la transaction sont verrouillés pour le compte de celle−ci).

Pour détecter les conflits entre transactions, les protocoles de synchronisation sont souvent basés sur le mode d'utilisation des objets. En général, les deux modes considérés sont la *lecture* et l'*écriture*. Les protocoles garantissent dans ce cas une politique d'utilisation des objets de type "un seul écrivain, plusieurs lecteurs" (voir IV.2.1 pour une définition précise des règles de verrouillage). Cette politique permet

---

(5) Si un verrou en écriture est relâché avant la fin de la transaction T qui a effectué le verrouillage, alors l'objet modifié devient visible à d'autres transactions. Si finalement T est rejetée, alors les transactions qui ont utilisé cet objet doivent aussi être abandonnées.

d'éviter l'apparition des dépendances entres transactions, ce qui est une condition suffisante pour assurer la sérialisation des transactions.

L'intérêt essentiel présenté par la détection des conflits sur la base des modes d'accès lecture/écriture réside dans le fait que la mise en œuvre du protocole est indépendante des applications transactionnelles, grâce à la nature syntaxique des informations requises pour la détection des conflits.

L'inconvénient de ce type d'approche, par rapport à une approche qui prend en compte la sémantique des opérations (voir plus bas), est qu'elle fournit un degré de concurrence inférieur à celui qu'on pourrait atteindre tout en gardant la sérialisation comme critère de correction.

Pour augmenter la concurrence dans le système on peut diminuer le grain du verrouillage. Par exemple, dans une base de données relationnelle où les données sont organisées sous la forme de relation, le degré de concurrence est plus élevé si on verrouille uniquement le tuple contenant la donnée utilisée que si on verrouille le tableau auquel la donnée appartient.

Une autre méthode est de prendre en compte des informations sur la sémantique des opérations lors de la détection des conflits [47] [52] [9] . Cette approche est facilitée par l'introduction des types abstraits de données et des objets atomiques dans la conception des systèmes transactionnels (voir I.2). En contrepartie, elle rend les mécanismes plus complexes et l'automatisation des mécanismes est moins évidente (les concepteurs d'applications ne peuvent plus ignorer les problèmes dus à la concurrence). Pour ces raisons, il n'est intéressant d'utiliser cette approche que pour accéder aux objets utilisés très fréquemment.

I.1.2 Atomicité et Permanence

Les propriétés d'atomicité et de permanence (ou durabilité) sont liées au point de terminaison d'une transaction. Reprenons la représentation schématique d'une transaction présentée dans la Fig. 1.1. Les crochets marquent des *points de cohérence observables*. Lorsque la transaction s'exécute seule, elle fait passer le système d'un état cohérent initial dans un état cohérent final, le système étant dans un état incohérent transitoire durant l'exécution de la transaction. Si la transaction n'arrive pas au point de terminaison, alors le système doit revenir à son état initial. Dans le cas contraire, le nouvel état doit être rendu observable et stable, ç.à.d. que les effets deviennent accessibles à d'autres applications et permanents (donc insensibles aux pannes). On dit alors que la transaction est *validée*.

L'interruption d'une transaction peut être due à plusieurs types d'événements.

− La transaction ne peut pas respecter certaines contraintes de cohérence attachées aux objets qu'elle manipule (par exemple, dans une transaction bancaire, le compte qui doit être débité est vide).

− Un manque de ressources dû à des accès concurrents.

− Une interruption de la transaction par l'utilisateur.

− Une interruption de la transaction par le système, à cause d'une erreur dans le code de l'application transactionnelle (par exemple, une division par zéro).

Dans tout ces cas, regroupés dans [40] sous le terme de *faute directe de transaction*, la transaction est *rejetée* ou *abandonnée*. Par ailleurs, l'exécution d'une transaction peut être interrompue par une panne de matériel : défaillance de communication, défaillance de site, ou défaillance de la mémoire stable (ou secondaire). Une telle défaillance peut avoir des conséquences sur plusieurs transactions qui se trouvent en cours d'exécution dans le système. Les mécanismes dites de *reprise* doivent effectuer des actions pour garantir que le système retrouve un état cohérent après la réparation de la panne.

Les actions que le système transactionnel doit entreprendre, en cas de l'abandon d'une transaction ou en cas de reprise après panne, sont dépendantes de l'architecture du système sous−jacent. Dans la suite nous présentons l'influence de cette architecture sur les mécanismes d'abandon et de reprise (I.1.2.1) et les techniques de base utilisées par ces mécanismes (I.1.2.2). Enfin, nous mettons en évidence la manière dont ces mécanismes doivent être adaptés lorsqu'ils sont implantés dans un système réparti (I.1.2.3).

I.1.2.1 L'impact de l'architecture du système

Les actions à entreprendre lors de l'abandon d'une transaction ou en cas de reprise après panne sont dépendantes de l'architecture du système sous−jacent et du modèle de mise en œuvre des transactions. Plus précisément, il est nécessaire de prendre en compte :

− la politique de transfert des données entre la mémoire de stockage (disques) et la mémoire d'exécution (volatile) qui est le lieu de travail des transactions.

− le modèle de mise à jour adopté : celui à *effets immédiats* ou celui à *effets différés*.

On peut distinguer les politiques de transfert suivantes [6] :

− La première, n'impose aucune contrainte sur l'échange des données entre les deux niveaux de mémoire. Un objet qui a été modifié par une transaction peut être transféré dans la mémoire de stockage avant la validation de celle−ci. En cas d'abandon ou d'interruption de la transaction il est nécessaire de défaire ces effets

(on parle de technique *undo*). Par ailleurs, un objet qui a été modifié par une transaction arrivée au point de terminaison par validation peut ne pas être transféré dans la mémoire de stockage lorsqu'une panne survient. La perte du contenu de la mémoire volatile nécessite de refaire les effets de la transaction (on parle de technique *redo*).

− La seconde, impose le transfert des objets modifiés par une transaction dans la mémoire de stockage avant la validation de celle−ci. Par conséquent, il n'est jamais nécessaire de refaire une transaction (on parle de technique *no redo*).

− La troisième, interdit le transfert des objets modifiés par une transaction dans la mémoire de stockage avant la validation de celle−ci. Par conséquent, il n'est jamais nécessaire de défaire les effets d'une transaction (on parle de technique *no undo*).

Il existe aussi des architectures où il est possible de ne faire ni *undo*, ni *redo* ; cependant, cela nécessite la mise en place des techniques supplémentaires, telles que les *pages ombres* [23].

Le modèle de mise à jour définit le moment où les modifications effectués par une transaction apparaissent dans la base (l'union de l'espace de stockage avec l'espace de travail des transactions). Dans le cas du modèle à mises à jour immédiates (MAJI), l'effet d'une opération effectuée par une transaction est immédiatement répercuté sur la base, alors que dans le modèle à mises à jours différées (MAJD) cet effet est répercuté sur la base uniquement au moment où la transaction valide.

Lors de l'abandon d'une transaction il faut défaire les effets de celle−ci dans le cas du modèle MAJI, alors qu'il n'y a rien à faire dans le cas des MAJD. Lors de la validation d'une transaction, il faut répercuter ses effets sur la base dans le cas du modèle à MAJD, ce qui n'est pas nécessaire dans le cas des MAJI. Voir [25] pour une étude détaillée sur l'impact des modèles à MAJI et MAJD sur le contrôle de concurrence, le rejet, la validation et la reprise après pannes.

I.1.2.2 Techniques de base

Différentes techniques peuvent être utilisées pour mettre en œuvre l'atomicité et la durabilité des transactions. Elles sont basées sur la gestion d'informations redondantes, qui sont gardées dans des structures de données appelées *journaux*. Les informations stockées dans un journal doivent permettre de défaire et/ou refaire les effets des transactions.

Pour défaire les effets d'une transaction, un journal doit contenir des informations permettant de retrouver l'état initial des objets modifiés. Une telle information peut être la valeur d'un objet avant sa modification (appelée *image avant*), ou une

opération (appelés *opération inverse)* qui doit être appliquée à l'objet pour retrouver sa valeur initiale. Les journaux contenant ce type d'information (*undo logs* [22]) sont utilisés pour annuler les effets des transactions rejetées.

De façon symétrique, pour reproduire les effets d'une transaction, un journal doit contenir des informations permettant de retrouver l'état final des objets modifiés : soit la valeur de l'objet après la modification (appelée *image après*), soit l'opération (dite *directe*) qui a été appliquée à l'objet. Les journaux contenant ce type d'information, appelées *journaux après* (*redo logs*), sont utilisés pour la validation des transactions arrivées à leur point de terminaison.

Il existe deux types de journalisation, *physique* et *logique*. La journalisation est physique lorsqu'on enregistre des valeurs d'objet. Elle est logique dans le cas ou les enregistrements sont des opérations. Le choix du type de journalisation est lié à l'approche utilisée pour la détection des conflits. Dans le cas où le protocole de synchronisation implanté par le mécanisme de contrôle de concurrence est basé sur les opérations de type lecture/écriture, la journalisation peut être physique. En revanche, la prise en compte de la sémantique des opérations dans la détection des conflits nécessite une journalisation logique. Considérons par exemple un protocole de synchronisation qui permet à plusieurs transactions concurrentes d'appliquer une opérations *Add(x)* sur un même objet *Obj*, où *Add* additionne la valeur du paramètre *x* à la valeur de *Obj*. Alors le rejet d'une des transactions doit annuler uniquement l'effet de cette transaction (soustraire la valeur qu'elle a additionné, et pas revenir à l'état initial de *Obj*, car ceci pourrait annuler l'effet d'une autre transaction, qui à son tour, a modifié *Obj*).

I.1.2.3 L'impact de la répartition

Revenons sur la terminaison d'une transaction. Si la transaction est répartie, il n'est pas certain qu'elle puisse être validée, malgré le fait qu'elle s'est déroulée normalement jusqu'au point de terminaison. Par exemple, elle a pu modifier des objets se trouvant sur un site qui est ensuite tombé en panne. Si les effets de la transaction sont perdus sur le site défaillant, la transaction ne peut pas être validée.

Autrement dit, la validation d'une transaction répartie nécessite un protocole permettant d'établir un consensus entre les participants[6] de la transaction. La mise en œuvre d'un tel protocole est compliquée par le fait qu'il doit être résistant aux pannes (par exemple, il doit prendre en compte les pannes de communication). Par ailleurs, il est souhaitable qu'il puisse permettre aux participants d'abandonner la

---

(6) Une transaction répartie peut être vue comme un ensemble de participants qui représentent la transaction sur les sites où elle se déroule. Il existe plusieurs modèles qui définissent la relation entre les représentants : centralisé, hiérarchique, en réseau.

transaction à tout moment (par exemple, pour débloquer une ressource afin qu'une autre transaction puisse y accéder). En fait, ceci n'est pas toujours possible, chaque protocole ayant "une fenêtre de vulnérabilité" dans laquelle les participants n'ont pas le droit d'abandonner la transaction unilatéralement. Un participant entre dans la fenêtre de vulnérabilité lorsqu'il est prêt à valider ; après avoir communiqué cette décision aux autres participants, il reste bloqué jusqu'à l'obtention d'un consensus sur le mode de terminaison (par validation ou par abandon).

Dans la suite nous présentons le protocole de validation à deux phases [22] qui est le plus répandu, car assez général et performant.

Le protocole de validation à deux phases

Considérons le modèle centralisé (l'un des participants est le coordinateur et les autres sont les subordonnés).

1. Dans la 1−ère phase, le coordinateur demande aux participants s'ils peuvent valider la transaction.
   - Si un des participants répond NON, la transaction est abandonnée.
   - Un participant répond OUI, s'il a réussi à préparer la transaction pour la validation (par exemple, il a sauvegardé dans le journal les enregistrements permettant de *refaire* localement la transaction). De plus, en répondant OUI, le participant renonce au droit d'annuler la transaction unilatéralement (il entre dans la fenêtre de vulnérabilité) et il attend la décision du coordinateur.
2. Le coordinateur décide de valider la transaction si tous les participants ont répondu par OUI. Dans ce cas, il doit enregistrer dans le journal le fait que la transaction peut être considérée comme validée. À partir de cet instant, rien ne peut empêcher la validation de la transaction.

   Dans la deuxième phase, le coordinateur fait connaître sa décision aux participants des autres sites. Si la décision est celle de valider, les participants valident localement les effets de la transaction. Dans le cas contraire, les participants qui ont répondu OUI abandonnent localement la transaction.

I.2 Les objets atomiques

La programmation à objets doit son succès au fait qu'elle facilite l'utilisation des données à structure complexe et renforce la modularité des applications. À partir de l'instant où des applications transactionnelles ont été écrites en termes d'objets, il est apparu naturel d'étudier comment l'approche à objets pouvait faciliter la conception et améliorer les performances des transactions.

Le terme d'*objet atomique* est dérivé de celui d'*action* ou *activité atomique*.

Une action atomique est en fait une transaction. Ce terme est préféré par des nombreux chercheurs de la communauté des systèmes répartis, qui ont par ailleurs reconsidéré les propriétés des transactions. Les deux propriétés suivantes ont été retenues :

− l'*atomicité à la concurrence*, qui correspond à la propriété d'*isolation*, et

− l'*atomicité aux pannes*, qui regroupe les propriété d'*atomicité* et de *durabilité*.

Par extension, un type de données atomique [54] est un type abstrait ayant les propriété d'isolation et d'atomicité aux pannes. Un objet atomique est une instance d'un type de données atomique.

Dans la pratique, on attache à chaque objet atomique des mécanismes qui sont capables d'assurer leur propre isolation et atomicité aux pannes. Ceci conduit à une décentralisation des mécanismes qui, dans les architectures classiques, sont coordonnés par des modules spécialisés appelés gérants transactionnels.

Remarque : Dans le cadre de cette thèse, le concept d'*objet atomique* correspond à la définition ci−dessus. Cependant, lorsque nous parlons de la *propriété* d'*atomicité*, nous l'utilisons dans son acception canonique de propriété tout−ou−rien des transactions (correspondant à la lettre A du sigle ACID).

Une étude théorique des objets atomiques a été effectué par Weihl [53]. Il s'est intéressé à la formalisation des propriétés locales des objets atomiques, ainsi qu'aux conditions dans lesquelles l'utilisation des objets atomiques assure l'atomicité des actions.

L'atomicité locale garantit la cohérence interne d'un objet individuel, alors que l'atomicité globale assure la cohérence d'un ensemble d'objets utilisés par une action (ou transaction).

Weihl a identifié plusieurs types ou critères d'atomicité, parmi lesquels on peut distinguer l'*atomicité dynamique*, l'*atomicité statique* et l'*atomicité hybride*. Il a démontré que, pour assurer l'atomicité globale, les deux conditions suivantes doivent être respectées :

1. les transactions ne partagent que des objets atomiques,
2. le même critère d'atomicité doit être utilisé pour tous ces objets. Par exemple, si les objets ont la propriété d'atomicité dynamique, ceci garantit la sérialisation dynamique des transactions.

Une description informelle de ces critères peut être trouvée en [54].

Pour illustrer, considérons le cas de l'*atomicité dynamique*. Le comportement d'un objet ayant cette propriété peut être caractérisé de la manière suivante : dans le cas où un conflit apparaît dans une séquence d'opérations appliquée à l'objet par deux transactions concurrentes, une ou plusieurs des opérations exécutées par une des transactions sont retardées jusqu'à la terminaison (validation ou abandon) de l'autre transaction. L'ordre de sérialisation des transactions est déterminé de manière dynamique, en fonction de l'ordre d'accès aux objets.

Si les objets sont caractérisés par l'*atomicité dynamique*, alors l'ordre de sérialisation des transactions qui les utilisent est fonction de l'ordre d'accès aux objets. À l'opposé, si les objets sont caractérisés par l'*atomicité statique*, alors la sérialisation des transactions est faite selon un ordre pré−défini, fixé lors du début des transactions. L'*atomicité hybride* combine les deux techniques. Elle requiert la partition des transactions en deux catégories : celles qui ne font que des opérations de lecture et celles qui font aussi des mises à jour. Les transactions de la première catégorie sont sérialisées selon un ordre pré−défini qui est fixé au début de leur exécution, alors que pour les transactions de la deuxième catégorie l'ordre de sérialisation est fixé à leur validation. Un protocole de sérialisation hybride peut être réalisé en utilisant la technique d'estampillage [32].

Afin de garantir l'atomicité locale des objets, la mise en œuvre des mécanismes de contrôle de l'atomicité doit être réalisée au niveau des objets et non pas par le système transactionnel. Ceci permet d'exploiter la sémantique des objets typés pour accroître la concurrence des transactions. L'utilisation des différentes formes de commutativité entre les opérations définies sur un même objet, identifiés en [9], permettent de reconnaître comme sérialisables des exécutions qui ne le seraient pas en termes de conflits portant sur des opérations lire/écrire.

La spécification et la mise en œuvre des objets atomiques a été expérimentée par plusieurs projets de recherche. La spécification d'un type atomique nécessite deux parties, qui sont plus ou moins séparées en pratique :

- définition d'un état et des opérations qui peuvent le manipuler ;
- définition des contraintes sur l'exécution des opérations (description des exécutions concurrentes permises et des actions à entreprendre en cas de panne).

Différentes voies ont été exploitées dans la spécification des objets atomiques, dont les plus connues sont : la définition d'un nouveau langage, comme dans le cas d'Argus [38], l'extension de langages existants, comme dans le cas d'Avalon [31], et l'utilisation d'un langage à objets existant, dans le cas d'Arjuna [44].

Les outils fournis par Argus [53] ont une forte puissance d'expression permettant la définition de types atomiques très efficaces. Mais leur utilisation est complexe, ceci étant dû principalement au fait que les deux aspects mis en évidence ci−dessus sont mélangés. Les travaux présentés par Malta en [40], visent à remédier cet inconvénient par la séparation de ces aspects ; dans le langage proposé, la spécification d'une classe atomique comporte trois parties : une interface publique, une partie implantation et la partie contrôle qui définit la correspondance entre les deux premières, ainsi que les contraintes. Une approche similaire a été adoptée dans le langage PC++, une extension de C++, où, en plus de la partie privée et de la partie publique d'une définition de classe, celle−ci comporte une troisième partie, contenant des directives nécessaires à la synchronisation des opérations [56].

L'originalité d'Avalon et d'Arjuna est d'avoir proposé l'héritage pour la définition des types atomiques. Les mécanismes nécessaires à la synchronisation et à la récupération après panne sont définis par des des classes spécialisées. Des classes d'objets atomiques peuvent être définis par les programmeurs comme des sous−classes des classes spécialisées.

En conclusion, l'utilisation d'une approche à objets dans la programmation des applications transactionnelles présente d'importants avantages, qui se concrétisent par une modularité renforcée, plus de souplesse et un potentiel de concurrence accru (donc de meilleures performances). Cependant, l'approche objet accroît la complexité des systèmes transactionnels. Des nouveaux problèmes (qui ne sont pas abordés dans cette thèse) apparaissent dès l'instant où les opérations sur les classes (création, destruction et mise à jour) peuvent être invoquées en même temps que les opérations de consultation et de mise à jour des instances de classes (voir [8]).

I.3 Les modèles transactionnels

Le modèle des transactions plates est le modèle le plus simple et le plus répandu. Les transactions plates, comme leur nom l'indique, offrent aux applications un niveau de contrôle unique pour toutes les actions effectuées par une transaction : soit l'effet de l'ensemble des actions est rendu visible[(7)] (il y a *validation*), ou bien les actions composantes n'ont aucun effet dans le système (il y a *abandon* ou *rejet*).

Le fait qu'il n'y ait pas de contrôle au niveau des actions individuelles est une restriction qui peut être gênante, même pour une application aussi simple que la

(7) Nous considérons ici le cas où les transactions effectuent des opérations éléementaires de type lecture ou écriture.

réservation d'un voyage. Dans une telle application il est nécessaire de réserver des billets pour le transport entre les différents points du voyage, des chambres dans des hôtels, des voitures de location, etc. Les différentes réservations doivent répondre à des contraintes différentes, comme la disponibilité des places, les prix, etc. Considérons que chaque réservation correspond à l'une des actions de la transaction. Supposons que la réservation d'une chambre d'hôtel dans une ville donnée ne soit pas possible, alors le client voudrait annuler la réservation du train qui l'amène à cette ville et changer le trajet pour passer par une autre ville. Dans le modèle plat, ceci nécessite l'annulation de la transaction entière. Une nouvelle transaction doit être relancée, qui ré−exécute les réservations effectués jusqu'à ce point.

La définition des modèles transactionnels plus complexes est encore du niveau de la recherche. Les extensions apportées au modèle des transactions plates, appelé aussi modèle classique, répondent souvent à des besoins qui sont spécifiques à un domaine précis d'application. Afin de pouvoir décrire, comprendre et comparer différents modèles, des chercheurs ont abordé la formalisation des modèles transactionnels (voir ACTA [14], un formalisme pour la description des modèles transactionnels).

Tous les nouveaux modèles transactionnels cherchent à relâcher les contraintes imposées par les transactions. Un passage en revue succinct de quelques modèles transactionnels (voir [24] pour une présentation plus détaillée) nous permet d'avoir une image globale sur les techniques utilisées pour remédier aux inconvénients du modèle classique.

Les transactions avec points de reprise et les transactions enchaînées

L'objectif est de permettre la définition de points de contrôle intermédiaires, afin de permettre un retour en arrière partiel (et non pas total, comme dans le cas de l'abandon).

Les points de reprise sont des points où, à la demande du programme, l'état de la transaction est sauvegardé. Il s'agit d'une sauvegarde non−persistante des données utilisés par la transaction. Le programme peut ensuite demander le retour à un point de reprise donné. Cependant, en cas de panne, la transaction est annulée.

Dans le cas des transactions enchaînées, les points de contrôle sont des points de validation du travail déjà effectué par la transaction (on ne peut plus annuler les effets ainsi validés, mais on peut revenir en arrière au dernier point de validation). Cependant, il s'agit d'une validation spécifique, dans le sens où l'application continue à se dérouler dans le même contexte transactionnel qui est donc préservé, et que la validation est suivie par le début (i.e. le lancement) du morceau de transaction suivant. Les opérations de validation et l'opération de lancement qui la suit

sont atomiques (i.e. une autre transaction ne peut pas voir les effets partiels, même validés, de la transaction). L'avantage de ce modèle par rapport au précédent est que la transaction peut libérer, aux points de contrôle, des ressources qu'elle n'utilise plus.

Les transactions emboîtées.

Si dans le cas des modèles précédents une transaction peut être vue comme une séquence de sous−transactions, dans ce modèle, détaillé dans I.3.1, les sous−transactions sont organisées dans une structure hiérarchique. La validation d'une sous−transaction est relative : ses effets sont rendus visibles à la transaction mère et ne sont pas visibles de l'extérieur de la hiérarchie. Une sous−transaction peut être annulée, ce qui entraîne l'annulation de ses effets, ainsi que des effets de ses sous−transactions validées.

Les transactions multi−niveaux

Il s'agit d'une généralisation des transactions emboîtées. La caractéristique essentielle du modèle consiste en ce qu'une sous−transaction peut être validée dans le vrai sens du terme (et non pas seulement relativement à ses supérieures, comme dans le cas des transactions emboîtées), avant la validation de sa racine. Le modèle est basé sur l'hypothèse que les effets d'une sous−transaction validée peuvent être annulés par une *transaction compensatrice*, dans le cas où sa mère ou une de ses supérieures serait abandonnée.

Si le modèle des transactions emboîtées n'impose aucune restriction sur la programmation des sous−transactions (sur les traitements effectués ou sur les objets qu'elles manipulent), la mise en œuvre des transactions compensatrices nécessite une organisation hiérarchique du système. Celui−ci doit avoir une structure en couches, une couche étant composée par des objets et les opérations qui leurs sont associées, avec la restriction suivante : la mise en œuvre des objets d'une couche donnée n doit être réalisée uniquement à l'aide d'opérations définies au niveau $n-1$. Ceci est une condition nécessaire au maintien des propriétés ACID au niveau des transactions racines.

Le modèle des transactions multi−niveaux présente beaucoup d'intérêt dans le cas où les transactions manipulent des objets ([9]). Grâce à la structuration en couches, il est possible d'exploiter la propriété de commutativité des opérations à chaque niveau. De plus, les opérations appliqués aux objets peuvent ne pas être indivisibles. Il est suffisant qu'elle soient atomiques pour assurer la correction des exécutions concurrentes. Par conséquent, l'utilisation des objets par les

transactions multi−niveaux permet d'accroître le parallélisme, donc le potentiel de performances du système.

D'autres modèles, tel que les *transactions emboîtées ouvertes*, les *transactions longues*, les *sagas* etc., renoncent aux propriétés ACID afin de minimiser les pertes dues aux pannes (nous avons vu que même dans le cas des transactions enchaînées, une transaction interrompue par une panne est entièrement annulée), ainsi que de minimiser le temps durant lequel les ressources sont bloquées par une transaction. La validation précoce des sous−transactions est un moyen permettant d'atteindre cet objectif. Cependant, à l'opposé des transactions multi−niveaux, l'atomicité des transactions n'est plus respectée par ces modèles.

I.3.1 Le modèle des transactions emboîtées

Ce modèle (défini par Moss [42]) permet de regrouper plusieurs transactions, afin d'obtenir une transaction composée.Un niveau supplémentaire de contrôle est ainsi introduit : une transaction composante, ou *sous−transaction*, peut être abandonnée sans que cela entraîne obligatoirement l'abandon de la transaction composée. Selon la sémantique de l'application, la sous−transaction peut être relancée, ou bien une sous−transaction différente peut être lancée à sa place.

La composition des transactions peut être récursive, une sous−transaction pouvant être composée d'autres sous−transactions. Les transactions ont une structure hiérarchique et peuvent être représentées sous forme d'arbre. Les feuilles de l'arbre sont des transactions plates.

Dans la proposition de Moss, les traitements (consultation et mise à jour de données) sont effectués uniquement par les transactions feuilles, les transactions mère ayant un rôle de contrôle. Cette restriction a été levée par des extensions au modèle qui sont présentées dans le chapitre IV.

Lorsque la composition des transactions est une facilité utilisable au niveau applicatif, les transactions deviennent un moyen de structurer les applications. Pour que la décomposition des applications en sous−transactions soit vraiment utile en pratique, il est nécessaire d'assurer un certain degré de coopération entre les composantes. Par exemple, il est nécessaire que les ressources affectées à une transaction soient accessibles à ses sous−transactions. Les règles de visibilité présentées ci−dessous décrivent plus précisément la coopération des composantes dans le modèle défini par Moss. Des extensions permettant d'augmenter le degré de coopération au sein d'une transaction emboîtée sont présentées en [29].

Une transaction peut être vue comme une famille de transactions. Les transactions qui ont des sous−transactions sont appelées *mères*, et leurs

sous−transactions sont appelées *filles*. Les sous−transactions qui se situent au même niveau d'un arbre sont appelées *sœurs*. Les termes *ancêtre* et *descendants* sont aussi utilisés pour désigner les transactions se situant sur le même chemin d'accès dans un arbre. La relation d'ascendance (ou descendance) est réflexive. Par exemple, dans la Fig. 1.2 les ancêtres de la transaction T1 sont T et T1 ; les descendants de T1 sont T3 et T1. La *racine* de l'arbre est l'ancêtre de toutes les transactions de l'arbre. Les termes *supérieur* et *inférieur* sont les formes non réflexives des termes ancêtre et descendant. Par exemple, dans la Fig. 1.2 la transaction T1 a un seul supérieur, la transaction T, qui est en même temps la racine de l'arbre. La transaction T3 n'a pas d'inférieurs, mais elle a deux supérieurs : T1 et T.

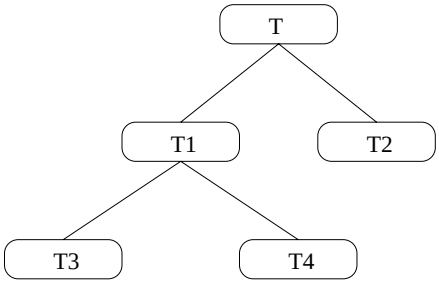

Fig. 1.2 : L'arbre représentant une transaction emboîtée

La mise en œuvre des transactions emboîtées nécessite des mécanismes plus complexes, mais apportent en contrepartie un plus de modularité, une amélioration dans le traitement des erreurs et des pannes, car celles−ci peuvent être confinées au niveau des sous−transactions atteintes, ainsi qu'un gain de performance lorsque des sous−transactions sont exécutées en parallèle.

Les caractéristiques des transactions emboîtées

Les sous−transactions sont exécutées en isolation et leur atomicité est relative :

- Une sous−transaction est isolée par rapport à d'autres sous−transactions sœurs (elles peuvent donc s'exécuter en parallèle), mais aussi par rapport à l'extérieur de sa hiérarchie transactionnelle. L'isolation des sous−transactions est assurée selon les règles suivantes :

  Les règles de visibilité. Les objets (ou ressources) d'une transaction sont accessibles à ses sous−transactions. Les modifications effectuées par

une sous−transaction deviennent visibles pour sa mère lors de la validation de la sous−transaction, mais elles ne deviennent pas visibles à l'extérieur de la famille car elles sont conditionnées par la validation de ses supérieurs (voir la règle de validation ci−dessous). Les modifications effectuées par une sous−transaction ne sont pas visibles à ses sœurs dans le cas où elles s'exécutent en parallèle.

• En ce qui concerne l'atomicité, si une sous−transaction est abandonnée, ses effets doivent être entièrement annulés et ses ressources libérées. Si elle est composée, alors ses filles doivent être aussi annulées, même dans le cas où celles−ci ont été validées. Ceci est nécessaire, car les filles ont pu utiliser les ressources de leur mère (selon les règles de visibilité). Il s'ensuit que la validation de toute sous−transaction est dépendante de la validation de sa mère, et, récursivement, de la validation de tous ses ancêtres jusqu'à la racine.

Les règles de validation. Les ressources ne sont pas libérées lors de la validation d'une sous−transaction, car sa validation finale n'est pas certaine (il s'agit d'une validation conditionnelle). Les résultats sont rendus accessibles à sa mère. La sous−transaction sera validée et ses résultats rendus visibles au monde extérieur seulement après la validation de la racine.

Les règles d'annulation. L'abandon d'une (sous−)transaction T implique l'annulation de ses effets, (annulation des modifications et libération des ressources), ainsi que l'abandon de toutes ses sous−transactions. Il s'agit d'un processus récursif qui entraîne l'abandon de l'arbre transactionnel dont T est la racine.

I.4 Intégration des concepts d'objet atomique et de transaction dans les systèmes répartis

Des exemples de systèmes répartis qui utilisent les transactions pour assurer la fiabilité des données, mais qui n'utilisent pas la sémantique des objets (les applications effectuant des opérations de lecture et écriture sur des fichiers, ou des enregistrement stockés dans des BD relationnelles) sont : Camelot [19], Locus (un système de fichiers répartis intégrant un mécanisme transactionnel qui étend le modèle de Moss pour permettre plus de coopération entre les composantes d'une transaction emboîtée) [50] ou QuickSilver [30].

Le système réparti Camelot est présenté dans la suite en même temps que l'environnement de programmation Avalon [17][31] qui, implanté au−dessus de

Camelot, a permis le développement des applications transactionnelles à base d'objets atomiques.

Nous avons plus particulièrement étudié les systèmes Argus [38], Arjuna [48] et Clouds [16] qui ont comme caractéristiques communes la répartition et l'intégration des transactions emboîtées et des objets. Ces trois systèmes visent le développement d'applications réparties fiables.

I.4.1 Argus

Projet de recherche mené au MIT de 1983 à 1987, son objectif était de concevoir à la fois un système et un langage pour la programmation d'applications réparties. Le système fournit un support pour l'exécution des programmes répartis (exécution sur un ensemble de sites reliés par un réseau local). Le langage fournit des abstractions permettant la construction de programmes répartis modulaires (les modules étant appelés *guardians*), dans lesquels la distribution des données est cachée, et leur cohérence est préservée grâce à l'intégration des *actions atomiques* (i.e des transactions). Les *guardians* encapsulent des ressources (i.e. des objets) qui peuvent être utilisées uniquement à travers de procédures spéciales appelées *handlers*. L'exécution d'une telle procédure se fait en tant que transaction. Étant donné que les appels peuvent être emboîtés, le système fournit des mécanismes pour la mise en œuvre des transactions emboîtées. Le modèle transactionnel fourni est celui défini par Moss, avec une extension permettant à une transaction de créer une transaction racine indépendante. Une telle transaction peut être validée indépendamment du mode de terminaison de sa mère. Une autre différence par rapport au modèle de Moss est que les transactions mères peuvent manipuler des objets. Cependant, il n'y a pas de support pour le parallélisme mère/fille (une transaction ne pouvant pas s'exécuter en parallèle avec ses ancêtres, ce qui est garantit par la suspension d'une mère pendant l'exécution de ses filles).

Revenons plus en détail sur la construction d'applications dans Argus. Une application est mise en œuvre par un ou plusieurs modules. Chaque module encapsule des objets et des processus. Les processus peuvent partager des objets dans leur module, mais le partage d'objets entre modules n'est pas possible ; l'utilisation d'un objet se trouvant dans un autre module se fait par l'appel à un traitant *(handler*).

Chaque module réside entièrement sur un site, mais plusieurs modules peuvent coexister sur un site. Les modules sont résistants aux pannes de site : après une telle panne, les objets constituant l'état stable de tout module résidant sur le site, sont restaurés sur la base des informations gérées dans une mémoire stable.

Argus fournit aussi des *actions* (i.e. des transactions) et des *types de données atomiques* prédéfinis, tel que les tableaux et les enregistrements atomiques. La

synchronisation des objets de ces types est assurée par un protocole de verrouillage à deux phases basé sur la classification des opérations en lectures et écritures. Les modifications des objets atomiques sont effectuées sur des copies. Les copies remplacent l'image stable des objets si la transaction qui les utilise est validée (un protocole de validation à deux phases assure l'atomicité des mises à jour effectuées aux niveaux des différents modules). Argus permet la définition de nouveaux types atomiques en tant que combinaison de types non−atomiques et de types atomiques prédéfinis.

I.4.2 Camelot/Avalon

Camelot est un système transactionnel conçu et mis en œuvre à CMU. Son importance réside dans le fait qu'il s'agit de la première implantation du modèle des transactions emboîtées, de manière à rendre ce concept directement utilisable pour le développement d'applications transactionnelles réparties.

Les facilités offertes par le système Camelot sont accessibles depuis des programmes écrits en C, C++ et Lisp, à travers une bibliothèque de procédures qui permet la mise en œuvre des applications et des serveurs de données. Typiquement, une application est structurée sous la forme : début de transaction (par appel à la primitive *Begin_Transaction*), appels à des serveurs de données pour différentes opérations, et fin de transaction (par la primitive *End_Transaction*). Les serveurs de données peuvent être construits sous la forme d'applications transactionnelles.

Les différents serveurs de données voient Camelot comme une couche logicielle construite au dessus du noyau Mach [1]. Cette couche (existante au niveau de chaque site) est composée de processus correspondant aux composantes d'un service transactionnel classique [24], tels que : le gérant transactionnel − qui assure le consensus des participants d'une transaction répartie sur le mode de terminaison de la transaction, le gérant de reprise − responsable de la restauration d'un état stable après une panne de site, le gérant de stockage − responsable de la gestion du cache et du journal, le gérant des communications − qui assure l'extension transparente des applications sur plusieurs sites, et qui gère les informations de répartition requises par les gérants transactionnels.

Le langage Avalon [31] apporte des extensions à des langages tel que C++, Lisp et Ada, dans le but de faciliter la programmation des applications transactionnelles. Les extensions sont construites au−dessus de Camelot. Nous présentons dans la suite quelques caractéristiques d'Avalon/C++, extension du langage C++.

Le modèle de programmation est similaire avec celui de Argus, les *guardians* étant remplacés par des serveurs qui encapsulent des objets et qui exportent des

opérations et des constructeurs. De même qu'en Argus, les objets peuvent être volatils ou persistants (l'état de ces derniers étant récupéré après une panne). Par ailleurs, Avalon fournit des types atomiques prédéfinis et des outils permettant la constructions de nouveaux types atomiques.

Cependant, le comportement des objets atomiques définis en Avalon/C++ correspond au critère d'*atomicité hybride* [53]. L'atomicité hybride est maintenue par un protocole de verrouillage à deux phases, mais la concurrence peut être accrue en prenant en compte l'ordre de sérialisation lors du traitement des conflits. Ceci est possible grâce au type prédéfini *tid* qui fournit aux programmeurs des opérations permettant de déterminer l'ordre de sérialisation des transactions.

Une autre caractéristique d'Avalon/C++ est le fait que les programmeurs peuvent définir pour les objets atomiques des procédures spécifiques de validation et d'abandon. Pour cela, Avalon/C++ fournit la classe *subatomic* ayant deux opérateurs publics virtuels *commit* et *abort*, qui peuvent être surchargés pour la définition des opérations spécifiques [17]. Les nouvelles classes d'objets atomiques peuvent être dérivées soit de la classe *subatomic*, soit, dans le cas où les mécanismes de validation et d'abandon fournis par défaut par Camelot sont suffisants, de la classe *atomic*.

I.4.3 Arjuna

De même qu'Argus, le système Arjuna a été conçu pour le développement et l'exécution d'applications réparties robustes (résistantes aux pannes). Arjuna ne propose pas un nouveau langage de programmation, les applications sont écrites dans le langage C++. En outre, le système Arjuna est mis en œuvre par une hiérarchie de classes C++.

Arjuna offre aux programmeur la possibilité de structurer les applications à l'aide de transactions emboîtées. Les applications sont constituées par des appels d'opérations appliquées sur des objets complexes (instances des classes C++). Les objets Arjuna sont persistants (la mise en œuvre actuelle utilise le système de fichiers Unix[(8)]) ; lorsqu'ils sont utilisés au sein de transactions, leur cohérence est maintenue même en présence de pannes de site ou de communication.

L'originalité d'Arjuna est d'avoir réalisé le support pour la programmation d'applications réparties et robustes, sous la forme d'une hiérarchie de classes, dans un langage de programmation à objets très répandu. Les programmeurs d'applications peuvent définir de nouvelles classes qui héritent des mécanismes fournis par les classes prédéfinies, en plaçant ces classes dans la hiérarchie (Fig. 1.3).

---

(8) Unix est une marque déposée par AT&T

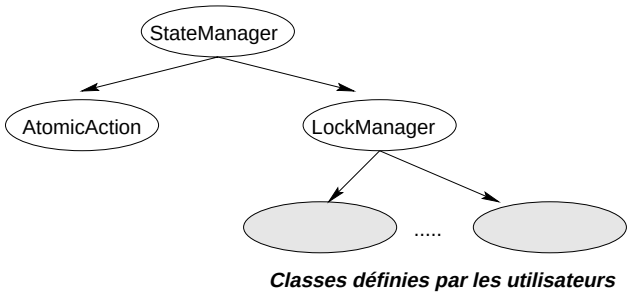

Fig. 1.3 : Vue simplifiée de la hiérarchie de classes de Arjuna

La classe *StateManager* fournit les mécanismes nécessaire à la création des objets persistants. Les objets persistants sont gardés dans la mémoire de stockage (*object store*). Ici les objets sont *passifs*, mais ils deviennent *actifs* lorsqu'ils sont utilisés par un programme. Les passages d'un état à l'autre se font par l'utilisation des opération fournies par la classe *StateManager*, opérations qui doivent être redéfinies dans chaque classe utilisateur afin de prendre en compte la structure spécifique des objets. L'activation d'un objet entraîne la création d'un processus serveur qui lui est associé et le chargement de l'état de l'objet dans l'espace d'adressage du serveur. La classe *StateManager* fournit également des mécanismes pour la mise en œuvre des transactions. Par exemple, l'opération de désactivation sert pour la sauvegarde de l'état de l'objet lors de la validation de la transaction racine qui l'utilise. Par ailleurs, cette classe fournit un mécanisme permettant la copie (en mémoire volatile) de l'état initial d'un objet pour le compte de la transaction qui le modifie, afin de pouvoir restaurer cet état en cas d'abandon de la transaction. Ce mécanisme est nécessaire pour la mise en œuvre des transactions emboîtées.

La classe *LockManager* fournit les mécanismes nécessaires au contrôle de la concurrence (elle met en œuvre un protocole de verrouillage à deux phases). Par conséquent, les classes utilisateur dérivés de cette classe, héritent de tous les mécanismes nécessaires pour la création des objets atomiques (le mécanisme de synchronisation est fourni par la classe *LockManager*, les mécanismes nécessaires à l'atomicité et à la permanence étant fournis par la classe *StateManager*).

La classe *AtomicAction* fournit les mécanismes pour la création et l'utilisation des transactions. Les transactions sont donc des objets, instances de la classe *AtomicAction*. Les programmes peuvent utiliser les transactions en créant des instances de la classe *AtomicAction*, et en invoquant les opérations *Begin*, *End* ou *Abort* (avec les significations habituelles) sur ces instances.

En comparaison avec Arjuna, l'environnement de programmation d'Argus offre plus de sécurité. Les classes d'Arjuna sont en effet faciles à utiliser mais laissent

plus de possibilités pour effectuer des erreurs de programmation. Un autre aspect positif d'Arjuna est que sa hiérarchie de classes est mieux adaptée pour être utilisée comme interface d'accès à une base de données orientée−objet que les modules d'Argus, qui sont actifs en permanence.

I.4.4 Clouds

Clouds est un système d'exploitation réparti développé à Georgia Institute of Tehnology. Clouds met en œuvre un modèle d'exécution à base d'objets et de processus légères − "object−thread model".

Les objets de Clouds sont des entités passives, de grosse taille, qui encapsulent du code et des données dans un espace d'adressage qui leur est propre. De plus, les objets sont persistants, nommés par une référence unique, et leur localisation est cachée à l'utilisateur. En Clouds, on ne retrouve pas le concept d'objet atomique tel qu'il est employé dans les systèmes déjà présentés, mais les programmeurs d'objets atomiques peuvent spécifier différentes propriétés de cohérence pour les opérations qui manipulent les objets.

Les entités actives sont les processus qui exécutent des appels de méthode au sein des objets. Par le jeu des appels imbriqués, les processus passent d'un objet à l'autre et peuvent arriver sur des machines différentes. Plusieurs processus peuvent s'exécuter en parallèle dans un objet et un objet peut être partagé par des processus qui s'exécutent sur des machines différentes grâce à un mécanisme de mémoire virtuelle répartie.

En dehors de l'intérêt présenté par Clouds du point de vue des mécanismes système qui y sont mis en œuvre (voir [15] pour une comparaison entre le mécanisme d'invocation d'objets de Clouds et d'autres approches adoptés dans différents systèmes répartis), nous nous sommes intéressés à Clouds pour l'originalité des concepts mis en œuvre pour le support de la fiabilité. L'objectif principal est la souplesse, afin de pouvoir exploiter la puissance du modèle d'exécution et de permettre la prise en compte des besoins des applications en termes de cohérence. Un deuxième objectif est d'éviter de recourir à l'assistance des programmeurs d'application pour la mise en œuvre des mécanismes spécifiques de synchronisation et de récupération après panne (donc, traitement automatique).

La souplesse est obtenue grâce à la possibilité d'avoir des processus caractérisés par différents degrés de cohérence : processus qui préservent la cohérence globale (*gcp*), c.a.d. la cohérence d'un ensemble d'objets dont le traitement est exécuté par le processus ; processus qui préservent la cohérence locale (*lcp*), c.a.d. que la cohérence des objets individuels est garantit, mais il n'y a pas de

contrôle inter−objet, et processus standard (*s*) (i.e. pas de cohérence). Le système assure automatiquement le degré de cohérence requis lors de l'exécution des différents types de processus. Lors de leur création, les processus sont standard. Par ailleurs, les opérations (les points d'entrée dans les objets) sont étiquetées, de manière statique, par : S (standard), LCP ou GCP, avec les même significations que pour les processus. Un processus peut se transformer (changer de type) lorsqu'il entre dans un objet, selon l'étiquette associée à l'entrée. Par exemple un processus standard se transforme en un *lcp* suite à l'invocation d'une opération étiquetée LCP. L'opération sera alors exécutée en isolation et ses effets seront validés ou annulés lors de sa terminaison. Si le même processus invoque ensuite une autre opération étiquetée LCP, le même scénario se répète ; il n'y a donc pas d'atomicité préservée pour l'ensemble des deux opérations successives (voir [13] pour plus de détails).

Les combinaisons des différents degrés de cohérence peuvent donner des exécutions avec des sémantiques très variées (à commencer par des transactions plates ou emboîtées, et jusqu'à la possibilité d'avoir un processus standard qui modifie des données lues ou écrites par un processus préservant la cohérence). Cependant, l'interaction des processus à l'intérieur des objets est contrôlable grâce aux étiquettes de cohérence.

En conclusion, Clouds permet des exécutions robustes et non−robustes et offre aux concepteurs d'objets les moyens pour atteindre le degré de cohérence requis par la sémantique des opérations qui manipulent les objets.

I.5 Conclusions

L'utilisation des transactions permet de simplifier la programmation des applications réparties, car le concepteur d'applications peut ignorer les problèmes dus aux accès concurrents et aux pannes. Ces problèmes sont automatiquement résolus par le système grâce aux mécanismes de contrôle de concurrence et aux mécanismes permettant de défaire ou de refaire les effets des transactions.

La contrepartie de la simplicité est un manque de souplesse. La simplicité du modèle peut être un inconvénient dans la mesure où les applications ont une structure complexe, avec des composantes qui coopèrent pour la réalisation d'une tâche. D'autre part, le manque de souplesse peut avoir des implications sur les performances des applications.

L'étude menée nous a permis d'identifier deux directions de recherche dont l'objectif est de répondre au problème du manque de souplesse des transactions.

• La première est un travail sur les modèles transactionnels. Parmi les modèles proposés, nous avons tout particulièrement étudié le modèle des transactions emboîtées. Ce modèle a eu beaucoup de succès, principalement en raison du fait qu'il améliore le traitement des pannes et qu'il permet des gains de performance grâce au parallélisme intra−transactionnel. De plus, si l'utilisation des transactions est sous le contrôle des applications, les transactions emboîtées deviennent un outil pratique pour la structuration des applications.

• La deuxième direction explore la piste des objets atomiques. Les transactions qui utilisent des objets atomiques sont plus souples, car les mécanismes de synchronisation et de récupérations après pannes sont décentralisés au niveau de chaque objet. Ceci permet d'adapter les mécanismes aux besoins spécifiques des objets.

Dans la suite nous nous intéressons à chacune de ces deux directions. L'étude approfondie du modèle des transactions emboîtées nous permet de le rendre plus souple, en levant plusieurs des restrictions qu'il impose. D'autre part, l'adoption des objets atomiques, motivée principalement par la décentralisation des mécanismes, permet d'obtenir encore plus de souplesse au niveau du modèle, mais aussi de simplifier la mise en œuvre des transactions par un allégement des mécanismes devant être implantés dans les couches basses du système.

Par ailleurs, l'étude des systèmes présentés ci−dessus nous a permis de tirer les enseignements suivants :

Arjuna et Avalon/C++ présentent un grand intérêt pour la conception d'un service transactionnel au dessus d'une plate−forme à objets persistants. L'approche qui consiste à construire des classes spécialisées mettant en œuvre les mécanismes de synchronisation et de récupération après panne peut être facilement adoptée dans le cas où la plate−forme permet l'exécution des applications écrites dans un langage de programmation à objets.

La hiérarchie de classes fournie par Arjuna nous a semblé particulièrement intéressante par son approche "tout objet" (par exemple, les transactions et les verrous sont mis en œuvre par des objets). Ce schéma est profitable pour la souplesse, car, d'un part, cela fournit à l'utilisateur la possibilité d'avoir le contrôle sur l'utilisation des transactions (ce qui est un pré−requis pour que le transactions emboîtées deviennent un outil pour structurer les programmes), et, d'autre part, cela permet la mise en œuvre de différents types de verrous.

Les transactions en Arjuna sont conformes au modèle des transactions emboîtées défini par Moss. La recherche d'une souplesse accrue nous a conduit à étudier la

conception de Clouds. Nous avons ainsi retenu l'idée de fournir le choix entre plusieurs degrés de cohérence. Cependant, au contraire de Clouds, nous avons décidé de garantir les propriétés ACID des transactions.

Chapitre II

Guide

II.1 Introduction

Guide est un projet de recherche qui s'est déroulé à l'Unité Mixte Bull−IMAG de 1986 à 1994. L'objectif du projet était la réalisation d'une plate−forme système pour des objets persistants et répartis, et d'un environnement de programmation qui facilite l'écriture d'applications réparties structurées en termes d'objets.

Le projet a connu deux étapes [27]. La première étape a été celle du prototype appelé Guide−1 construit au dessus de Unix. La définition du langage de programmation à objets Guide [4] et l'écriture de son compilateur ont également eu lieu dans cette étape. La deuxième version du prototype, appelé Guide−2, a été réalisé au dessus du micro−noyau Mach 3.0, pendant la deuxième étape du projet, commencée en 1990. Cette deuxième étape a eu comme objectif de rendre plus générique la couche logicielle qui permet la création et l'utilisation des objets persistants et répartis. Elle devait ainsi permettre l'exécution d'applications écrites non seulement dans le langage Guide, mais aussi dans d'autres langages largement utilisés tel que C++. En outre, la nouvelle architecture du système a permis une amélioration des performances, ainsi que plus de sécurité et de protection lors de l'accès aux objets.

Le Service Transactionnel Guide a été intégré dans le prototype Guide−2. Les classes spécialisées permettant l'utilisation des transactions et des objets atomiques sont écrites dans le langage Guide.

Ce chapitre présente le modèle de programmation et le langage Guide, ainsi que le modèle d'exécution des programmes Guide, et l'architecture de la plate−forme.

II.2 Le modèle de programmation

Le modèle de programmation à base d'objets défini par le projet Guide est intégré dans le langage de programmation du même nom. Les principales caractéristiques du langage sont : la séparation entre types et classes, l'héritage simple et conforme, la

répartition cachée, le contrôle explicite de la concurrence d'accès aux objets au moyen de conditions de synchronisation, et le traitement des exceptions.

L'unité de programmation Guide est l'objet : entité qui encapsule les données constituant son état, et un ensemble d'opérations, ou de méthodes, qui manipulent l'état. Les objets sont typés (sont des instances d'un type), identifiés par des références uniques au niveau du système, et composés (un objet peut contenir des références à d'autres objets).

Un programme (ou application) est décrit en termes de *types* (les descripteurs d'interface), de *classes* (les réalisations des types), et de *variables* qui encapsulent des références et qui servent à l'appel de méthodes sur des objets.

Dans la suite, nous présentons quelques unes des caractéristiques du langage.

II.2.1 Types, classes, héritage

Tout objet a une classe qui met en œuvre un type. Le type, qui est un descripteur d'interface, contient les signatures des opérations applicables à l'objet et les variables d'état visibles. La classe, qui est une réalisation particulière d'un type, définit la représentation de l'état de l'objet (le type des variables privées) et le code des opérations.

Exemple. Définition du type `Boîte` et de la classe `Une_boîte`, une mise en œuvre possible du type `Boîte` :

```
TYPE Boite IS
    // variables d'état visibles
    Nom : String [80];
    Propriétaire : Ref Personne;
    // signature méthodes
    METHOD Déposer (IN Ref Élément);//ajout d'élément
                                    dans la boîte
    METHOD Prendre (OUT Ref Élément);//retrait d'un
                                    élément de la boîte
END Boite.


CLASS Une_boîte
IMPLEMENTS Boîte IS
    // variable d'état
    CONST taille : Integer = 15 ;
    nb_elt, dernier, premier : Integer ;
    Contenu : Array [taille] OF Ref Élément ;
    // code des méthodes
    METHO Déposer (IN elt : Ref Élément);
    BEGIN
      ...
```

```
    END Mettre;

    METHOD Prendre (OUT elt : Ref Élément);
    BEGIN
      ...
    END Prendre;
END Une_boîte.
```

Les variables d'état (visibles ou privées) sont typées et le type peut être : un type élémentaire (`Integer`, `Char`, `Boolean`), un type structuré (`Array`, `String`, `Record`), ou une référence sur un type construit (`Ref` aType).

Les constructeurs d'objets du système Guide sont les classes. À chaque classe est associée une opération de génération qui sert à la création des objets, instances de la classe. Une classe est elle−même un objet ; l'opération de génération d'objets implantée par la méthode New est définie sur les objets de type "classe" dont héritent tous les autres types.

Exemple. Création d'un objet de type Boîte, et appel de méthode sur l'objet :

```
e : Ref Élément; /* variables */
b : Ref Boîte;
........
b := Une_boîte.New;/* création objet b */
........
b.Déposer(e); /* appel de méthode appliqué à b */
```

Tous les objets d'un même type ont un comportement commun. Grâce à la notion de sous−type, il est possible de spécialiser certains comportements. La relation de sous−typage conduit à une organisation hiérarchique des types. Une hiérarchie analogue existe entre les classes.

Par exemple, le sous−typage permet de spécifier le fait qu'une boîte à lettres est une boîte. Les objets de type `Boîte_à_lettres` héritent de toutes les propriétés du type `Boîte` ; de plus, d'autres propriétés peuvent être définies ou des propriétés du super−type peuvent être modifiés par surcharge.

```
TYPE Boîte_à_lettres SUBTYPE OF Boîte IS
    Code : Integer;
    // propriété supplémentaire
    METHOD Concat (IN Ref Boîte_à_lettres) : Ref
    Boîte_à_lettres;
    // propriétés surchargées
    METHOD Déposer (IN Ref Lettre);
    METHOD Prendre (OUT Ref Lettre);
END Boîte_à_lettres.
```

Par définition, un sous−type est conforme à son super−type. Voir [4] pour plus de détailles sur la conformité. Dans notre exemple, la règle de conformité est respectée si le type `Élément` est un sous−type du type `Lettre`.

II.2.2 Concurrence et synchronisation

Le parallélisme peut être programmé de manière explicite avec l'instruction :

```
CO_BEGIN
    <ident> : <appel de méthode> ;
    ...
CO_END <condition terminaison>;
```

Plusieurs activités sont lancées en parallèle pour exécuter les appels de méthode spécifiés, l'activité mère étant suspendue pendant ce temps. L'activité mère est réactivée lorsque l'expression booléenne constituant la condition de terminaison de l'instruction devient vraie. Les activités filles qui sont alors encore actives sont automatiquement détruites. L'expression booléenne est une conjonction de variables booléennes `<ident>` associées aux activités filles. La valeur d'une telle variable est indéterminée tant que l'activité est en cours d'exécution, et elle devient vraie dès que l'activité se termine.

La synchronisation des activités concurrentes qui partagent un objet peut être spécifiée en plaçant une clause de contrôle à la fin de la définition de sa classe :

```
CONTROL
    <nom méthode> : <condition d'activation> ;
    ...
```

Une `<condition d'activation>` est une expression booléenne fonction de l'état des objets, des paramètres d'entrée de la méthode et des compteurs de synchronisation associés à chaque méthode. Pour chaque méthode m, les compteurs suivants sont définis :

```
invoked(m) : nombre total d'appels de la méthode m,
started(m) : nombre d'appels de la méthode m qui ont
             été acceptés,
completed(m): nombre d'exécutions terminées,
pending(m) : nombre d'activités qui sont en attente
              d'exécution de la méthode m,
current(m) : nombre d'appels en cours d'exécution.
```

Pour les conditions les plus fréquemment rencontrées dans les programmes, un certain nombre de mots−clés les représentant sont offerts au programmeur. Par

exemple, EXCLUSIVE(m1, m2, ..mk), désigne la condition d'exclusion mutuelle entre les méthodes mi, c'est à dire : Σcurrent(mi) = 0.

II.3 Le support d'exécution

Dans la suite nous présentons les concepts de base permettant l'exécution des méthodes sur des objets pouvant être partagés et répartis sur les différents sites constituant un système Guide. Un bref regard sur l'architecture du système permet d'identifier les composantes qui mettent en œuvre les concepts présentés.

II.3.1 Les concepts de base

Une application en cours d'exécution se compose d'un ensemble de flots d'exécution appelés **activités**, qui effectuent des appels de méthode sur des objets. Les activités et les objets font partie d'un espace d'adressage multi−sites appelé **domaine**, qui met en œuvre l'application. Pour qu'une activité puisse utiliser un objet, celui−ci doit être couplé dans le domaine (il y a une mise en correspondance entre l'objet et une zone de mémoire virtuelle contiguë). Si une application effectue un appel de méthode sur un objet O qui se trouve déjà couplé sur un site distant, alors le domaine qui met en œuvre l'application s'étend sur le site distant et O est couplé à l'extension du domaine. Étant donné qu'il était déjà couplé dans un domaine, l'objet O devient partagé par plusieurs domaines.

L'étude des applications réalisées en Guide ont montré que la plupart des objets utilisés sont de petite taille (moins de 300 octets), ce qui justifie leur regroupement dans des entités de plus grande taille appelées **grappes**. L'unité de couplage dans l'espace d'adressage d'un domaine est en fait la grappe, qui est donc aussi l'unité de partage entre domaines. Ainsi, deux domaines partageant un objet partagent la grappe qui le contient.

La persistance des objets est implicite : ils survivent à la mort du domaine qui les a créés. Afin qu'ils puissent résister aux pannes de site, les objets sont recopiés dans une mémoire permanente composée d'un ensemble de disques. Ceci conduit à un modèle de mémoire à deux niveaux :

− La mémoire de stockage, composée de **volumes** contenant des grappes. Chaque volume est géré par un **service de stockage** (voir Fig. 2.1). Guide a plusieurs services de stockage, chacun pouvant être réparti [10].

− La mémoire d'exécution, composé de l'ensemble des mémoires volatiles des machines et des zones disque gérées par des mécanismes de pagination (l'espace de pagination est donc séparé de celui de stockage).

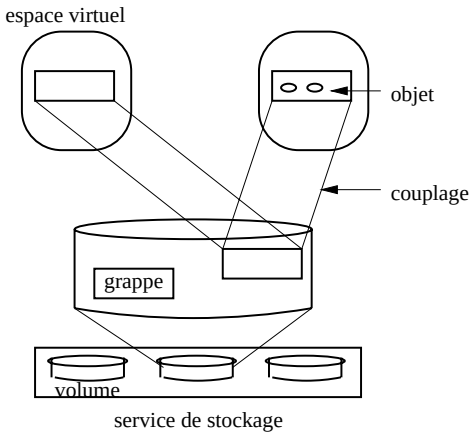

Fig. 2.1 : Organisation de l'espace de stockage

Avant toute utilisation, un objet doit être transféré de la mémoire de stockage vers la mémoire d'exécution. Pour des raisons d'efficacité, le transfert porte non pas sur un objet, mais sur sa grappe. Cependant, le chargement en mémoire du contenu de ces grappes à partir de l'espace de stockage est réalisé à la demande, par des fautes de page. Donc, l'unité de transfert entre la mémoire de stockage et la mémoire d'exécution est la *page*. L'opération de **couplage** permet de rendre accessibles les grappes aux applications ; cette opération associe une portion d'espace d'adressage virtuel à la zone de mémoire réservée pour une grappe.

Lorsqu'une grappe couplée dans l'espace virtuel n'est plus utilisée (tous les domaines qui ont couplé la grappe se sont terminés) la grappe est découplée et sauvegardée sur disque de manière atomique. De plus, le système permet la sauvegarde d'une grappe sur demande explicite.

Le partage des grappes pose le problème suivant : soit O1 et O2 deux objets dans une grappe G. Supposons que G est partagée par deux domaines D1 et D2, D1 utilisant O1 et D2 utilisant O2 (il n'y a donc aucune relation de dépendance entre D1 et D2, le partage de G étant un *faux partage*). Lorsqu'un des domaines se termine, soit D1, G reste couplée en mémoire d'exécution car elle est aussi couplée par D2. Une panne de site peut donc mener à la perte des modifications effectuées par D1 sur O1. Une solution serait de demander explicitement la sauvegarde de G lors de la terminaison de D1. Cependant, ceci peut avoir comme effet de bord la sauvegarde de O2 dans un état incohérent (à ce moment D2 est en cours d'exécution). Si D2 n'arrive pas à se terminer proprement, O2 reste sur le disque dans un état

incohérent. Le problème du faux partage des grappes montre que la mise en œuvre des transactions nécessite des mécanismes de sauvegarde de grain plus fin, ç.à.d. au niveau de l'objet.

Revenons sur le partage des objets. Deux domaines qui partagent un objet O doivent être isolés de manière à ce qu'une erreur d'adressage qui a lieu dans un des domaines lors de l'utilisation de O, ne puisse pas provoquer une erreur dans l'autre domaine. L'isolation des domaines est assurée par le fait qu'il n'y a pas de partage de mémoire entre domaines. Chaque domaine comporte son propre espace de mémoire virtuel dans lequel on couple dynamiquement les objets (en fait on y couple les grappes contenant les objets). Les activités s'exécutent dans ces espaces de mémoire virtuels appelés **contextes** (voir Fig. 2.2). Un domaine comporte au moins un contexte sur chacun des sites qu'il a visité.

Un autre problème posé par le partage des objets est l'isolation entre objets. Pour qu'une erreur qui survient dans un objet ne puisse pas entraîner la modification d'un autre objet, il faudrait disposer d'un espace virtuel par objet, ce qui ne serait acceptable que dans le cas de "gros" objets. Une solution de compromis est de coupler les objets appartenant à des propriétaires différents dans des espaces virtuels différents. Une erreur d'adressage ne peut ainsi affecter que des objets appartenant au même propriétaire. Par conséquent, un domaine peut comporter plusieurs contextes sur un même site, dans le cas où l'application utilise des objets appartenant à des propriétaires différents.

La Fig. 2.2 illustre le schéma de partage et d'isolation des objets, ainsi que l'isolation des domaines. Chacun des domaines D1 et D2 contient une seule activité qui s'exécute dans plusieurs contextes par le biais des représentants d'activités (chaque flot d'exécution étant réalisé par un processus léger appelé "thread" en Mach). Un représentant d'activité est créé lors du premier appel de méthode sur un objet qui doit être couplé dans un nouveau contexte. La création d'un nouveau contexte est due soit à un changement de propriétaire (l'objet invoqué appartient à un propriétaire différent du propriétaire de l'objet courant) soit à un changement de site (l'objet appelé se trouve dans la mémoire d'un site différent).

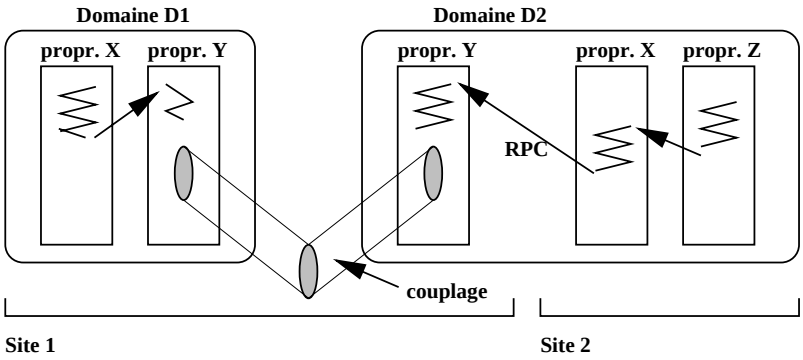

Fig. 2.2 : Isolation des domaines et des objets

Cette présentation est incomplète, car elle n'aborde pas des aspects fondamentaux du système, tel que la désignation et la localisation, la protection (les droits d'accès). Ces aspects, présentés en [12], ne sont pas indispensables pour la compréhension du travail effectué dans cette thèse.

II.3.2 Architecture du système

La machine virtuelle qui met en œuvre les concepts décrits ci−dessus fournit une interface qui donne accès à des services relatifs à la gestion des objets et à la gestion de l'exécution. Les composantes de la machine virtuelle (voir Fig. 2.3) sont brièvement décrites dans la suite.

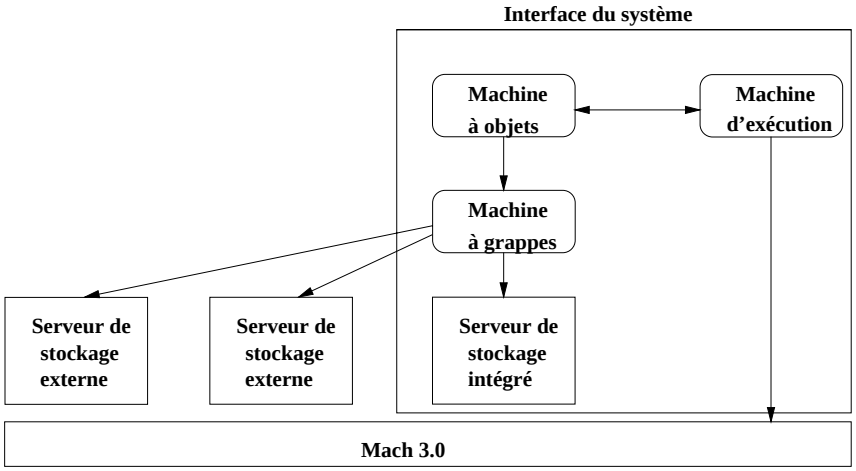

Fig. 2.3 : Architecture globale du système

La gestion des objets est réalisée par la **machine à objets**. Elle permet aux compilateurs de créer des classes, et offre une interface utilisée lors de l'exécution pour la création des instances et pour les appels à ces instances. La machine à objets et construite sur la **machine à grappes**, qui lui fournit des mécanismes de création,

destruction et couplage de grappes. Le stockage des grappes est assuré par un des services de stockage. Le serveur de stockage intégré est rapide, alors que d'autres serveurs externes garantissent une plus haute disponibilité. La **machine d'exécution** met en œuvre les domaines multi−contextes et les activités. Elle fournit aussi le mécanisme d'appel à distance.

II.4 Conclusion

Le système Guide fournit un langage et une plate−forme système pour le développement et l'exécution des applications réparties. Les applications sont structurées en terme d'objets, une application pouvant être représentée comme un arbre d'appels de méthodes sur des objets. Le modèle de programmation est intégré dans le langage Guide, qui est fortement typé et caractérisé par la séparation des types et des classes.

Les objets Guide sont persistants et ont une image stable sur disque. La persistance et le partage des objets créent les prémisses pour la coopération entre applications et au sein des applications. D'autres facilités pour la mise en œuvre des applications sont la transparence de la répartition et le parallélisme explicite. Des clauses de contrôle permettent la définition de conditions pour l'exécution des méthodes sur les objets d'une même classe. Ces clauses peuvent être utilisées pour la synchronisation des méthodes concurrentes au niveau des objets pris séparément.

Le modèle d'exécution du système Guide fournit les concepts nécessaires à la persistance et à la répartition transparente des objets, ainsi qu'au partage des objets. Les mécanismes sont rendus efficaces par le groupement des objets en grappes, qui sont les unités de partage et de persistance au niveau du système. Les grappes, stockées sur disque et gérées par des serveurs de stockage, sont rendues accessibles aux applications par couplage dans les espaces d'adressage virtuels que constituent les contextes. La mise en œuvre des contextes garantit l'isolation des applications qui partagent des grappes, mais aussi l'isolation des objets qui tout en étant utilisés par la même application ont des propriétaires différents. Une application Guide, représentée par un domaine, est composée d'un ensemble de contextes répartis sur les sites où l'application s'exécute. Les activités s'exécutent dans les différents contextes sur des objets qui sont couplés dans ces contextes.

Le système Guide a cependant des manques sur le plan de la cohérence qu'il garantit aux objets. D'un part, les mécanismes de synchronisation sont fournis seulement au niveau des objets et il n'est pas possible de synchroniser des activités relativement à un groupe d'objets. En particulier, il n'y a rien qui puisse permettre la sérialisation des activités qui partagent plusieurs objets.

Deuxièmement, le mécanisme qui assure la recopie des objets sur disque n'a pas un grain suffisament fin pour tenir en compte du faux partage dû aux grappes. En outre, il manque un mécanisme permettant de coordonner et de rendre atomique la sauvegarde des objets appartenant à des grappes couplées dans la mémoire des sites différents. La réalisation d'un tel mécanisme ne pose pas seulement le problème de l'extension des fonctions système déjà existantes ou du rajout des nouvelles fonctions système, mais aussi des questions sur les moyens qu'il faut fournir aux applications pour le contrôle de ces mécanismes (quels sont les outils qui doivent être fournis et comment les mettre en œuvre).

Chapitre III

Les caractéristiques générales du Service Transactionnel Guide

III.1 Introduction

Dans le premier chapitre de la thèse, nous nous sommes intéressés aux propriétés des transactions et des objets atomiques. Grâce à ces propriétés, les transactions et les objets atomiques peuvent être utilisés pour protéger les données contre les incohérences engendrées par les interférences d'actions concurrentes, ou par les différentes catégories de pannes.

Le Service Transactionnel Guide (STG) met en œuvre des moyens permettant aux applications Guide d'utiliser des transactions et des objets atomiques. La conception du STG a été guidée par le souci de satisfaire aux impératifs de facilité d'utilisation et de minimisation du surcoût imposé aux applications qui utilisent le service.

Pour faciliter l'utilisation des transactions et des objets atomiques, les moyens mis à la disposition des concepteurs d'applications doivent être adaptés à la programmation à base d'objets. Il est aussi souhaitable de minimiser les contraintes imposées aux applications qui utilisent le service transactionnel.

La mise en œuvre des transactions et des objets atomiques nécessite des mécanismes coûteux. Pour concilier la cohérence avec les performances, il est nécessaire de fournir aux programmeurs des moyens permettant d'exprimer les besoins des applications sur le plan de la cohérence.

Le STG remplit les conditions mentionnées ci−dessus, grâce notamment à sa *souplesse* et à son *approche de mise en œuvre à base d'objets*. Dans la suite de ce chapitre (section III.2) nous nous proposons de détailler ces caractéristiques et de présenter des exemples afin d'illustrer la manière dont la souplesse du STG permet la prise en compte des besoins des applications.

Un premier aperçu sur l'utilisation du Service Transactionnel Guide est fourni dans la section III.3.

La dernière partie (section III.4) traite des aspects liés au modèle transactionnel qui définit le fonctionnement du STG. Nous mettons en évidence les caractéristiques requises pour que le modèle assure la souplesse du STG.

III.2 Les caractéristiques

III.2.1 Un service transactionnel à base d'objets

Le STG comporte deux sortes de composantes : des classes spécialisés ainsi que des mécanismes implantés au niveau du noyau Guide (Fig. 3.1). Une partie des classes spécialisées constitue l'interface du service (les classes *Transaction*, *Atomic* et *Verrou*). Les classes spécialisées qui ne sont pas utilisées directement par les applications sont représentées par des boîtes en pointillés dans la Fig. 3.1. Les nouvelles primitives rajoutées au niveau système sont accessibles à travers les classes spécialisés.

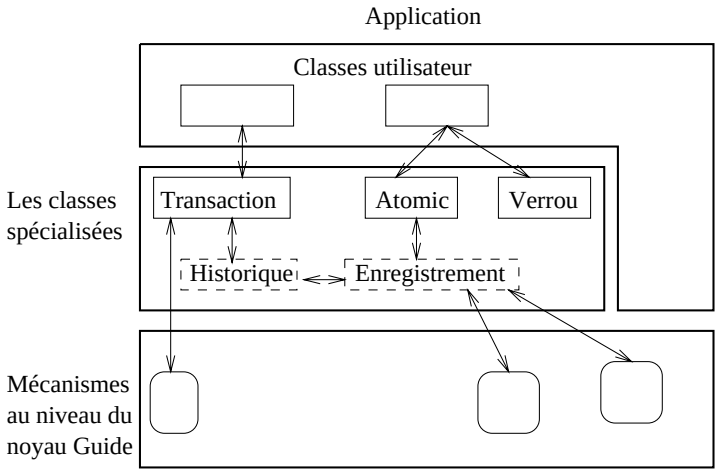

Fig. 3.1 : L'architecture du Service Transactionnel Guide

L'approche à base d'objets vise deux objectifs : la facilité d'utilisation du service et la facilité de sa mise en œuvre.

Facilité d'utilisation

L'utilisation des fonctions fournies par le service se fait au moyen des classes spécialisées, par l'instanciation des classes et par des appels de méthode sur les instances. En outre, des nouvelles classes peuvent être définies par la spécialisation des classes fournies.

Les entités spécifiques fournies par le service, telles que les *transactions* et les *verrous*, sont implantées par des objets spécialisés, instances des classes spécialisées. Ces objets sont créés et manipulés de la même manière que les objets normaux. Par conséquent, l'approche à base d'objets conserve l'uniformité des applications transactionnelles.

Facilité de mise en œuvre

Une partie considérable des fonctions fournies par le STG sont implantés par les classes spécialisées. La mise en œuvre de ces fonctions par des classes écrites dans le langage Guide présente l'avantage de pouvoir exploiter le support fourni par le système sous−jacent pour la manipulation des objets persistants et répartis : nommage, adressage, mécanismes d'invocation de méthode en local et à distance, transparence de la répartition, etc.

Un autre avantage de l'approche à base d'objets est le fait qu'il n'est pas nécessaire d'étendre le langage de programmation pour pouvoir utiliser le service. Nous n'avons donc pas eu à effectuer de modifications au niveau du compilateur Guide.

L'approche qui consiste à fournir aux applications des classes prédéfinies mettant en œuvre des mécanismes nécessaires aux traitements de type transactionnel a été adoptée dans le système Arjuna [48], mais aussi dans le langage de programmation Avalon/C++ [17]. Une comparaison entre la mise en en œuvre de cette approche dans Guide et dans les deux systèmes cités ci−dessus est présentée en VII.2.

III.2.2 La souplesse du service transactionnel Guide

À l'origine, les objets atomiques ont été conçus pour être utilisés conjointement avec les transactions. Plus précisément, les transactions n'utilisent que des objets atomiques et les objets atomiques sont utilisés uniquement à l'intérieur de transactions (voir par exemple Argus [39]).

Notre approche, détaillée dans les sous−sections suivantes, permet plus de souplesse ; on peut dire qu'il s'agit d'une "utilisation à la carte" des transactions et des objets atomiques. Premièrement, les transactions peuvent utiliser des objets non−atomiques (voir III.2.2.1). Cependant, le STG garantit les propriétés ACID seulement pour les transactions utilisant uniquement des objets atomiques. Deuxièmement, les objets atomiques peuvent être utilisés hors transactions (voir III.2.2.2). Dans ce cas de figure, le STG garantit que l'exécution d'une opération appelée sur un objet atomique se fait en isolation et de manière tout−ou−rien. Cependant, la permanence des effets de tels opérations n'est pas garantie en cas de panne. Finalement, le STG n'impose aucune contrainte dans l'utilisation des transactions dans la programmation d'applications (voir III.2.2.3).

III.2.2.1 L'utilisation des objets non−atomiques

On peut constater que les applications ont souvent besoin d'utiliser des objets non−atomiques ; il s'agit dans ce cas d'objets locaux où partagés dont l'utilisation ne met pas en cause la cohérence des traitements. Par exemple, les objets système (c.a.d. des objets qui fournissent l'interface de certains services système) peuvent ne pas être atomiques. Le contrôle de la concurrence pour l'accès à des objets de cette catégorie n'est pas assuré par le service transactionnel, mais par des mécanismes système ad−hoc (sémaphores, etc.). D'autre part, le retour en arrière est pour certains objets impossible, et pour d'autres n'est jamais nécessaire.

> Les objets Guide sont par défaut non−atomiques. Le STG fournit la classe spécialisée *Atomic* permettant la construction de classes d'objets atomiques (voir III.3.1.3). Les objets atomiques sont des instances de ces classes.
>
> Les programmeurs peuvent librement décider sur la catégorie des objets utilisés par une application.

III.2.2.2 L'utilisation des objets atomiques hors transactions

Il existe des situations permettant d'effectuer des traitements non−transactionnels en utilisant des objets atomiques : il s'agit des situations dans lesquelles l'atomicité des objets est suffisante pour assurer la sémantique des traitements. C'est le cas pour les traitements effectuant seulement des opérations de lecture (voir **Exemple 1**), mais quelquefois les traitements peuvent aussi effectuer des modifications (voir **Exemple 2**).

Exemple 1. Considérons une requête R soumise à un serveur WWW qui effectue le parcours en lecture d'un ensemble d'objets, par exemple afin d'afficher l'ensemble

des publications d'un auteur donné. Dans le cas où les objets composant l'ensemble sont atomiques, la requête R peut ne pas être exécutée en tant que transaction, car l'indivisibilité des opérations de lecture appliquées aux objets de l'ensemble est suffisante (il n'est pas nécessaire de garantir l'isolation du traitement composé par les différentes opérations de lecture). En effet, dans l'hypothèse où l'ensemble des publications présente la sémantique des ensembles faibles (*weak sets*, voir [55]), une autre requête concurrente peut modifier l'ensemble (insérer ou modifier un élément) avant que la requête R soit terminée, sans que cela soit grave.

Exemple 2. Considérons un objet atomique qui met en œuvre un cache de documents pour un serveur WWW. Son rôle est de garder localement des documents qui sont fréquemment utilisés. Un document qui ne se trouve pas dans le cache, y est rajouté par une opération de mise à jour lorsqu'on tente d'y accéder (opération *Insertion*). De temps en temps, le contenu du cache est sauvegardé sur un support de mémoire stable, (par exemple un disque), pour que les informations qui s'y trouvent ne soient pas entièrement perdues dans le cas où la machine tomberait en panne (opération *Sauvegarde*). Au cours de l'opération de sauvegarde, les documents qui n'ont pas été utilisés depuis un certain temps sont éliminés du cache.

Analysons les propriétés qui sont souhaitées pour les opérations d'*Insertion* et de *Sauvegarde*.

• L'opération d'*Insertion* effectue une mise à jour de l'objet cache. Afin que la cohérence du cache soit garantie, l'opération doit être effectuée en isolation et de manière tout−ou−rien. Ces propriétés sont assurées par le fait que le cache est mis en œuvre par un objet atomique. D'autre part, la propriété de permanence n'est pas essentielle dans ce cas : il n'est pas critique de perdre les modifications survenues entre la dernière sauvegarde et le moment de la panne.

Il résulte que l'opération d'*Insertion* peut être effectuée en dehors d'une transaction. L'exécution de l'opération au sein d'une transaction aurait apporté en plus la durabilité des effets (ou la permanence), propriété qui n'est pas nécessaire dans ce cas.

• L'opération de *Sauvegarde* nécessite toutes les propriétés garanties par l'utilisation des transactions.

> Le STG permet l'utilisation des objets atomiques en dehors des transactions. Un même objet atomique peut être utilisé tantôt par une transaction, tantôt hors transaction.

III.2.2.3 L'utilisation non−contrainte des transactions

L'utilisation (i.e la création, le lancement et la terminaison) des transactions dans Guide est décidée par le programmeur : ces opérations doivent être programmées explicitement. Par opposition, il existe des systèmes répartis où, par exemple, une sous−transaction est créée automatiquement à chaque fois qu'une transaction invoque un objet situé sur un site distant (voir [42]).

Les programmeurs utilisent les transactions uniquement lorsqu'ils considèrent, compte tenu de la sémantique des traitements à effectuer, que toutes les propriétés AIP sont nécessaires. Dans l'**Exemple 2**, l'opération d'*Insertion* ne nécessite pas l'utilisation de transactions, alors que l'opération de *Sauvegarde* fait appel aux opérations de création, lancement et terminaison de transaction. Cette stratégie est en accord avec notre objectif initial, qui est de pouvoir prendre en compte les besoins des applications.

> Toute application Guide est par défaut *non−transactionnelle*, même si elle utilise des objets atomiques. Elle devient *transactionnelle* dès lors qu'une transaction est créée et lancée ; elle reste transactionnelle jusqu'à la fin de la transaction (par validation ou abandon) ; ensuite, elle redevient non−transactionnelle.
>
> Pendant toute la durée de vie d'une transaction, les opérations appliquées sur des objets atomiques sont effectuées pour le compte de cette transaction. Concernant les opérations qui s'exécutent hors transaction et sont appliquées aux objets atomiques, le STG garantit les propriétés IR *d'isolation et d'atomicité*, mais ne garantit pas la permanence de leurs effets.

III.3 La programmation des applications utilisant le service transactionnel Guide

Cette section présente l'utilisation des principales classes spécialisées qui constituent l'interface du STG (III.3.1.).

III.3.1 Les classes spécialisées

Parmi les classes spécialisées, les classes *Transaction*, *Verrou* et *Atomic* sont directement utilisées par les programmeurs d'applications transactionnelles. Cependant, elles utilisent à leur tour d'autres classes spécialisées qui sont par défaut cachées aux programmeurs (voir Fig. 3.1).

Dans la suite nous présentons les classes mentionnées du point de vue de leur mode d'utilisation.

III.3.1.1 La classe Transaction

Les transactions sont représentées au niveau d'un programme par des instances de la classe *Transaction* ; ces instances sont aussi appelées *objets transaction*. La classe *Transaction* fournit des opérations pour définir le début d'une transaction (méthode *Début*) et son mode de terminaison (méthodes *Validation* et *Abandon*). Une variable d'état privée permet de connaître la référence de la mère de tout objet transaction.

Exemple 3. Une transaction de transfert bancaire.

```
Jacques,Hélène : REF Compte; {Objets de la classe Compte}
T  :  REF  Transaction  ;  {  Objet  spécialisé  de  type
                              Transaction }

T.Début() ; {Début transaction T}
IF (Jacques.Débit(1000) = Succès)
 THEN
    IF (Hélène.Crédit(1000) = Succès)
     THEN
        T.Validation() ; {Terminaison par validation}
     ELSE
        {Le crédit a échoué}
        T.Abandon() ; {Terminaison par abandon}
    END ;
 ELSE
    {Le débit a échoué}
    T.Abandon() ; { Terminaison par abandon }
END ;
```

III.3.1.2 La classe Verrou

Les verrous sont aussi des objets, instances de la classe *Verrou* (voir IV.5.1 pour les détails de mise en œuvre). Cette classe met en œuvre une politique de synchronisation de type "un seul rédacteur, plusieurs lecteurs". Elle peut être surchargée dans le cas où le concepteur d'une application a besoin de définir des nouvelles politiques de synchronisation.

Les verrous, instances de la classe *Verrou* ou d'une sous−classe de *Verrou*, sont crées par le mécanisme de contrôle de concurrence (voir **Exemple 4** dans III.3.1.3). Les instances de la classe *Verrou* doivent être initialisées par une valeur correspondant au mode d'utilisation de l'objet à verrouiller (*lecture* ou *écriture*).

Le schéma simplifié de la classe *Verrou* est présenté ci−dessous. La méthode *ConflitMode* permet de détecter si un verrou à poser est en conflit avec un autre verrou déjà posé sur l'objet, compte tenu de la politique de synchronisation adoptée.

```
CLASS Verrou
IS
{ Variables d'état }
 Propriétaire : ..
 Mode : ..
 ....
{ Méthodes }
METHOD ConflitMode (IN autreVerrou : REF Verrou) :
Boolean ;
...
...
END Verrou .
```

III.3.1.3 La classe Atomic

La classe *Atomic* permet la création des objets atomiques et fournit les mécanismes nécessaires pour garantir les propriétés de ces objets et des transactions qui les utilisent.

Les objets atomiques sont des instances des classes atomiques définies comme sous−classes de la classe spécialisée *Atomic* (voir **Exemple 4**).

Parmi les mécanismes mis en œuvre par la classe *Atomic*, ce chapitre met en évidence le contrôle de la concurrence, qui est accessible par le biais de la méthode *Contrôle*. Cette méthode est directement utilisée dans la programmation des classes d'objets atomiques (voir IV.5.1 pour détails de mise en œuvre).

Dans la suite nous présentons un schéma simplifié de la classe *Atomic* ainsi que des exemples d'utilisation de cette classe dans la programmation des classes atomiques.

```
{ Schéma simplifié de la classe Atomic }
CLASS Atomic
IS

{ Méthode pour la synchronisation des accès }
METHOD Contrôle (IN verrou : REF Verrou,
                 nb_essais : Integer) : Boolean ;
....
....
END Atomic .
```

Exemple 4. Définition de la classe atomique *Compte* utilisée dans la transaction de transfert bancaire. Les instances de *Compte* sont des objets atomiques.

```
CLASS Compte SUBCLASS OF Atomic
IS
{ Variables d'état }
 ....
{ Méthodes }
METHOD Débit (IN somme : Real) : Boolean ;
    v : REF Verrou ; { Déclaration objet verrou }
    nb_essais : Integer ; { Nombre d'essais de verrouillage
                            en cas d'échec }
    résultat : Boolean ;
BEGIN
    { Code spécifique aux objets atomiques }
    v := new Verrou(WRITE) ;  { Création verrou en
                                mode écriture }
    nb_essais := 5 ;
    { Essai de verrouillage }
    résultat = SELF.Contrôle(v, nb_essais);

    { L'essai réussit si v n'est pas en conflit avec d'autres
    verrous posés sur l'objet Compte }
    IF (résultat = Succès)
        THEN
            ----------------------
            | traitement effectif |
            ----------------------
        ELSE
            ----------------------------------------
            | le traitement ne peut pas avoir lieu  |
            | Message d'erreur ou exception ...     |
            ----------------------------------------
    END ;

    RETURN résultat ;

END Débit ;
....

END Compte .
```

Les méthodes d'une classe atomique comportent du code spécifique et du code correspondant au traitement que la méthode doit effectuer. Le code spécifique comprend nottament la création d'un objet verrou et l'appel à la méthode *Contrôle*. La méthode *Contrôle* met en œuvre des mécanismes liés au contrôle de la concurrence (voire suite), mais aussi des fonctions nécessaires au maintient de l'atomicité des transactions qui utilisent les objets atomiques (voir chapitre V).

Lors d'un appel de méthode sur un objet atomique, si la méthode s'exécute pour le compte d'une transaction, la méthode *Contrôle* essaie de verrouiller l'objet atomique pour le compte de cette transaction (la méthode *Contrôle* fait appel à la méthode *ConflitMode* fournie par la classe *Verrou*). En cas d'échec, l'essai de verrouillage est répété un certain nombre de fois (déterminé par la valeur du paramètre *nb_essais*). Le traitement effectif ne sera effectué que dans le cas où le verrouillage a été possible ; dans le cas contraire il n'y a pas de blocage. L'exécution continue ; le programmeur peut, si besoin est, prévoir l'envoi d'un message d'erreur ou l'exécution d'un traitement d'exception.

Si une méthode invoquée ne s'exécute pas pour le compte d'une transaction, l'instance de la classe *Verrou* créée est appelée *marque* (il s'agit d'un verrou simplifié). La méthode *Contrôle* garantit dans ce cas que l'exécution de la méthode est indivisible, ce qui est suffisant pour assurer la propriété d'isolation de l'objet atomique.

L'avantage de cette approche de contrôle explicite de la concurrence consiste en ce que le programmeur dispose de la possibilité de définir et d'utiliser une classe spécifique de verrous, à la place de la classe *Verrou*, et de mettre ainsi en œuvre une politique de synchronisation différente de celle fournie par défaut.

L'opération de déverrouillage d'un objet est implantée par une autre méthode fournie par la classe *Atomic*, appelée *Déverrouillage*. Cependant, l'appel à cette méthode est effectué de façon implicite lors de la terminaison de la transaction courante.

L'opération de démarquage est faite avant la terminaison de la méthode invoquée sur l'objet atomique, de manière explicite, par l'appel à la méthode *Libération* fournie par la classe *Atomique* (voir III.3.3 pour un exemple d'utilisation).

III.3.2 Les méthodes transactionnelles

> Une méthode est appelée *transactionnelle* si le traitement qu'elle réalise est effectué pour le compte d'une transaction.
>
> La transaction pour le compte de laquelle s'exécute une méthode transactionnelle est appelée *transaction courante*.

Une méthode est transactionnelle si elle crée et lance une transaction. En outre, une méthode qui est appelée par une méthode transactionnelle est à son tour transactionnelle, même si elle ne crée pas de transaction.

La méthode *Débit* définie dans l'**Exemple 4** ne crée pas de transaction. Elle est cependant transactionnelle lorsqu'elle est utilisée tel que dans l'**Exemple 3**, car la méthode qui met en œuvre le transfert bancaire est transactionnelle.

Autrement dit, une méthode qui lance une nouvelle transaction est toujours transactionnelle, tandis qu'une méthode qui ne lance pas de transaction peut être transactionnelle ou non−transactionnelle, en fonction de son contexte d'appel.

III.3.2.1 Restrictions dans la programmation des méthodes transactionnelles

Les conditions restrictives dans la programmation des méthodes transactionnelles sont présentées à l'aide de l'exemple suivant :

Exemple 5. Programmation d'une méthode transactionnelle.

```
CLASS C_At SUBCLASS OF Atomic
IMPLEMENTS T_At
IS
{ Variables d'état }
O1, O2 : REF T_AT ; { objets atomiques }
...
{ Méthodes }
METHOD M  ; { Créatrice de transaction }
    T : REF Transaction ;
    v : REF Verrou ;
    nb_essais : Integer ;
    résultat : Boolean ;
    O3 : REF TypeNonAt ;
BEGIN
    { Création objet transaction }
    T := new Transaction ;
    { Début transaction }
    T.Début () ;
        v := new Verrou(WRITE) ;
        nb_essais := 4 ;
        { Essai de verrouillage }
        résultat = SELF.Contrôle(v, nb_essais);
        IF (résultat = Succès)
            THEN
                ----------------------
                | traitement effectif |
                ----------------------
            ELSE
                T.Abandon () ; { Fin transaction }
        END ;
        ...
    T.Validation () ; { Fin transaction }
    ..
END M ;

METHOD M1 ; { Créatrice de transaction }
```

```
...
END M1 ;

METHOD M2  ; { Ne crée pas de transaction }
...
END M2 ;


END C_At .
```

• Premièrement, le début et la fin d'une transaction doivent être programmés au sein d'une même méthode. Dans l'**Exemple 4** la transaction T est entièrement comprise dans la méthode M.

• La deuxième restriction est liée à la programmation des actions concurrentes au sein d'une transaction. Le langage Guide fournit l'instruction `CO_BEGIN .. CO_END <condition_terminaison>` qui permet la programmation des appels de méthodes concurrentes. Comme présenté dans II.2.2, les méthodes ainsi appelées sont exécutées par des activités concurrentes. Dans le cas où une telle méthode crée une transaction, des précautions doivent être prises dans la programmation de l'expression `<condition_terminaison>`, afin de permettre la terminaison par validation ou par abandon de ces transactions. Ainsi, la condition de terminaison doit devenir vraie seulement après la terminaison des activités qui exécutent les transactions.

Considérons que le traitement effectif réalisé par la méthode M dans l'**Exemple 5** contient l'appel d'une méthode M1 sur deux instances O1 et O2 de la même classe C_At. Supposons que la méthode M1 crée à son tour une transaction. La transaction créée par M1 est une sous−transaction de T (la transaction créée par M). Si les deux appels à M1 sont effectués en parallèle, le code du traitement effectif va contenir :

```
CO_BEGIN
    c1 : O1.M1;{ sous−transaction 1 }
    c2 : O2.M1;{ sous−transaction 2 }
CO_END ALL ;
```

La condition de terminaison ALL (équivalent à `c1 AND c2`) devient *`vraie`* après la terminaison des deux activités concurrentes, donc après la terminaison des deux méthodes concurrentes, et compte tenu de la première restriction, après la terminaison des deux sous−transactions concurrentes.

III.3.2.2 Le concept de transaction courante

Afin d'illustrer le concept de *transaction courante*, considérons encore l'**Exemple 5**. Supposons que le code correspondant au traitement effectif contient :

```
O3.Meth ; {objet non−atomique}
CO_BEGIN
    c1 : O1.M1;{ sous−transaction 1 }
    c2 : O2.M1;{ sous−transaction 2 }
CO_END ALL ;
O1.M2;
```

Soit O une instance de la classe atomique C_At. Lors de l'appel de la méthode M sur l'objet O, le traitement est effectué pour le compte de la transaction T. De plus, le traitement effectué sur les objets atomiques auxquels on accède à partir de l'objet cible O se fait soit pour le compte de T, soit pour celui d'une de ses sous−transactions.

La figure suivante représente les objets atteints (directement ou indirectement) à partir de l'objet O, et la transaction pour le compte de laquelle ces objets sont utilisés.

− Les objets atomiques utilisés pour le compte de T sont (en grisé clair dans la Fig. 3.2) : O, O1 (accédé directement par l'appel O1.M2, car la méthode M2 ne crée pas de transaction) et O33 (objet atomique accédé indirectement en passant par l'appel intermédiaire O3.Meth, et en supposant que Meth1 ne crée pas de transaction).

− Étant donné que la méthode M1 crée une nouvelle transaction, chacun des appels O1.M1 et O2.M1 s'exécutent pour le compte d'une sous−transaction de T.

− O3 n'étant pas atomique, le traitement effectué par Meth sur O3 se fait à l'extérieur de la transaction T.

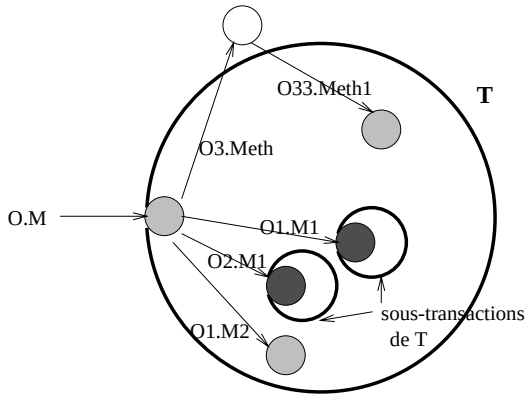

Fig. 3.2 : La portée des transactions Guide

III.3.3 Utilisation des objets atomiques en dehors des transactions

Les objets atomiques utilisés au sein d'une transaction sont verrouillés pour le compte de celle−ci et les verrous sont libérés de manière transparente et automatique lors de sa terminaison. Lorsqu'un objet atomique est utilisé hors transaction, la méthode qui lui est appliquée est non−transactionnelle. Dans ce cas, le verrou posé sur l'objet, appelé *marque*, doit être libéré à la terminaison de la méthode (l'objet doit être *démarqué*). Contrairement au déverrouillage, l'opération de démarquage est explicite. Elle est réalisée par une autre méthode de la classe *Atomic*, appelée *Libération*. La méthode *Libération* est appelée à la fin de toute méthode définie dans une classe atomique, sauf dans le cas où la méthode crée une transaction. En effet, une méthode qui crée une transaction s'exécute pour le compte de celle−ci ; l'objet cible est dans ce cas verrouillé, pas marqué.

Pour illustration, voici le code des méthodes M1 et M2 de la classe C_At introduite dans l'**Exemple 5** :

```
METHOD M1  ;
    T1 : REF Transaction ;
    v1 : REF Verrou ;
    nb_essais : Integer ;
    résultat : Boolean ;
BEGIN
    T1 := new Transaction ;
    T1.Début () ;
        v1 := new Verrou(WRITE) ;
        nb_essais := 4 ;
        résultat = SELF.Contrôle(v1, nb_essais);
        IF (résultat = Succès)
            THEN
                { traitement effectif }
                ...
            ELSE
                T1.Abandon () ;
        END ;
        ...
    T1.Validation () ;
    ..
END M1 ;

METHOD M2  ;
    v2 : REF Verrou ;
    nb_essais : Integer ;
    résultat : Boolean ;
BEGIN
    ..
```

```
        v2:= new Verrou(READ) ;
        nb_essais := 4 ;
        { Contrôle de concurrence }
        résultat = SELF.Contrôle(v2, nb_essais);
        IF (résultat = Succès)
            THEN
                { traitement effectif }
                ...
                SELF.Libération(v2) ;
            ELSE
                ...
        END ;
    END M2 ;
```

Nous pouvons remarquer que les méthodes M1 et M2 sont programmées de manière similaire en ce qui concerne le contrôle de la concurrence (appel à la méthode *Contrôle*). Cependant, la méthode M2 ne crée pas de transaction. L'appel à la méthode *Libération* assure le démarquage de l'objet atomique à la fin du traitement, dans le cas où le contexte d'exécution de M2 est non−transactionnel. Dans le cas où M2 est appelée par une méthode transactionnelle (par la méthode M dans le cas de l'**Exemple 5**), la méthode *Libération* a un rôle différent. Nous allons montrer dans le chapitre IV que l'appel à la méthode *Libération* dans un contexte transactionnel sert à l'enchaînement des méthodes concurrentes s'exécutant au sein d'une transaction (voir IV.5.1.7 pour plus de détails).

III.4 Un modèle transactionnel étendu pour un service transactionnel souple

Le service transactionnel Guide repose sur un modèle transactionnel qui offre le support pour les transactions emboîtées. Nous avons étendu le modèle des transactions emboîtées défini par Moss [41] afin d'assurer la souplesse du STG. Les extensions portent sur le modèle de concurrence et sur le modèle d'atomicité. En ce qui concerne le modèle de concurrence, le modèle de Moss n'inclut pas d'opérations non−transactionnelles et n'admet qu'une seule forme de parallélisme intra−transactionnel : l'exécution parallèle des sous−transactions sœurs. D'autre part, le modèle de Moss fournit une seule forme de cohérence, la cohérence globale des objets utilisés par une transaction. Dans la suite nous présentons les extensions que nous avons adoptées, en essayant de justifier leur nécessité. Les modèles de concurrence et d'atomicité sont présentés en détail dans les chapitres IV et V.

III.4.1 Support pour le parallélisme étendu

L'objectif est de permettre le plus de concurrence possible, tout en gardant l'isolation des transactions et des méthodes non−transactionnelles appliquées aux objets atomiques.

Le modèle de concurrence intra−transactionnelle proposé par Moss a été étendu par le rajout du support pour le parallélisme entre transactions mères et filles, ainsi que pour le parallélisme à l'intérieur d'une transaction sans l'utilisation de sous−transactions. L'extension vise non seulement à accroître le degré de parallélisme, mais aussi à permettre plus de coopération entre les composantes d'une transaction (coopération intra−transactionnelle). Pour saisir la nécessité des différentes formes de concurrence au sein d'une transaction, considérons l'exemple suivant :

Exemple 6. Soit T une transaction effectuant les appels de méthode O.M1 et O.M2 en parallèle, où O est un objet atomique.

```
T : REF Transaction ;
O : REF ..;

T.Début();
    CO_BEGIN
        c1 : O.M1 ;
        c2 : O.M2 ;
    CO_END ALL ;
T.Validation ;
```

Dans l'hypothèse où M1 et M2 créent des transactions, nous retrouvons le cas classique de concurrence des sous−transactions (cas I dans la Fig. 3.3). Supposons que la méthode M1 crée une transaction T1 et que la méthode M2 ne crée pas de transaction ; alors M1 s'exécute pour le compte de la sous−transaction T1, et M2 s'exécute pour le compte de T. Nous sommes dans un cas de parallélisme mère−fille (cas II en Fig. 3.3). Si aucune des méthodes M1 et M2 ne crée de transaction, l'objet O est utilisé par deux processus concurrents s'exécutant pour le compte de la même transaction T (cas III en Fig. 3.3). Ce cas montre qu'il est nécessaire de fournir du support pour le parallélisme intra−transactionnel, au delà du parallélisme des sous−transactions.

Ces différentes formes de parallélisme peuvent engendrer des problèmes de cohérence, car elles peuvent mettre en cause l'isolation des transactions et des méthodes appliqués aux objets atomiques. Le modèle de concurrence présenté dans le chapitre IV prend en compte ces différentes formes de parallélisme intra−transactionnel, de manière à préserver la cohérence des objets atomiques. De plus, le modèle de concurrence prend aussi en compte l'exécution concurrente des transactions et des méthodes non−transactionnelles, ainsi que la concurrence entre des méthodes non−transactionnelles.

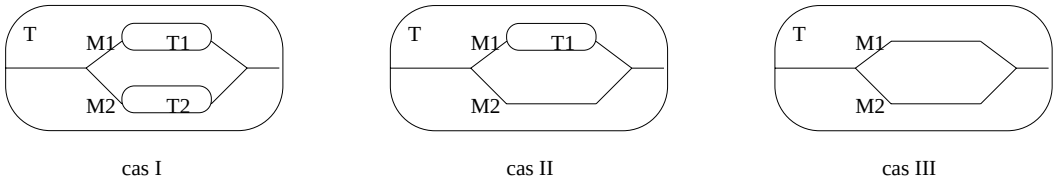

Fig. 3.3 : Les différents cas de parallélisme intra−transactionnel

III.4.2 Support pour différents degrés de cohérence

Dans le chapitre I nous avons mis en évidence deux niveaux de cohérence : global et local. La cohérence globale (garantie par la propriété d'atomicité globale) maintient un groupe d'objets en cohérence (maintient à la fois leurs cohérence interne et la cohérence des uns par rapport aux autres). La cohérence locale (garantie par la propriété d'atomicité locale) fait référence uniquement à la cohérence interne d'un objet.

Le Service Transactionnel de Guide fournit les deux niveaux de cohérence. La cohérence globale est garantie pour tous les objets atomiques utilisés par une transaction. La cohérence locale est garantie pour tout objet atomique utilisé à travers une méthode non−transactionnelle. Enfin, le système ne garantit pas la cohérence des objets non−atomiques.

Par conséquent, les propriétés des objets Guide sont déterminées par :

- leur catégorie (atomiques ou non−atomiques),
- leur contexte d'utilisation (transactionnel ou non−transactionnel).

La catégorie d'un objet est fixée statiquement, lors de la compilation de sa classe. Les propriétés d'un objet atomique sont dépendantes de son contexte d'utilisation.

Le contexte d'utilisation d'un objet atomique est transactionnel si la méthode qui lui est appliquée est transactionnelle (cf. à III.3.2, si elle est exécutée pour le

compte d'une transaction). Dans ce cas, le STG garantit la cohérence globale des objets qui se trouvent dans la portée de la transaction courante.

Les propriétés garanties par le STG	Objet atomique	Objet non−atomique
Contexte transactionnel	Cohérence globale	Pas de garanties
Contexte non−transactionnel	Cohérence locale	Pas de garanties

Si le contexte d'utilisation d'un objet est non−transactionnel, le STG garantit uniquement la cohérence interne : l'isolation par rapport aux transactions et aux non−transactions concurrentes ainsi que l'atomicité (exécution de manière tout−ou−rien) des opérations en présence des pannes. La permanence des modifications effectuées par une méthode non−transactionnelle sur un objet atomique n'est pas garantie (dans l'absence des pannes la permanence est assurée par le service de stockage du système Guide). De plus, l'application ne peut pas demander l'annulation des modifications effectuées, car il n'y a pas de transaction pour contrôler le retour arrière. En conséquence, l'atomicité des opérations non−transactionnelles doit être assurée par des mécanismes qui fonctionnent indépendamment de ceux qui assurent l'atomicité des transactions.

III.5 Conclusion

Le Service Transactionnel Guide (STG) est adapté à la programmation des applications réparties à base d'objets grâce à son interface constituée par des classes spécialisés. Les concepts de *transaction* et d'*objet atomique,* mis à la disposition des concepteurs d'application, facilitent la structuration des applications. Le STG offre beaucoup de souplesse dans l'utilisation des transactions et des objets atomiques, permettant ainsi la prise en compte des besoins des applications en termes de cohérence.

Afin de garantir les caractéristiques énoncées, nous avons élaboré un modèle transactionnel étendu, avec support pour les transactions emboîtées. Les extensions adoptées permettent d'augmenter le degré de parallélisme, ainsi que de garantir aux objets utilisés par une application le degré de cohérence correspondant à la sémantique des traitements qui leurs sont appliqués.

Les chapitres suivants présentent les deux volets du modèle : le contrôle de la concurrence et le contrôle de la cohérence face aux pannes.

Chapitre IV

Le contrôle de la concurrence

IV.1 Introduction

Ce chapitre présente le modèle de concurrence, ainsi que les mécanismes qui, sur la base de ce modèle, assurent le contrôle de la concurrence dans le Service Transactionnel Guide (STG).

Le chapitre précédent à montré que, pour réaliser l'objectif de souplesse du STG, il est nécessaire d'adopter un modèle transactionnel dérivé de celui des transactions emboîtées proposé par Moss [42]. Le modèle de Moss a été brièvement présenté dans I.3.1. Afin de mieux comprendre les extensions apportées à ce modèle, nous revenons dans IV.2 sur certains aspects du modèle de Moss qui sont liés au contrôle de la concurrence. Ces aspects sont concrétisés par un ensemble de règles de verrouillage. Nous mettons en évidence le fait que les règles de verrouillage du modèle de Moss étendent les règles de verrouillage du modèle des transactions plates par l'intégration d'une règle de compatibilité entre les composantes d'une transaction emboîtée. Cette règle de compatibilité rend possible l'exécution concurrente des sous−transactions.

Nous avons adopté une démarche similaire pour définir un ensemble de règles de verrouillage qui étendent les règles de verrouillage de Moss. Des nouvelles règles de compatibilité ont été introduites, afin de permettre des formes variées de concurrence au sein d'une transaction emboîtée et pour assurer le support du parallélisme entre transactions et méthodes non−transactionnelles. Le modèle est enrichi par des règles de marquage qui établissent les conditions d'utilisation des objets atomiques par les méthodes non−transactionnelles. Il faut noter que l'ensemble des règles ainsi obtenues intègre certaines extensions au modèle de Moss existantes dans la littérature, et qui ont été, dans un premier temps, adaptées à la programmation à base d'objets, ensuite au modèle de programmation de Guide.

Le modèle de concurrence étendu a été élaboré de manière incrémentale. Le support pour les différentes formes de concurrence, aussi bien que de coopération entre les composantes transactionnelles, est rajouté par étapes au modèle de concurrence de Moss. Ces étapes sont décrites dans la section IV.3. L'ensemble

des règles de verrouillage et de marquage finalement obtenues se retrouve dans la section IV.4.

La description du modèle est suivie par des éléments de réalisation qui mettent en évidence le fonctionnement des différentes composantes concernées par le contrôle de la concurrence, tant au niveau des classes spéciales qu'au niveau du noyau du système Guide.

IV.1.1 Les différentes formes de parallélisme

Voici les formes de parallélisme que notre modèle doit fournir.

• À l'intérieur d'une transaction nous distinguons trois types de parallélisme.

− Le *parallélisme des sous−transactions* : apparaît lorsque les filles d'une transaction peuvent s'exécuter en parallèle. Le modèle de Moss prévoie uniquement cette forme de parallélisme, dans laquelle il peut y avoir concurrence entre les transactions d'un même arbre seulement si elles ne se trouvent pas sur le même chemin.

− Le *parallélisme mère/fille* : apparaît lorsqu'une transaction peut s'exécuter en parallèle avec ses filles. Comme une transaction fille peut avoir à son tour des sous−transactions avec qui lesquelles elle s'exécute en parallèle, nous identifions un cas plus général de parallélisme : le *parallélisme transaction/inférieur*. Dans ce cas, il peut y avoir de la concurrence entre des transactions qui se trouvent sur le même chemin d'un arbre transactionnel.

− Le *parallélisme transaction/descendant* est une généralisation du cas précédant à la situation où des processus concurrents peuvent être créés pendant l'exécution d'une transaction et s'exécutent pour le compte de cette même transaction. C'est comme si la transaction s'exécutait en parallèle avec elle−même. La réunion des cas de parallélisme de type *transaction/transaction* et de type *transaction/inférieur*, conduit au cas général de parallélisme de type *transaction/descendant*.

• Les différentes formes de parallélisme pouvant exister à l'extérieur des transactions sont présentées dans le tableau suivant :

Table I

	transaction	méthode non−transactionnelle
transaction	(1)	(2)
méthode non−transactionnelle	(2)	(3)

(1) C’est le cas classique, prévu dans le modèle de Moss. Il s’agit de traiter la concurrence de transactions qui se trouvent dans des arbres différents.

(2) On prend en compte la concurrence entre une transaction et une méthode non−transactionnelle.

(3) On envisage les accès concurrents des méthodes non−transactionnelles à un même objet atomique.

IV.2 Le contrôle de la concurrence dans le modèle de Moss

Si, dans le modèle classique des transactions plates, le verrouillage est un moyen pour assurer la synchronisation des transactions, dans le modèle de Moss le verrouillage assure la synchronisation des composantes d’une même transaction emboîtée, ainsi que la synchronisation des composantes appartenant à des transaction emboîtées distinctes (voir Fig. 4.1).

Avant de présenter les règles de verrouillage de Moss, nous rappelons les règles de verrouillage des transactions plates.

IV.2.1 Les règles de verrouillage classiques

Les règles de verrouillage classiques sont basées sur deux modes d’accès aux objets : en *lecture* et en *écriture*. Elles mettent en œuvre une politique de synchronisation de type "plusieurs lecteurs, un seul rédacteur" : le mode d’accès en écriture est conflictuel avec tous les autres et le mode d’accès en lecture est compatible seulement avec lui même.

Les règles de verrouillage sont basées sur le principe qu’une transaction doit verrouiller tout objet atomique avant de l’utiliser. Elles précisent aussi les conditions dans lesquelles les verrous posés par une transaction peuvent être relâchés.

Règles de verrouillage classiques

R1 : Une transaction T peut verrouiller un objet en lecture si et seulement si aucune autre transaction n’a verrouillé l’objet en écriture.

R2 : Une transaction peut verrouiller un objet en écriture si et seulement si aucune autre transaction n’a verrouillé l’objet (en aucun mode).

R3 : Les verrous posés par une transaction ne sont pas relâchés avant sa terminaison.

Les règles de verrouillage classiques garantissent la sérialisation et la recouvrabilité des transactions plates. La règle **R1** empêche une transaction T de lire des objets qui se trouvent dans un état instable, ce qui garantit l'isolation de T par rapport à des transactions concurrentes. La règle **R2** empêche T de modifier les objets lus ou modifiés par d'autres transactions en cours d'exécution, ce qui protège les transactions concurrentes des effets de T. Les règles **R1** et **R2** sont basées sur le principe qu'un verrou posé sur un objet ne doit pas être relâché tant que la transaction utilise l'objet. La règle **R3** garantit que l'abandon d'une transaction n'entraîne pas de "rejets en cascade".

IV.2.2 Les règles de verrouillage de Moss

L'extension des règles de verrouillage classiques répond aux nécessités suivantes :

- Garantir l'isolation des sous−transactions concurrentes.
- Garantir les règles de visibilité du modèle, définies en I.3.1.

L'impact du modèle de concurrence

Le modèle de Moss permet l'exécution parallèle des sous−transactions. En revanche, les formes de parallélisme mère/fille ou transaction/descendant (IV.1.1) ne sont pas supportées.

Plusieurs stratégies peuvent être adoptées pour empêcher l'exécution d'une transaction en concurrence avec ses inférieurs. Par exemple en Argus [38] une transaction mère est suspendue pendant l'exécution de ses filles. Dans le prototype réalisé par Moss, seules les transactions feuilles (qui n'ont pas de sous−transactions) ont le droit d'effectuer des opérations de lecture ou d'écriture sur les données.

La correction du modèle des transactions emboîtées est assurée par la *sérialisation des sous−transactions sœurs* (ce qui garantit la cohérence des objets partagés par des composantes d'une même transaction globale), ainsi que par la *sérialisation des arbres transactionnels* (ce qui garantit la cohérence des objets partagés par des composantes de transactions globales différentes).

Considérons d'abord la concurrence interne à une transaction. Soit T une transaction emboîtée avec des composantes qui s'exécutent en parallèle (Fig. 4.1). Soit T4 et T2 deux composantes concurrentes qui ne se trouvent pas sur le même chemin dans l'arbre transactionnel. Supposons que T2 entre en conflit avec T4 lors de l'accès à l'objet atomique O1 (par exemple, T4 a déjà verrouillé O1 en écriture lorsque T2 tente de le lire). Les règles **R1** et **R3** empêchent la visibilité de l'effet de

T4 sur O1, jusqu'au moment de la validation de T4. Cependant, cette protection n'est pas suffisante, car la validation de T4 est dépendante de la validation de ses supérieurs. Pour verrouiller l'objet O1, T2 doit attendre la validation de tous les supérieurs de T4 qui se trouvent sur un autre chemin qu'elle dans l'arbre. Dans le cas considéré, T2 pourra verrouiller O1 en lecture seulement après la validation (ou le rejet) de T1. Si T1 pouvait voir les modifications de T4 avant la validation de T1, un éventuel abandon de T1 entraînerait l'annulation des effets de T4 et donc le rejet de T2. Il faut noter que T1 est la sœur de T2 ; l'utilisation de l'objet O1 par la transaction T2 seulement après la validation de T1 revient à sérialiser les transactions sœurs T2 et T1.

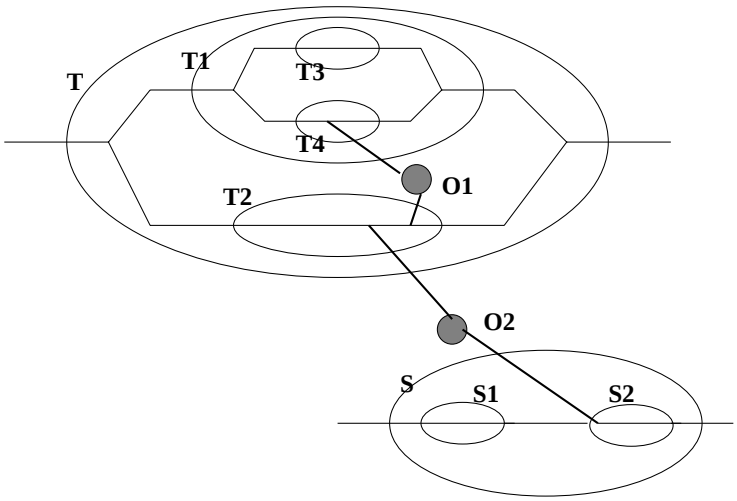

Fig. 4.1 : Le partage des objets atomiques entre différentes sous−transactions

Considérons ensuite le cas de concurrence entre des transactions qui appartiennent à des arbres différents. Supposons que la sous−transaction S2 entre en conflit avec la sous−transaction T2 lors de l'accès à l'objet atomique O2 (Fig. 4.1). Le raisonnement ci−dessus peut être appliqué à nouveau. Les règles de verrouillage classiques ne suffisent pas, car les effets de la sous−transaction T2 doivent être rendus visibles à l'extérieur de son arbre transactionnel seulement après la validation de la transaction racine T. Ceci revient a sérialiser les transactions T et S.

Les problèmes présentés ci−dessus sont en partie résolus par l'introduction du concept d'*héritage ascendant* des verrous :

L'héritage ascendant des verrous

> Lors de la validation d'une sous−transaction T_i, la transaction mère T_j hérite les verrous de T_i.

En d'autres termes, les objets verrouillés par une sous−transaction restent verrouillés après sa validation pour le compte de sa mère, qui devient ainsi le propriétaire des verrous hérités.

Ce mécanisme garantit l'isolation de la transaction globale, car les verrous posés par une composante ne sont pas relâchés avant la validation de la racine. L'héritage ascendant des verrous est donc suffisant pour garantir la sérialisation des arbres transactionnels.

L'impact des règles de visibilité

Les règles de verrouillage **R1** et **R2** doivent être étendues pour permettre à une sous−transaction d'utiliser les objets verrouillés par ses ancêtres. Ceci doit se faire sans restrictions, car les composantes qui se trouvent sur le même chemin dans leur arbre transactionnel ne peuvent pas s'exécuter en concurrence. L'extension est concrétisée par la règle de compatibilité suivante :

Règle de compatibilité I

> Considérons qu'une transaction essaie de verrouiller un objet atomique déjà verrouillé dans un mode conflictuel. Cette demande de verrouillage est acceptée, si et seulement si, le propriétaire du verrou déjà posé est un ancêtre de la transaction demandant le verrouillage.

La règle de compatibilité et l'héritage ascendant des verrous sont suffisants pour résoudre la sérialisation des composantes sœurs :

− L'héritage ascendant assure l'isolation d'une composante T_i par rapport aux sœurs concurrentes. Considérons le cas de la composante T1 dans l'exemple illustré en Fig. 4.1. Grâce à l'héritage ascendant l'objet O1 reste verrouillé après la validation de T4, ce qui assure l'isolation de T1 par rapport à T2. Après la validation de T1, ses verrous son hérités par sa mère T (le verrou posé sur O1 est hérite par T).

− La règle de compatibilité permet aux sœurs d'une composante T_i de verrouiller les objets utilisés par T_i, après sa validation. Dans l'exemple considéré, T2 pourra lire l'objet O1 après la validation de T1 grâce à la règle de compatibilité, car le propriétaire du verrou en écriture posé sur O1 devient T.

Les règles de verrouillage de Moss sont obtenues en incorporant la règle de compatibilité et le concept d'héritage ascendant des verrous dans les règles de verrouillage classiques :

Règles de verrouillage de Moss

R1 : Une transaction peut verrouiller un objet en lecture si et seulement si toutes les transactions qui détiennent un verrou en écriture sur l'objet sont ses ancêtres.

R2 : Une transaction peut verrouiller un objet en écriture si et seulement si toutes les transactions qui détiennent un verrou sur l'objet (dans un mode quelconque) sont ses ancêtres.

R3 : Lorsqu'une transaction est validée, ses verrous sont gardées par sa mère (si elle existe). Si la transaction validée est une racine (ou une transaction plate) ses verrous sont relâchés.

R4 : Lorsqu'une transaction est abandonnée, ses verrous sont relâchés.

Les règles **R1** et **R2** incorporent la Règle de compatibilité I dans les règles **R1** et **R2** initiales. Le principe de l'héritage ascendant des verrous est capté par la règle **R3** qui spécifie le traitement des verrous lors de la validation d'une transaction. La règle **R4** spécifie le traitement des verrous lors de l'abandon d'une transaction.

Pour la mise en œuvre de ces règles, les verrous doivent être caractérisés par les attributs suivants :

- **propriétaire** : la transaction pour le compte de laquelle l'objet est verrouillé (la transaction qui détient le verrou).
- **mode d'accès** : *lire* ou *écrire*.

IV.3 Extensions au modèle de Moss

Des nombreuses extensions sont proposées dans la littérature afin d'enrichir le modèle des transactions emboîtées. Nous nous sommes intéressés plus particulièrement aux extensions qui visent les objectifs suivants :

- Permettre différentes formes de parallélisme au sein des transactions.
- Faire cohabiter les transactions avec les méthodes non−transactionnelles.
- Augmenter le degré de coopération entre les composantes d'une même transaction.

Les deux premières catégories d'extension visent l'amélioration des performances en permettant plus de parallélisme dans le système. La troisième catégorie se propose d'améliorer l'utilisation des transactions emboîtées en tant qu'outils pour la structuration des applications.

IV.3.1 Support pour des formes variées de concurrence au sein d'une transaction

La Règle de compatibilité I incorporée dans les règles de verrouillage de Moss est incompatible avec les autres formes de parallélisme définies dans IV.1.1. Dans la suite nous allons présenter des extensions progressive de la Règle de compatibilité I, afin de permettre le parallélisme mère/fille et le parallélisme transaction/transaction.

Extension pour le parallélisme de type mère/fille

Conformément à la **Règle de compatibilité I**, une transaction fille a le droit d'utiliser sans restrictions les objets verrouillés par sa mère. Dans l'hypothèse où la fille peut s'exécuter en concurrence avec sa mère, il est nécessaire de restreindre ce droit, sinon l'abandon de la fille risque d'entraîner celui de la mère, ainsi que le montre l'exemple suivant.

Exemple 1. Supposons que la transaction T verrouille en lecture l'objet atomique O et que la transaction T1, fille de T, verrouille en écriture le même objet O. Si T et T1 s'exécutent en parallèle, on peut avoir les deux scénarios suivants, dans lequel T devient dépendante de l'issue de T1 :

Scénario 1

1. T verrouille O en lecture.
2. T1 verrouille O en écriture.
3. T1 modifie O.
4. T lit O.

Scénario 2

1. T verrouille O en lecture.
2. T lit O.
3. T1 verrouille O en écriture.
4. T1 modifie O.
5. T lit O.

Dans les deux cas, si T1 est abandonnée, T doit être rejetée. De plus, l'exécution correspondante au scénario 2 n'est pas sérialisable.

L'objectif est de reformuler la **Règle de compatibilité I** de manière qu'elle garantisse l'isolation d'une transaction par rapport à ses filles concurrentes, tout en conservant l'esprit de la règle de visibilité du modèle de Moss.

La solution proposée par Walter [51] est basée sur le principe qu'une transaction doit pouvoir verrouiller dans un mode conflictuel un objet déjà verrouillé par sa mère (ou, plus généralement, par un de ses supérieurs), seulement si sa mère (le supérieur) n'utilise pas réellement l'objet, par exemple si ce supérieur a hérité du verrou posé sur l'objet.

Pour mettre en œuvre cette solution, il faut différencier les verrous acquis par une transaction des verrous dont elle a hérité. De plus, il faut préciser qu'un verrou hérité ne donne pas à son propriétaire le droit d'utiliser l'objet verrouillé.

Concrètement, les verrous doivent être caractérisés par un *état,* qui peut être :

- *détenu*, pour les verrous acquis par une transaction, ou
- *retenu*, pour les verrous hérités par la transaction

Une transaction *détient* donc les verrous qu'elle a acquis, et *retient* les verrous acquis par ses sous−transactions validées.

L'état des verrous est pris en compte seulement pour la détection des conflits entre les transactions qui se trouvent en relation d'ascendance.

Ainsi, soit T une transaction qui tente de verrouiller l'objet atomique O (Fig. 4.2). On suppose que O se trouvait déjà verrouillé par T' en écriture. Dans le cas I, où T' est un supérieur de T, T peut verrouiller O seulement si T' retient le verrou sur O. Remarquons que, dans le cas où T' détient un verrou sur O, une situation d'interbloquage peut apparaître : T attend de pouvoir verrouiller O et T' attend la fin de T pour pouvoir valider, donc pour déverrouiller O. Cette carence, due aux limitations imposées par le modèle de Moss, peut être remédiée par le support de l'héritage descendant des verrous (voir IV.3.3). Dans le cas II, où T' n'est pas un supérieur de T, l'état du verrou posé sur O n'est pas pris en compte ; le conflit est détecté sur la base du mode d'accès.

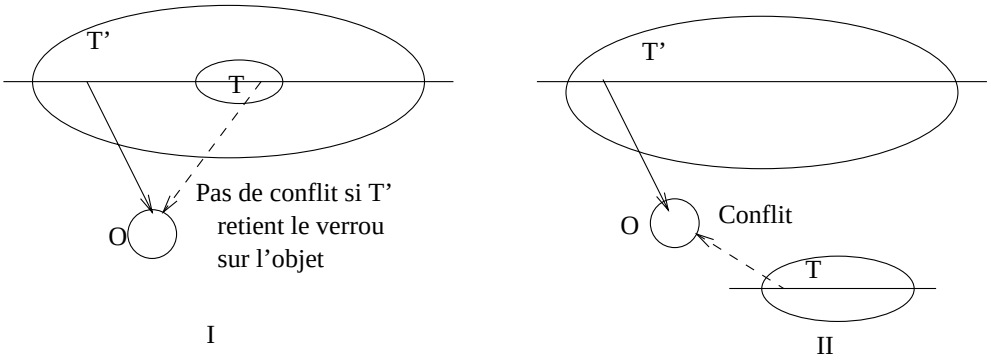

Fig. 4.2 : Détection d'un conflit entre deux transaction T et T'

L'intégration du concept d'*état* d'un verrou introduit une première modification de la Règle de compatibilité I. La règle ainsi obtenue est :

Règle de compatibilité II

Soit la situation où une transaction T essaie de verrouiller un objet atomique qui est déjà verrouillé par la transaction T' dans un mode conflictuel.

Si T et T' sont identiques, la demande de verrouillage est acceptée. Dans le cas contraire, la demande de verrouillage est acceptée si et seulement si les deux conditions suivantes sont simultanément vraies :

- T' est un supérieur de T, et
- l'état du verrou posé par T' est *retenu.*

Le parallélisme de type transaction/transaction

Selon la **Règle de compatibilité II**, une transaction ne peut jamais se trouver en conflit avec elle même, ceci afin de permettre à une transaction d'effectuer des opérations successives sur un même objet. Si des processus concurrents peuvent être exécutés pour le compte d'une même transaction, alors ces processus pourraient effectuer des opérations conflictuelles sur un même objet atomique, dont la cohérence interne n'est alors plus garantie.

Pour garantir la cohérence interne d'un objet atomique utilisé d'une manière concurrente au sein d'une transaction, il suffit d'ordonnancer les processus concurrents. Ceci nécessite l'utilisation de l'identité des processus pour la détection des conflits. Hermes/ST [33] est un exemple de système transactionnel qui utilise cette approche afin de permettre le parallélisme au sein d'une même transaction.

Nous proposons une approche basée sur l'ordonnancement des méthodes concurrentes conflictuelles.

Pour détecter les conflits entre les méthodes concurrentes, l'exécution du traitement effectué par une méthode qui est appelée sur un objet atomique est précédée par une opération de verrouillage. De même qu'en Hermes/ST, cette opération prend en compte l'identité du processus qui exécute la méthode dans la détection des conflits. Cependant, afin d'assurer l'ordonnancement des méthodes conflictuelles, à la terminaison d'une méthode qui a verrouillé un objet atomique l'état du verrou passe en *retenu.*

La règle de compatibilité suivante étend la **Règle de compatibilité II** par l'intégration de *l'identité du processus* qui exécute les méthodes transactionnelles.

Règle de compatibilité III

Soit la situation où une transaction T essaie de verrouiller un objet atomique O qui est déjà verrouillé par la transaction T' dans un mode conflictuel. Soit V' le verrou posé sur O, et P' le processus qui exécute, pour le compte de T', la méthode appelée sur O. On distingue les deux situations suivantes :

I. Si T et T' sont identiques, la demande de verrouillage est acceptée, selon l'identité des processus, si :

- P' et le processus courant sont identiques (il s'agit dans ce cas des méthodes successives appliquées sur O).
- Ces deux processus sont différents, mais l'état de V' est *retenu* (car dans ce cas, la méthode concurrente qui a posé le verrou V' est terminée).

II. Si T et T' sont des transactions différentes, la demande de verrouillage est acceptée, si et seulement si les deux conditions suivantes sont vraies:

- T' est un supérieur de T, et
- l'état du verrou V' posé par T' est *retenu*.

L'extension présentée ci−dessus garantit la *cohérence locale* des objets atomiques utilisés en concurrence au sein d'une transaction. Néanmoins, il est nécessaire de vérifier que la correction du modèle est toujours assurée. Pour cela, on constate que, premièrement, l'isolation de la transaction n'est pas remise en cause

par cette extension, car l'état des verrous n'est pas pris en compte lors de la détection des conflits entre deux transactions différentes qui ne se trouvent pas sur le même chemin d'un arbre transactionnel (il s'agit du cas II, Fig. 4.2). Deuxièmement, on constate que le changement en *retenu* de l'état d'un verrou, après la terminaison de la méthode qui l'a posé, correspond bien à la sémantique de cet attribut, car l'objet verrouillé n'est plus réellement utilisé. Ce deuxième constat montre que l'isolation de la transaction par rapport aux propres sous−transactions est respecté.

Pour mettre en œuvre la Règle de compatibilité III, les verrous doivent être caractérisés par les attributs suivants :

- **propriétaire** : l'identité de la la transaction pour le compte de laquelle l'objet est verrouillé ;
- **mode** : *lire* ou *écrire* ;
- **état** : *détenu* ou *retenu* ;
- **processus** : l'identité du processus qui exécute la méthode transactionnelle effectuant la prise de verrou.

En conclusion, on peut assurer le support des différentes formes de parallélisme définies en IV.1.1, moyennant le remplacement de la Règle de compatibilité I par la Règle de compatibilité III proposée ici. On constate que l'introduction de l'attribut *état* pour les verrous doit être prise en compte non seulement par le mécanisme d'héritage ascendant (lors de la validation des sous−transactions), mais aussi lors de la terminaison des opérations.

IV.3.2 Support pour la concurrence des transactions et des méthodes non−transactionnelles

Le modèle de Moss doit de plus être étendu pour la prise en compte de deux nouvelles formes de concurrence, dues à l'existence des méthodes non−transactionnelles :

1. La concurrence entre transactions et méthodes non−transactionnelles partageant un même objet atomique.
2. La concurrence entre des méthodes non−transactionnelles appliquées à un même objet atomique.

Pour garantir la cohérence des objets atomiques partagés dans ces cas dans la situation où les accès concurrents sont conflictuels, il est nécessaire de respecter les règles suivantes :

1. Si un objet atomique est utilisé par une transaction, celle−ci doit se terminer (être validée ou annulée) avant qu'une méthode non−transactionnelle concurrente puisse utiliser l'objet.
2. Si une méthode non−transactionnelle est appliquée à un objet atomique, elle doit se terminer avant qu'une transaction concurrente puisse verrouiller l'objet.
3. Si une méthode non−transactionnelle est appliquée à un objet atomique, elle doit se terminer avant qu'une méthode non−transactionnelle concurrente puisse utiliser l'objet.

L'extension que nous proposons pour satisfaire à ces conditions est basée sur l'observation qu'une méthode non−transactionnelle effectue un *marquage* de l'objet atomique cible, de la même manière qu'une transaction *verrouille* les objets qu'elle utilise. La différence essentielle entre marquage et verrouillage provient du fait que le verrou reste posé jusqu'à la fin de la transaction, alors que le marquage est actif uniquement pendant la durée de l'exécution de la méthode non−transactionnelle. À la fin de l'exécution d'une méthode non−transactionnelle appliquée à un objet atomique, cet objet est démarqué.

Les règles de marquage qui seront présentées dans la section IV.4.2 vont permettre de détecter si une méthode non−transactionnelle entre en conflit, soit avec une transaction concurrente, soit avec une autre méthode non−transactionnelle concurrente.

D'autre part, les règles de verrouillage doivent être étendues pour détecter si un conflit apparaît lorsqu'une transaction essaie de verrouiller un objet marqué. Le conflit est détecté sur la base du mode d'accès.

IV.3.3 Support pour l'héritage descendant des verrous

L'héritage ascendant des verrous permet un certain degré de coopération entre les composantes d'une transaction emboîtée. Par exemple, deux sous−transactions sœurs peuvent partager un objet atomique O, dans le sens que si l'une modifie O l'autre peut le lire ou le modifier après la validation de la première. Cependant, la coopération entre une transaction et ses sous−transactions est limitée par la prise en compte du parallélisme mère/fille : si une transaction a verrouillé un objet atomique O, aucune de ses filles ne pourra l'utiliser (en effet, le verrou posé sur O reste dans l'état *détenu* jusqu'à la terminaison de la transaction, or la transaction ne peut pas se terminer avant ses filles). Cette limitation peut empêcher certaines

décompositions de tâches entre les membres d'une hiérarchie transactionnelle, comme dans l'exemple suivant.

Exemple 2. Considérons une application qui parcourt une liste d'objets et applique à chaque objet une opération de mise à jour. Soit T la transaction créée pour effectuer le traitement correspondant au premier objet de la liste, O1 (voir Fig. 4.3). T verrouille O1 en lecture pour déterminer l'objet suivant O2. L'opération de mise à jour de O1 est effectuée par T1, une sous−transaction de T.

Pour optimiser l'application, les opérations de mise à jour devraient être effectuées de manière asynchrone. Il suffirait pour cela de lancer T1 en parallèle avec le traitement de O2 : de cette manière, les mises à jour des différents objets seraient effectuées par des processus parallèles. Cependant, ce parallélisme est pour l'instant impossible, étant donné qu'à partir du moment où la transaction T détient un verrou en lecture sur O1 la transaction fille T1 ne peut plus verrouiller O1 en écriture.

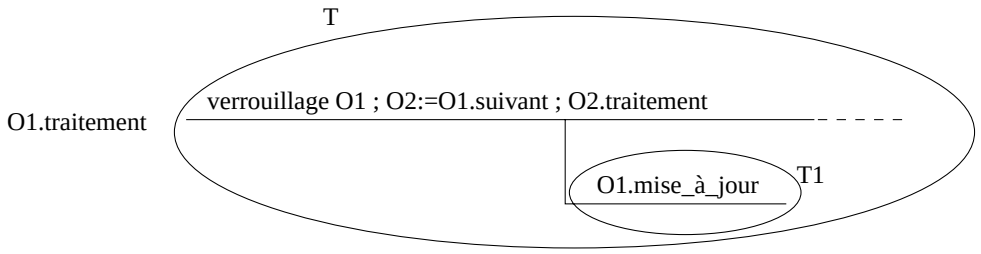

Fig. 4.3 : Exemple de structuration d'un traitement à l'aide des transactions emboîtées

Nous résoudrons ce problème en faisant appel au mécanisme d'*héritage descendant*, qui élargit le champ de coopération entre les composantes d'une transaction. Grâce à ce mécanisme, une sous−transaction pourra, sous certaines conditions, verrouiller un objet utilisé par un de ses supérieurs. L'héritage descendant est une opération explicite : la décision de l'utiliser peut être prise uniquement par l'application.

L'héritage descendant se concrétise par une *offre* de verrou. Lorsqu'une transaction T qui détient un verrou posé sur un objet *offre* son verrou, elle renonce à l'utilisation de l'objet, en faveur de ses sous−transactions. En conséquence, une des sous−transactions de T peut verrouiller cet objet dans le mode de son choix. Dans l'exemple précédent, T peut offrir le verrou posé sur O1, pour permettre à la sous−transaction T1 de verrouiller et mettre à jour O1.

Pour ce faire, Harder et Rothermel (voir [29]) ont étendu le modèle de Moss par l'introduction d'une règle permettant à une transaction d'offrir des verrous. Pour que la sémantique de l'offre de verrou soit celle définie ci−dessus, il suffit de modifier l'état du verrou offert en *retenu*. Soit T' une transaction qui offre le verrou V' qu'elle avait préalablement posé sur l'objet O1 (Fig. 4.4). Si la transaction T n'est pas un descendant de T', alors, conformément à la Règle de compatibilité III, elle ne peut pas verrouiller l'objet O1 en écriture. En conséquence, l'offre du verrou ne met pas en cause l'isolation de la hiérarchie de T'. Par contre, si T est une sous−transaction de T' alors T pourra verrouiller l'objet O1 dans le mode désiré.

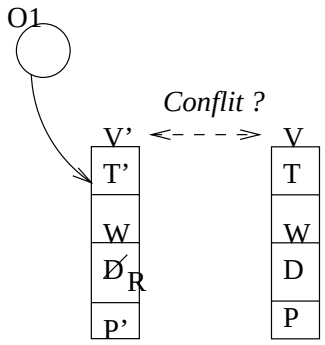

Fig. 4.4 : L'offre du verrou V' permet la pose de V si T est sous−transaction de T'

L'héritage descendant est utilisé par une transaction lorsqu'il s'agit de permettre à ses sous−transactions l'accès à un objet qu'elle avait verrouillé auparavant. Toutefois, en offrant son verrou la transaction perd le contrôle sur le mode d'accès à l'objet. Les auteurs [29] montrent que ce contrôle reste pourtant possible si la transaction qui offre son verrou acquiert un nouveau verrou dans un mode dégradé par rapport à celui du verrou offert.

L'opération d'offre de verrou est appelée *offre contrôlée*, lorsqu'elle est accompagnée par la prise d'un nouveau verrou dans un mode dégradé.

Par exemple, si la transaction T' souhaite que ses sous−transactions puissent lire l'objet O1 sans qu'elles puissent le modifier, elle offre V' et prend un nouveau verrou V'' en lecture (voir Fig. 4.5). Comme V'' est dans l'état *détenu*, les sous−transactions ne pourront pas verrouiller O1 en écriture. Il faut noter que, conformément à la Règle de compatibilité III, les transactions qui ne font pas partie de la hiérarchie de T ne peuvent ni lire ni écrire l'objet O1.

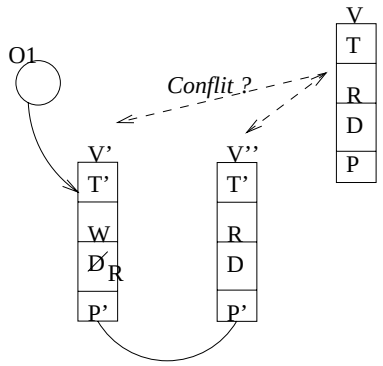

Fig. 4.5 : La transaction T peut verrouiller l'objet O1 en lecture si elle est sous−transaction de T'

IV.4 Le modèle étendu de contrôle de concurrence

Le modèle de concurrence des transactions Guide doit étendre le modèle de Moss selon les trois directions présentées dans la section précédente. Le modèle est à concrétiser par des règles de verrouillage et de marquage qui spécifient les conditions dans lesquelles une transaction ou une méthode non−transactionnelle peuvent utiliser un objet atomique.

Le modèle de concurrence est défini de manière à prendre en compte le modèle d'exécution du système Guide (voir II.3.1). Nous rappelons seulement que dans Guide, les objets sont des entités passives et les activités sont des entités actives, une activité étant un flot d'exécution de méthodes appliqués à des objets cible.

Lorsque l'objet cible est atomique, l'exécution de l'opération mise en œuvre par la méthode est précédée par le verrouillage ou le marquage de l'objet.

− Si la méthode est transactionnelle (voir III.3.2), alors l'objet est verrouillé pour le compte de la transaction courante.

− Dans le cas contraire, l'objet est marqué.

Étant données les différentes formes de parallélisme considérées, les règles de verrouillage doivent permettre la détection de conflits de différentes natures, à savoir :

1. entre la transaction demandant le verrouillage et d'autres transactions concurrentes
2. au sein d'une même transaction

3. entre la transaction demandant le verrouillage et des méthodes non−transactionnelles concurrentes

De manière analogue, les règles de marquage doivent permettre la détection de conflits entre la méthode non−transactionnelle effectuant le marquage et une transaction concurrente ou une autre méthode non−transactionnelle concurrente.

Enfin, les règles de verrouillage et de marquage doivent assurer l'enchaînement des transactions et des méthodes non−transactionnelles, par rapport aux objets atomiques qu'elles partagent (voir IV.3.2).

Les verrous ont les attributs suivants :

- **propriétaire** : la transaction pour le compte de laquelle l'objet se trouve verrouillé à un instant donné. L'attribut propriétaire, d'un verrou qui a été acquis par une sous−transaction T, change lors de la validation de T, et lors de la validation des supérieurs de T.
- **mode** : *lire* ou *écrire* ;
- **état** : *détenu* (au moment du verrouillage) ou *retenu* (après la validation de la transaction qui a effectué le verrouillage).
- **activité** : l'identité de l'activité qui exécute la méthode.

Les marques n'ont ni propriétaire, ni état ; elles sont uniquement caractérisées par le **mode** d'utilisation de l'objet et par l'identité de l'**activité** qui exécute la méthode non−transactionnelle.

IV.4.1 Les règles de verrouillage étendues

Les règles de verrouillage **RVn**, $n \in [1,6]$, présentées dans la suite, étendent les règles de verrouillage de Moss, **Rn**, $n \in [1,4]$ (voir IV.2.2) de la façon suivante :

− Les règles **RV1** et **RV2** sont des extensions, respectivement, des règles **R1** et **R2**. La **Règle de compatibilité I**, intégrée dans les règles **R1** et **R2**, est remplacée par la **Règle de compatibilité III**, qui prend en compte les différentes formes de concurrence au sein d'une transaction. Les règles **RV1** et **RV2** intègrent aussi une nouvelle règle de compatibilité prenant en compte les conflits de type transaction/méthode non−transactionnelle. Il s'agit de la **Règle de compatibilité IV**, définie dans la suite.

− La règle **RV3** étend la règle **R3**. Cette règle spécifie le traitement des verrous lors de la validation d'une transaction (héritage ascendant).

− La règle **RV4** reproduit la règle **R4**. Elle spécifie le traitement des verrous lors de l'abandon d'une transaction.

− La règle **RV5** spécifie le traitement des verrous lors de la terminaison d'une méthode transactionnelle. Elle est nécessaire pour le support du parallélisme de type transaction/transaction.

− La règle **RV6** définit la sémantique de l'héritage descendant des verrous.

L'énoncé des règles est suivi par des commentaires sur le passage des règles de verrouillage de Moss aux règles de verrouillage de notre modèle.

Règles de verrouillage étendues

Soit Mt une méthode transactionnelle appelée sur l'objet atomique O, pour le compte de la transaction T, et exécutée par une activité Act.

RV1 : Mt peut verrouiller l'objet O en lecture si et seulement si :

1. aucune transaction différente de T ne *détient* un verrou en écriture sur le même objet ;
2. toutes les transactions qui *retiennent* des verrous en écriture sur l'objet sont des ancêtres de T ;
3. dans le cas où l'objet est marqué en écriture, alors l'attribut *activité* de la marque doit être égal à Act.

RV2 : Mt peut verrouiller l'objet O en écriture si et seulement si :

1. aucune autre transaction différente de T ne *détient* un verrou, dans un mode quelconque, sur le même objet ;
2. toutes les transactions qui *retiennent* des verrous sur l'objet sont des ancêtres de T ;
3. dans le cas où l'objet est marqué, alors l'attribut *activité* de la marque doit être égal à Act.

RV3 : Lorsque la transaction T est validée, sa mère (si elle existe) hérite de tous les verrous de T (elle devient la propriétaire des verrous *détenus* et des verrous *retenus* par T). Les verrous hérités restent dans le même mode ; leur état change en *retenu* dans le cas où ils étaient dans l'état *détenu* avant la validation de T.

Une transaction racine relâche tous ses verrous (qu'ils soient détenus ou retenus) lorsqu'elle est validée.

RV4 : Lorsque la transaction T est abandonnée, tous ses verrous sont relâchés.

RV5 : Soit V le verrou posé sur O. Lorsque Mt se termine, l'état de V passe en *retenu* dans le cas où Mt n'a pas créé de transaction.

RV6 : Soit V le verrou posé sur O. T peut offrir V à ses sous−transactions. L'état de V passe alors dans *retenu*. Si l'offre est *contrôlée* (cf. IV.3.3), T prend un verrou V' dans un mode dégradé par rapport au mode de V.

Commentaires

Les règles de verrouillage peuvent être regroupées en trois catégories. Celles (**RV1**, **RV2**) qui assurent la détection des conflits, celles (**RV3**, **RV4**, **RV5**) qui assurent l'enchaînement des transactions et des méthodes non−transactionnelles, et celle (**RV6**) qui fournit le support pour l'héritage descendant.

Détection des conflits

Les conditions 1 − 3 dans chacune des règles **RV1** et **RV2** couvrent les différents types de conflits qui peuvent être engendrés par les différentes formes de parallélisme. La détection des conflits se fait à l'aide des attributs des verrous et des marques.

La condition 1 traite le cas général de conflit entre deux transactions. La condition 2 est nécessaire pour le support du parallélisme transaction/descendant. Elle permet aussi bien l'exécution concurrente d'une transaction avec ses supérieurs, que l'exécution concurrente des méthodes au sein d'une même transaction. Enfin, la condition 3 sert à la détection des conflits de type transaction/méthode non−transactionnelle. Ce dernier type de conflit est ignoré dans le cas où une méthode transactionnelle est emboîtée dans la méthode non−transactionnelle avec laquelle elle se trouve en conflit. Plus concrètement, lorsqu'une méthode non−transactionnelle Mn, appliquée à un objet atomique O, invoque (directement ou indirectement) une méthode transactionnelle Mt sur le même objet O, alors Mt et Mn ne sont jamais considérées en conflit[(3)]. La condition 3 correspond à une règle de compatibilité supplémentaire, formulé comme suit :

Règle de compatibilité IV

Soit une méthode transactionnelle qui essaie de verrouiller un objet atomique ayant déjà été marqué dans un mode conflictuel.

---

(3) Conformément au modèle d'exécution de Guide, deux requêtes qui sont emboîtées et qui sont invoqués sur le même objet atomique sont exécutés par la même activité.

La demande de verrouillage est acceptée si, et seulement si, l'activité qui exécute la méthode transactionnelle est la même que l'activité qui a exécuté la méthode non−transactionnelle ayant marqué l'objet.

Enchaînement des transactions et des méthodes transactionnelles

L'ordonnancement des transactions est garanti par l'héritage ascendant des verrous, mécanisme intégré dans la règle **RV3**, ainsi que par le déverrouillage des objets atomiques (**RV4**) lors de la validation d'une transaction racine ou plate, et lors de l'abandon d'une transaction quelconque.

La règle **RV5** garantit l'ordonnancement des méthodes transactionnelles s'exécutant au sein d'une même transaction. Considérons deux méthodes concurrentes, Mt' et Mt, invoquées dans l'ordre sur l'objet atomique O pour le compte de la transaction T. Pour que Mt puisse verrouiller l'objet O selon la condition 2 de la règle **RV2**, il est nécessaire que le verrou posé sur O devienne *retenu* lors de la terminaison de la méthode Mt'. La Fig. 4.6 illustre la programmation et l'exécution de telles méthodes (dans cet exemple, l'appel O.M1 correspond à la méthode Mt' et l'appel O.M2 correspond à la méthode Mt). Il faut noter que la condition nécessaire pour que les méthodes Mt et Mt' soient exécutées pour le compte de la même transaction T est qu'elles ne créent pas de transaction. C'est justement cette condition qui est prise en compte dans la règle **RV5**.

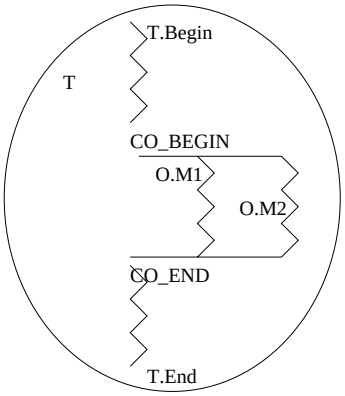

Fig. 4.6 : La programmation et l'exécution de méthodes concurrentes au sein d'une transaction

IV.4.2 Les règles de marquage

Le marquage d'un objet atomique est nécessaire pour détecter si une méthode non−transactionnelle invoquée sur cet objet n'est pas en conflit avec des transactions ou avec d'autres méthodes non−transactionnelles concurrentes utilisant le même objet. L'opération de démarquage, effectuée lors de la terminaison de toute méthode non−transactionnelle, assure l'enchaînement des méthodes non−transactionnelles et des transactions avec lesquelles elles partagent un objet atomique. Elle assure aussi l'ordonnancement des méthodes non−transactionnelles concurrentes invoquées sur le même objet cible.

Règles de marquage

RM1 : Une méthode non−transactionnelle peut marquer un objet en lecture si et seulement si :

1. aucune transaction ne détient ni ne retient de verrou en écriture sur le même objet ;
2. si l'objet est marqué en écriture, alors le marquage a été effectué par l'activité courante ;

RM2 : Une méthode non−transactionnelle peut marquer un objet en écriture si et seulement si :

1. aucune transaction ne détient ni ne retient un verrou sur le même objet ;
2. si l'objet est marqué, alors le marquage a été effectué par un l'activité courante ;

RM3 : Lorsqu'une méthode non−transactionnelle appliquée à un objet atomique se termine, l'objet cible est démarqué.

Commentaires

La détection des conflits est assurée par les règles **RM1** et **RM2**. Dans chacune de ces règles :

− la condition 1 est nécessaire pour le support du parallélisme de type méthode non−transactionnelle/transaction. Cette condition, combinée avec les règles de verrouillage **RV3** et **RV4**, garantit qu'une méthode non−transactionnelle R, invoquée sur un objet atomique O et qui entre en conflit avec une transaction T, est

ordonnancée avec un ascendant de T. En effet, conformément aux règles **RV3** et **RV4**, O reste verrouillé jusqu'à la validation de la racine de T, ou jusqu'à l'abandon d'un des ancêtres de T.

− la condition 2 est nécessaire pour le support du parallélisme de type méthode non−transactionnelle/méthode non−transactionnelle : elle détecte les conflits entre les méthodes non−transactionnelles. Cependant, un tel conflit peut être toléré dans le cas où une méthode non−transactionnelle est emboîtée dans la méthode non−transactionnelle avec laquelle elle est en conflit(3). Plus concrètement, lorsqu'une méthode non−transactionnelle R', appliqué à un objet atomique O, invoque (directement ou indirectement) une méthode non−transactionnelle R sur le même objet O, alors R et R' ne sont jamais en conflit. Ceci correspond à la règle de compatibilité suivante, qui a été intégrée dans la condition 2 des règles **RM1** et **RM2**.

Règle de compatibilité V

> Soit une méthode non−transactionnelle R, qui essaie de marquer un objet atomique ayant déjà été marqué dans un mode conflictuel.
>
> La demande de marquage est acceptée si, et seulement si, l'activité qui exécute R est la même que l'activité ayant marqué l'objet.

IV.5 Éléments de réalisation

La mise en œuvre du modèle de contrôle de la concurrence est assurée en partie par des classes spécialisées et en partie par des mécanismes système.

La classe spécialisée *Verrou* met œuvre les objet *verrou* et *marque*. La classe *Atomic* fournit les mécanismes nécessaires pour la mise en œuvre des règles de verrouillage étendues et des règles de marquage. Ces mécanismes s'appuient sur des mécanismes de plus bas niveau, implantés dans le noyau du système Guide.

IV.5.1 Les mécanismes implantés par les classes spécialisées

IV.5.1.1 La gestion des verrous et des marques

La gestion des verrous et des marques est réalisée localement, au niveau de chaque objet atomique. Les objets atomiques détiennent un attribut appelé *Liste-Contrôle*, hérité de la classe Atomic, qui sert à enregistrer les verrous et les marques posés sur un objet atomique. L'opération de verrouillage, ainsi que celle de marquage, consiste à rajouter un élément dans la liste. L'opération de déverrouillage consiste à enlever de la liste tous les verrous posés par une certaine transaction, et l'opération de démarquage consiste à enlever la marque posée par une méthode non−transactionnelle.

IV.5.1.2 Les verrous

Un verrou est une instance de la classe *Verrou*. Cette classe fournit les attributs nécessaires pour la caractérisation des verrous conformément à IV.4 : *Propriétaire* (la référence de l'objet transaction (cf. III.3.1.1) qui détient ou retient le verrou), *État* (l'état du verrou), *Mode* (le mode de verrouillage) et *Activité* (l'identificateur de l'activité qui exécute le verrouillage). Lors de la création d'un verrou, le mode d'accès est reçu en paramètre. L'attribut *État* est initialisé par la valeur *détenu*. Les valeurs des attributs *Propriétaire* et *Activité* définissent le contexte d'utilisation de l'objet atomique. Elles sont est déterminées à l'aide des mécanismes implantés au niveau système (voir IV.5.2).

La principale méthode fournie par la classe *Verrou* est *ConflitMode*. Son rôle est de détecter les conflits sur la base du mode d'accès, conformément à la politique de synchronisation "un seul écrivain, plusieurs lecteurs". La méthode *ConflitMode* est appelée par la méthode *Contrôle* (comme détaillé sous IV.5.1.4).

IV.5.1.3 Les marques

Pour des raisons de simplicité, les marques sont des instances de la classe *Verrou*. La principale différence entre un verrou et une marque réside dans la valeur de l'attribut *Propriétaire*. Cet attribut vaut la référence nulle pour les marques, ce qui indique le fait que le contexte d'exécution est non−transactionnel. D'autre part, la valeur de l'attribut *État* n'est pas significative pour les marques, mais seulement pour les verrous.

IV.5.1.4 Le verrouillage et le marquage

Les opérations de verrouillage et de marquage sont mises en œuvre par la méthode *Contrôle* de la classe *Atomic*. Cette méthode prend en paramètre un objet de la classe *Verrou* et essaie d'ajouter cet objet à la *ListeContrôle*. Il y a deux cas de figure :

I. Si l'attribut *Propriétaire* de l'objet reçu en paramètre est une référence non−vide, alors l'opération effectuée par *Contrôle* correspond au verrouillage de l'objet cible. L'opération de verrouillage met en œuvre les Règles de verrouillage étendues **RV1** et **RV2** ; elle vérifie s'il n'y a pas incompatibilité entre le verrou reçu en paramètre et les éléments de la *ListeContrôle*. Tout d'abord, en utilisant la méthode *ConflitMode* fournie par la classe *Verrou*, on identifie les conflits dus au mode d'accès. Dans le cas où un tel conflit est détecté, les **Règles de compatibilité III** et **IV** sont utilisées. Si une des règles est vérifiée, le verrouillage est accepté. Sinon, le verrouillage n'est pas possible.

II. Si l'attribut *Propriétaire* de l'objet reçu en paramètre est une référence vide, la méthode *Contrôle* effectue le marquage de l'objet cible. Elle met en œuvre les

Règles de marquage **RM1** et **RM2**, en vérifiant s'il n'y a pas incompatibilité entre la marque reçue en paramètre et les éléments de *ListeContrôle*.

La méthode *Contrôle* reçoit un deuxième paramètre, qui est le nombre d'essais de verrouillage/marquage successifs à effectuer, dans le cas où l'opération de verrouillage ou de marquage échouerait. Le résultat retourné par la méthode *Contrôle* indique si l'opération a réussi.

Les **Exemples 4 − 7** de la III.3.1.3 illustrent les utilisations de la méthode *Contrôle* dans la programmation des objets atomiques.

IV.5.1.5 Le déverrouillage

Le déverrouillage consiste à enlever les verrous posés pour le compte d'une transaction. Il est effectué lors de l'abandon d'une (sous−)transaction, ou à la validation d'une transaction racine. Cet opération met en œuvre les règles de verrouillage étendues **RV3** (en partie) et **RV4.**

L'opération de déverrouillage, implantée par la méthode *Déverrouillage* de la classe *Atomic*, est invoquée lors de l'exécution du protocole de terminaison d'une transaction. Elle reçoit en paramètre la référence de la transaction en cours de terminaison (soit T), et parcourt la *ListeContrôle* de l'objet atomique en enlevant de cette liste tous les verrous dont le propriétaire est T.

IV.5.1.6 Le démarquage

L'opération de démarquage consiste à enlever la marque posée sur un objet atomique par une méthode non−transactionnelle, lors de la fin de la méthode. Elle est implantée par la méthode *Libération* de la classe *Atomic*, méthode qui met en œuvre la règle de marquage **RM3**. Cette méthode doit être appelée explicitement à la fin de toute méthode d'une classe atomique qui ne crée pas de transaction. En effet, une telle méthode est susceptible d'être non−transactionnelle (voir programmation méthode M2 présentée dans III.3.3).

IV.5.1.7 L'enchaînement des méthodes transactionnelles concurrentes

L'enchaînement des méthodes transactionnelles concurrentes qui s'exécutent pour le compte d'une même transaction, est assuré par la règle de verrouillage **RV5**. Étant donné que la méthode *Libération* spécifiée ci−dessus est appelée à la terminaison de toute méthode qui ne crée pas de transaction, la règle de verrouillage **RV5** peut être facilement mise en œuvre par *Libération*.

Cette mise en œuvre commence par déterminer si le contexte courant est transactionnel ou non. Si c'est le cas, l'objet reçu en paramètre est un verrou, et son état

doit passer en *retenu*. Sinon, l'objet reçu en paramètre est une marque, qui doit être enlevée de la *ListeContrôle*.

Nous obtenons l'algorithme suivant pour la méthode *Libération* :

```
si contexte_transactionnel alors
    /* règle RV5 */
    verrou.État <– retenu
sinon
    /* règle RM3 */
```

IV.5.1.8 La propagation des verrous vers le haut (ou l'héritage ascendant)

L'héritage ascendant des verrous est nécessaire lors de la validation d'une sous−transaction. L'héritage ascendant est implantée par la méthode *Propagation* de la classe *Atomic*, qui met en œuvre la règle de verrouillage étendue **RV3**.

La méthode *Propagation* est invoquée sur un objet atomique (qui a été utilisé par une sous−transaction T en cours de validation) et prend en paramètre la référence de l'objet T. La *ListeContrôle* de l'objet atomique est parcourue pour mettre à jour l'attribut *Propriétaire* des verrous appartenant à T. Le nouveau propriétaire est la mère de T (identité de la mère d'une transaction peut être facilement déterminée, car cette information fait partie de son état, voir III.3.1.1). En même temps, l'attribut *État* des verrous hérités devient *retenu*. Finalement, les verrous sont transférés de la liste de T dans la *ListeContrôle* de la transaction mère.

```
pour_tout v en ListeContrôle faire
    si v.Propriétaire == T alors
        v.Propriétaire <– T.Mère
        v.État <– retenu
        Liste.enlever(v)
        ParentList.rajouter(v)
```

IV.5.1.9 L'héritage descendant

L'héritage descendant est implanté par la méthode *Offrir* de la classe *Atomic.* Appliquée à un objet atomique, la méthode *Offrir* prend en paramètre le verrou posé sur cet objet, et dont l'état doit passer en retenu. L'**Exemple 2** de IV.3.3 peut être programmé de la manière suivante :

```
METHOD Traitement (paramètres) ;
 v1 : REF Verrou ;
 OK : Boolean ;
BEGIN
    T.BEGIN
        v1 := new Verrou(READ) ;
        OK := SELF.Contrôle(v1, 5) ;
        IF OK THEN
```

```
            O_Suivant := SELF.Next ;
            /* Le verrou posé est offert */
            SELF.Offrir(v1);
            CO_BEGIN
              O_Suivant.Traitement (paramètres) ;
              SELF.MiseÀJour (paramètres) ;
            CO_END ;
          ELSE
            ...
      T.END ;
  END traitement ;
```

La méthode *Offr*ir a une deuxième version, *OffrirContrôle*, qui permet de contrôler le mode d'accès des inférieurs à l'objet offert. Cette méthode crée un nouveau verrou dans un mode dégradé par rapport au mode du verrou offert, et rajoute ce verrou à la *ListeContrôle*. Elle change ensuite en *retenu* l'état du verrou offert.

IV.5.2 Les mécanismes système

Le contrôle de la concurrence d'accès aux objet atomiques est réalisé par les opérations de verrouillage et de marquage. Ces opérations sont basées sur les concepts de transaction courante − pour le verrouillage, et d'activité courante − pour le verrouillage et le marquage. Nous avons apporté des modifications au système Guide pour la gestion des transactions en cours d'exécution. Des nouvelles fonctions système permettent d'enregistrer les transactions créées et de déterminer les transactions pour le compte desquelles les méthodes transactionnelles sont exécutées. Enfin, une autre fonction système permet de déterminer l'identité de l'activité exécutant une méthode invoquée sur un objet atomique.

IV.5.2.1 Le concept de transaction courante

Dans un système distribué et multi−activités tel que Guide, les activités sont les entités les plus appropriées pour véhiculer l'information qui identifie la transaction courante. Dans notre cas, il s'agit de la référence de l'objet transaction qui représente la transaction courante. Par défaut, les applications Guide ne sont pas transactionnelles ; lors de la création de l'activité initiale d'une application Guide il n'y a pas de transaction courante et la référence nulle est enregistrée auprès de l'activité initiale. Par conséquent, tout appel de méthode sur un objet atomique s'exécute en tant que méthode non−transactionnelle. Dès l'instant où une transaction est créée et lancée, elle devient la transaction courante et la référence de la transaction est enregistrée auprès de l'activité en cours. Si l'activité s'étend sur d'autres sites, les extensions d'activité héritent de cette référence, ce qui permet à une transaction répartie de verrouiller et d'utiliser des objets qui se trouvent dans la mémoire d'exécution de différents sites. Il s'ensuit que toute méthode appliquée à un objet

atomique est exécutée pour le compte de la transaction courante, indépendamment de sa localisation. Il faut aussi noter que lors d'une extension d'activité (suite à l'instruction CO_BEGIN), l'activité fille hérite de la transaction courante de sa mère ; en conséquence, si la mère est non−transactionnelle, ses filles sont non−transactionnelles. La transaction courante change lors de la création d'une sous−transaction, ainsi que lors de la terminaison (par validation ou abandon) d'une sous−transaction. Dans ce dernier cas, la mère de la sous−transaction devient la nouvelle transaction courante. Lors de la terminaison d'une transaction racine, la référence enregistrée auprès de l'activité courante redevient nulle.

La fonction système *setcurrent_tr* a pour rôle d'enregistrer l'identificateur de la nouvelle transaction courante lors du début et de la terminaison d'une transaction. Elle est appelée par les méthodes *Begin*, *Commit* et *Abort* de la classe Transaction. Cette fonction enregistre dans le descripteur de l'activité courante la référence de l'objet transaction reçue en paramètre (*TransId*). Une deuxième fonction système, appelée *getcurrent_tr*, permet de déterminer l'identité de la transaction courante en retournant le *TransId* enregistré dans le descripteur de l'activité courante. Cette fonction est appelée lors du verrouillage/marquage d'un objet atomique. La référence de la transaction courante retournée sert à initialiser l'attribut *Propriétaire* du verrou ou de la marque.

IV.5.2.2 Le concept d'activité courante

Nous avons besoin de remonter l'identité de l'activité courante au niveau applicatif pour le support des différentes formes de parallélisme. Pour la mise en œuvre des règles de verrouillage étendues et des règles de marquage, les verrous et les marques doivent être caractérisés par l'activité courante.

Nous avons donc implanté la fonction système *getcurrent_act* qui retourne l'identificateur de l'activité courante. Cette fonction est appelée lors de l'initialisation des verrous et des marques.

IV.6 Conclusion

Le modèle de contrôle de concurrence proposé et présenté ci−dessus garantit la cohérence des objets atomiques partagés par des opérations transactionnelles et par des opérations non−transactionnelles. Ce modèle répond aux nécessités induites par la souplesse du service transactionnel Guide. L'approche de mise en œuvre, à base d'objets et de classes spécialisées, facilite le développement des applications utilisant des objets atomiques et des transactions. Cette approche a aussi l'avantage de faciliter la mise en œuvre du service transactionnel ; à l'exception

des mécanismes système présentés en IV.5.2, la mise en œuvre du contrôle de la concurrence est réalisée à l'aide des outils de l'environnement Guide, sans modifier cet environnement.

Le modèle présenté peut être envisagé comme une extension du modèle des transactions emboîtées de Moss. Il permet non seulement les différentes formes de parallélisme intra−transactionnel, mais aussi le parallélisme entre méthodes non−transactionnelles utilisant des objets atomiques, ainsi qu'entre transactions et méthodes non−transactionnelles. La cohérence des objets atomiques est garantie par les opération de verrouillage ou de marquage, selon leur contexte d'utilisation : transactionnel, respectivement non−transactionnel.

Dans le modèle de Moss, les règles de verrouillage intègrent le concept d'héritage ascendant des verrous, ainsi que la règle de visibilité permettant à une sous−transaction de verrouiller les objets déjà utilisés par ses ancêtres (ce qui revient à la règle de compatibilité entre une sous−transaction et ses ancêtres). Tout en gardant l'héritage ascendant des verrous, qui garantit l'isolation des arbres transactionnels, nous avons intégré dans les règles de verrouillage des nouvelles règles de compatibilité pour permettre différentes formes de parallélisme. Nous avons aussi introduit des règles de marquage, qui prennent en compte la concurrence entre transactions et non−transactions ou entre les méthodes non−transactionnelles. Finalement, le modèle proposé intègre l'héritage descendant des verrous, afin de permettre plus de souplesse dans la structuration des applications en termes de (sous−)transactions.

Parmi les systèmes qui ont adopté des modèles étendus de contrôle de concurrence, on peut citer Argus et Arjuna. Leurs modèles transactionnels fournissent un support pour le parallélisme mère−fille. Dans le système LOCUS, le modèle transactionnel offre l'héritage descendant des verrous, mais sans assurer un contrôle sur l'utilisation des verrous hérités par les sous−transactions. Enfin, en Hermes/ST, le modèle transactionnel assure le support pour toutes les formes de parallélisme présentées, sans offrir cependant l'héritage descendant des verrous.

Le modèle étendu proposé ici présente l'avantage d'augmenter le parallélisme sans pour autant sacrifier la cohérence des données. De plus, les extensions apportées facilitent la structuration des applications à l'aide des transactions emboîtées.

Chapitre V

L’atomicité

V.1 Introduction

Le présent chapitre, ainsi que le suivant, sont consacrés aux mécanismes qui préservent l’atomicité des transactions et des opérations non−transactionnelles appliquées aux objets atomiques, en présence des événements de types suivant :

1. L’abandon d’une transaction à la demande de l’application.
2. L’interruption de l’application pendant l’exécution d’une transaction, suite à une erreur logicielle. À l’opposé du cas précédant, il s’agit d’un abandon de transaction qui n’est pas programmé.

   L’interruption d’une opération non−transactionnelle appliquée à un objet atomique, suite à une erreur logicielle.
3. L’interruption de l’application pendant l’exécution d’une transaction, ou l’interruption d’une opération non−transactionnelle appliquée à un objet atomique, suite à une panne matérielle (défaillance de site ou défaillance de communication).

L’atomicité des transactions garantit la cohérence *globale* de tous les objets atomiques utilisés pour le compte d’une transaction (cf. III.4.2). L’atomicité des opérations non−transactionnelles appliquées aux objets atomiques garantit la cohérence *locale* des objets atomiques.

Les mécanismes qui garantissent la propriété d’atomicité sont fortement dépendants de la politique de mise à jour de la mémoire, et plus généralement de l’architecture de la mémoire. Nous présentons dans V.2 le principe de fonctionnement de ces mécanismes dans le cadre des contraintes imposées par le système sous−jacent.

La section V.3 décrit la mise en œuvre des mécanismes permettant la prise en compte de la résistance aux pannes des transactions non−réparties. Les extensions nécessaires pour la prise en compte des problèmes spécifiques résultant de la répartition sont présentées dans le chapitre suivant. Enfin, l’atomicité des méthodes non−transactionnelles appliquées aux objets atomiques est abordée dans la section V.4.

V.2 Le modèle d'architecture adopté

Si le modèle de concurrence du STG a été défini de manière à répondre aux besoins des applications (approche descendante), le choix du modèle qui garantit la cohérence des objets atomiques en présence de pannes a été fortement influencé par des contraintes liées à l'architecture du système Guide (approche ascendante). Nous commençons donc par rappeler ces contraintes, afin d'identifier les mécanismes nécessaires pour garantir l'atomicité et la permanence des transactions. Après une présentation globale de ces mécanismes (V.2.2 et V.2), nous nous intéressons aux outils qui garantissent la cohérence locale des objets atomiques (V.2.4).

V.2.1 Les contraintes architecturales

En Guide, toute modification d'objet est effective immédiatement au niveau de la mémoire d'exécution. Les modifications effectuées au niveau de la mémoire d'exécution sont copiées dans la mémoire de stockage de manière différée. Par défaut, l'image d'une grappe(1) présente en mémoire d'exécution n'est mise à jour en mémoire de stockage que lorsqu'elle n'est plus utilisée par aucune application. Ceci est rendu possible par le fait que la mémoire de pagination est physiquement séparée de la mémoire de stockage.

Considérons d'abord le cas d'une transaction qui n'arrive pas à la validation (qui est abandonnée ou interrompue par une panne). Étant donnée la mise à jour immédiate de la mémoire d'exécution, et compte tenu du partage des grappes, il est nécessaire d'annuler (défaire) les effets de la transaction au niveau de la mémoire d'exécution. Le principe de fonctionnement du mécanisme mis en œuvre dans ce but est présenté dans V.2.2. Cependant, étant donnée la séparation entre les mémoires de stockage et de pagination, la mémoire de stockage n'est pas altérée pendant l'exécution d'une application. Par conséquent, il n'y a rien à défaire au niveau de la mémoire de stockage. Ceci est vrai même dans le cas d'une transaction mère ayant des filles validées : la validation d'une sous−transactions étant conditionnelle, elle n'entraîne pas de mise à jour au niveau de la mémoire de stockage.

Concernant les transactions racines qui arrivent au point de terminaison, le mécanisme présenté dans V.2.3 garantit que la validation se fait de manière tout ou rien, et que si la transaction est validée alors ses effets sont permanents.

---

(1) Unité de stockage et de partage qui regroupe des objets (voir II.3.1).

V.2.2 Le principe du mécanisme d'annulation

Cette sous−section identifie les mécanismes nécessaires à l'annulation des effets d'une transaction au niveau de la mémoire d'exécution. Ces mécanismes sont nécessaires lors de l'abandon d'une transaction (l'abandon pouvant être soit programmé, soit la conséquence d'une interruption imprévue).

Les copies−avant

L'approche de mise à jour immédiate de la mémoire d'exécution conduit à la nécessité de gérer des copies d'objets afin de pouvoir annuler les effets d'une transaction.

> Une *copie−avant* est une copie de l'état d'un objet avant qu'il ne soit modifié pour le compte d'une transaction donnée.

Étant donné qu'un objet peut être modifié à plusieurs niveaux d'une transaction emboîtée, plusieurs copies−avant peuvent lui être associés.

Les listes d'intentions

L'annulation des effets d'une transaction dans le STG est basée sur un mécanisme de *listes d'intentions* (ce ne sont que des intentions, car les effets ne sont pas rendus persistants avant que la validation de la transaction soit certaine).

> Une *liste d'intentions* est une structure de données associée à une transaction, qui sert à retrouver les verrous que celle−ci a posé, les objets qu'elle a modifié, ainsi que les copies−avant des objets modifiés.

La constitution des listes d'intentions se fait de manière dynamique, pendant la phase active des transactions. Dans le cas d'une transaction emboîtée, plusieurs composantes peuvent être actives à un instant donné, donc plusieurs listes d'intentions peuvent coexister.

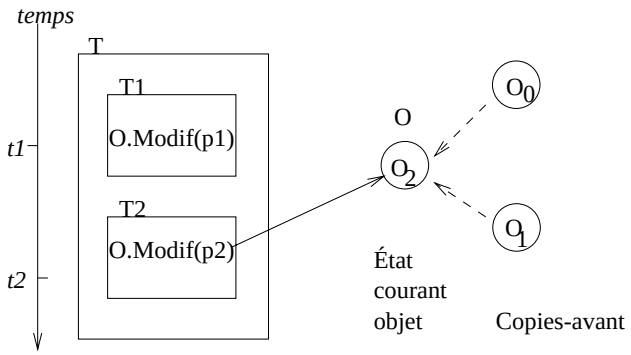

Fig. 5.1 : Copies avant associées à un objet utilisé par une transaction emboîtée

Exemple 1. Soit T une transaction mère de deux sous−transactions T1 et T2 qui modifient dans l'ordre l'objet atomique O. Nous désignons par O_0 l'état initial de l'objet O (l'état avant la modification effectuée par T1), par O_1 l'état après la modification effectuée par T1, et par O_2 l'état après la modification effectuée par T2. La Fig. 5.1 représente l'état de l'objet O et de ses copies−avant, à l'instant t2. L'état O_1 est nécessaire pour le cas où la sous−transaction T2 devrait être annulée, tandis que l'état O_0 est nécessaire pour le cas où T devrait être annulée (T1 est déjà validée à l'instant t2). Ces copies−avant sont gérées par les listes d'intentions associées à la transaction T et à la sous−transaction T2.

V.2.3 Le principe du mécanisme de validation

Le mécanisme qui rend les effets d'une transaction racine visibles de manière atomique garantit aussi la permanence des effets de la transaction. Il s'agit d'un protocole de validation utilisant un journal après, dont voici une brève description :

− La première phase du protocole est dédiée à la construction du journal−après (contenant l'état actualisé des objets modifiés) sur un support de mémoire non−volatile. Les objets modifiés sont retrouvés à l'aide d'informations enregistrées dans la liste d'intentions associée à la transaction. Si cette phase se termine avec succès, la transaction peut être considérée comme *validée* (elle sera validée même en cas de panne par les mécanismes de reprise, sur la base des informations enregistrées dans le journal après).

− La deuxième phase du protocole est dédiée à la mise à jour effective de la mémoire de stockage et au déverrouillage des objets atomiques. Le déverrouillage, effectué en mémoire d'exécution, est nécessaire, car les objets peuvent rester en mémoire après la fin de la transaction.

V.2.4 L'atomicité des méthodes non−transactionnelles

L'utilisation des objets atomiques hors de transactions est une facilité offerte par le STG, qui permet de trouver un meilleur équilibre entre les objectifs de performance et les besoins de cohérence des applications.

Pour assurer de manière efficace la cohérence locale des objets atomiques, nous avons recherché une solution qui minimise les interactions du programme avec le STG. Nous avons donc rejeté l'utilisation des copies−avant pour l'annulation des effets partiels au niveau de la mémoire d'exécution et nous avons décidé de tirer profit des mécanismes déjà offerts par Guide pour le couplage/découplage des grappes. En effet, si le STG devait gérer des copies−avant pour les objets atomiques utilisés hors transaction, ceci reviendrait à exécuter toute opération qui s'applique à un objet atomique comme une transaction.

Le fonctionnement du mécanisme qui assure la cohérence locale des objets atomiques hors transaction est détaillé en V.4.

V.3 L'atomicité des transactions non−réparties

Les mécanismes présentés dans cette section doivent garantir l'atomicité des transactions non−réparties en cas d'abandon de la transaction et en cas de panne de site. Une transaction est non−répartie si tous les objets atomiques qu'elle utilise se trouvent dans la mémoire d'exécution du site où la transaction est lancée. En outre, dans le cas de transactions emboîtées, les sous−transactions doivent être créées et lancées sur le même site que la transaction racine.

Dans un premier temps, nous considérons le cas dans lequel la transaction est abandonnée à la demande de l'application (V.3). Le mécanisme présenté est entièrement basé sur la technique des listes d'intentions. L'extension de ce mécanisme pour la prise en compte des interruptions dues aux erreurs logicielles (ou aux pannes d'application) est présentée dans V.3.2. Nous terminons la section par la prise en compte des pannes de site (V.3.1).

V.3.1 L'abandon d'une transaction

Le mécanisme présenté, basé sur l'utilisation des listes d'intentions, est mis en œuvre en grande partie par des classes spéciales. Après la description des composantes du mécanisme (V.3.1.1), nous présentons les deux aspects de son fonctionnement : la constitution de listes d'intentions (V.3.1.2) et leur utilisation dans le cadre des protocoles de terminaison (V.3.1.3). Le protocole d'abandon des transactions défini en V.3.1.3, est repris en V.3.1.4 afin de démontrer de manière

informelle qu'il assure l'atomicité des transactions abandonnées dans le cas non−réparti.

V.3.1.1 Les composantes

Tout objet transaction (instance de la classe spéciale *Transaction*) possède un attribut appelé *historique* qui met en œuvre la liste d'intentions de la transaction. Un objet historique (instance de la classe spéciale *Historique*) est constitué par un ensemble d'enregistrements (instances de la classe spéciale *Enregistrement)*. Il existe des enregistrements de deux types : de verrouillage et de restauration. Les *enregistrements de verrouillage* servent à retrouver les objets verrouillés par la transaction (donc les verrous qu'elle a posé) et permettent le déverrouillage des objets lors de l'annulation des effets d'une transaction, ou lors de la validation de sa racine. Les *enregistrements de restauration* servent à retrouver les objets modifiés, ainsi que leur copie−avant, et permettent la restauration des objets modifiés par une transaction. Les enregistrements de restauration sont également utilisés pendant la 1ère phase de validation d'une transaction racine, pour la construction du journal−après.

L'existence des objets *historique* et *enregistrement* est cachée aux applications. Ces objets spéciaux sont manipulés par le biais des opérations fournies par les classes spéciales. Il n'est pas nécessaire d'assurer la résistance aux pannes des objets *historique* et *enregistrement*, car les mécanismes de reprise ne les utilisent pas.

La liaison entre les différentes composantes est illustrée dans la figure Fig. 5.2.

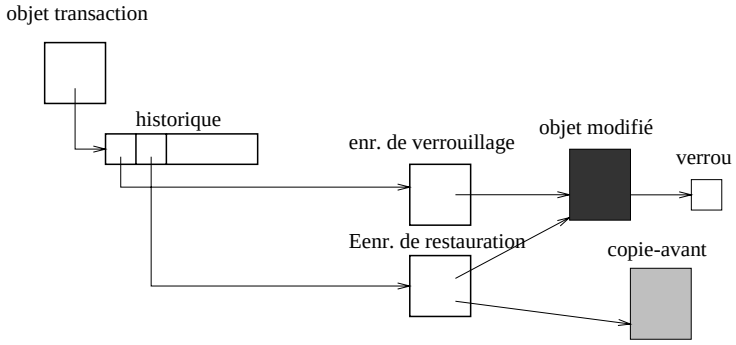

Fig. 5.2 : Les composantes du mécanisme des listes d'intentions

V.3.1.2 La constitution de l'historique

Les informations enregistrées dans l'historique doivent permettre la mise en œuvre des règles de validation et d'abandon des transactions emboîtées, règles énoncées en I.3.1.

L'abandon d'une transaction emboîtée implique l'annulation de ses effets, aussi bien que l'annulation des effets de ses sous−transactions validées (les effets des sous−transactions abandonnées ont déjà été annulés). Il s'ensuit que l'historique doit être composé d'enregistrements associés aux objets que la transaction a elle même utilisés et d'enregistrements associés aux objets utilisés par ses sous−transactions validées. En conséquence, lors de la validation d'une sous−transaction son historique est intégré à celui de sa mère. On dit que la mère hérite de l'historique de sa fille, en même temps qu'elle hérite des verrous posés par cette fille.

L'interaction avec le mécanisme de contrôle de la concurrence

Les objets utilisés pour le compte d'une transaction sont enregistrés dans son historique grâce au mécanisme de contrôle de concurrence. Conformément à III.3.1, l'utilisation effective d'un objet atomique par une transaction est conditionnée par l'issue de l'opération *Contrôle* qui tente de verrouiller l'objet pour le compte de la transaction. Un autre rôle de la méthode *Contrôle* est de créer, si nécessaire, des enregistrements associés à l'objet et de les insérer dans l'historique de la transaction courante. Un enregistrement de verrouillage est créé si la transaction utilise l'objet pour la première fois dans le mode d'accès spécifié. Si le mode d'accès est "en écriture", et s'il s'agit de la première utilisation de l'objet dans ce mode, un enregistrement de restauration est créé. De plus, la création d'une copie−avant est demandée au système (voir la description de la fonction système *doCopy* V.3.2.4) ; son adresse est stockée dans l'enregistrement de restauration.

La participation du mécanisme de validation des sous−transactions

Ce mécanisme fonctionne sur la base d'un algorithme qui permet à la transaction mère d'hériter de l'historique de sa fille. Cet algorithme est mis en œuvre par la méthode *Validation* de la classe *Transaction*.

Algorithme de validation d'une sous−transaction

Lors de la validation d'une sous−transaction, les enregistrements se trouvant dans son historique sont insérés dans l'historique de sa mère, à condition que les règles suivantes soient vérifiées :

• Pour tout *enregistrement de restauration* :

L'objet atomique référence doit ne pas avoir déjà été modifié par la transaction mère ou par l'une de ses filles validées. En cas contraire, l'enregistrement de restauration est détruit, car il existe dans la liste d'intentions de la mère un enregistrement de restauration qui référence le même objet atomique, ainsi qu'une copie−avant de cet objet qui est antérieure à celle détenue par la fille (voir la Fig. 5.3 et les commentaires).

• Pour tout *enregistrement de verrouillage* :

L'objet atomique référencé n'a pas encore été utilisé par la transaction mère, ou par une de ses filles validées. En cas contraire, le mode d'utilisation de la sous−transaction doit être plus restrictif que celui de la mère.

La Fig. 5.3 développe l'**Exemple 1** illustré dans la Fig. 5.1, en mettant en évidence les listes d'intentions associées aux transactions T et T2, qui sont actives à l'instant t2. Les copies−avant associées à l'objet O peuvent être retrouvées à l'aide des enregistrements E1 et E2. E1, créé lors de la première modification de O, a été inséré dans l'historique de la transaction T lors de la validation de T1. E2 a été inséré dans l'historique de T2 lors de la deuxième modification de O. Si T2 est validée, E2 n'est pas inséré dans l'historique de T, conformément aux règles de ci−dessus.

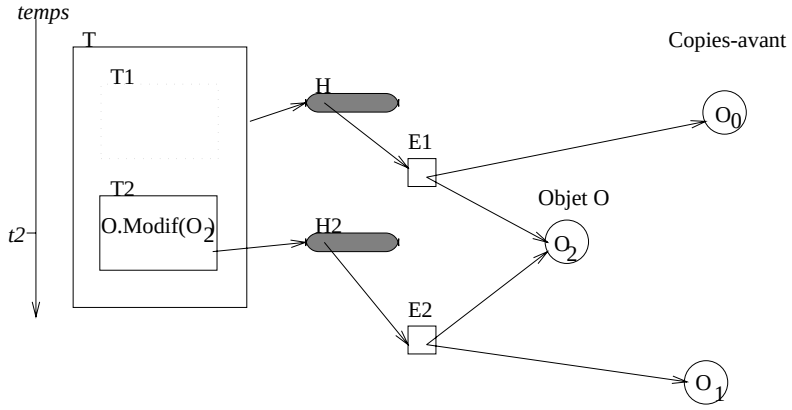

Fig. 5.3 : Les listes d'intention associés à une transaction emboîtée

Éléments de mise en œuvre

L'opération qui effectue l'insertion d'un enregistrement dans la liste d'intentions d'une transaction est mise en œuvre par une méthode, dénommée *Ajout*, fournie par la classe *Transaction*. Cette méthode est utilisée par le mécanisme de contrôle

de la concurrence et par le mécanisme de validation des sous−transactions. Elle prend en paramètre la référence de l'objet enregistrement a insérer dans l'historique de l'objet transaction cible.

V.3.1.3 L'utilisation de l'historique

Les algorithmes de terminaison mis en œuvre par la classe *Transaction* fonctionnent sur la base d'informations enregistrées dans l'historique. La validation d'une sous−transaction a été présentée dans V.3.1.2, car elle fait partie intégrante du mécanisme de constitution des listes d'intentions. Voici maintenant l'algorithme de validation d'une transaction racine.

Algorithme de validation d'une transaction racine

Phase 1. Pour tout enregistrement de restauration présent dans l'historique, la journalisation de l'objet référencé est demandée au système.

Phase 2−validation (exécutée si la phase 1 s'est terminée avec succès). Pour tout enregistrement de restauration présent dans l'historique, la sauvegarde de sa copie après est demandé au système.

Phase 2−annulation. Pour tout enregistrement de restauration présent dans l'historique, l'état de l'objet référencé est restauré sur la base de sa copie−avant. Pour tout enregistrement de verrouillage présent dans l'historique, l'objet référencé est déverrouillé.

L'abandon d'une transaction (ou d'une sous−transaction) est effectué selon l'algorithme suivant :

Algorithme d'abandon d'une transaction

- Pour tout enregistrement de restauration présent dans l'historique, l'état de l'objet référencé est restauré sur la base de sa copie−avant.
- Pour tour enregistrement de verrouillage présent dans l'historique, l'objet référencé est déverrouillé.

Éléments de mise en œuvre

− La journalisation et la sauvegarde des objets atomiques, effectuées par des mécanismes implantés au niveau des serveurs de stockage du système Guide, ne sont pas détaillés dans cette thèse. Les méthodes *Préparation* et *Sauvegarde*, fournies par la classe *Atomic*, constituent l'interface de ces mécanismes. D'autre part, les méthodes *Préparation* respectivement *Sauvegarde* sont appelées sur

l'objet atomique à préparer ou à sauvegarder par les méthodes *PrepareEffet* et *ValidEffet* fournies par la classe *Enregistrement*.

− Le mécanisme de restauration d'un objet atomique est implanté au niveau du noyau Guide (la fonction système *doRestore* en V.3.2.5). La classe *Enregistrement* fournit la méthode *AbandonEffet*, qui, lorsqu'elle est appliquée à un enregistrement de restauration, appelle la fonction *doRestore*.

− Le déverrouillage d'un objet atomique est mis en œuvre par la méthode *Déverrouillage* fournie par la classe *Atomic* (voir IV.5.1.5). Cette méthode est invoquée lorsque la méthode *AbondonEffet* est appliquée à un enregistrement de verrouillage.

− Les algorithmes de validation et d'abandon présentés ci−dessus sont mis en œuvre par les méthodes *ValidEffectifTop* − pour la validation d'une transaction racine, *ValidEffectifNested* − pour pour la validation d'une sous−transaction, et *AbandonEffectif*. Les méthodes de *Validation* et d'*Abandon* fournies par la classe *Transaction* sont réduites alors à l'appel de ces méthodes et au changement de la transaction courante (cf. IV.5.2.1) :

```
METHOD Validation ;
BEGIN
    IF  Mère  =  Nil  THEN  /*  l'attribut  Mère  contient  la
                référence de la transaction mère */
        SELF.ValifEffectifTop
    ELSE
        SELF.ValifEffectifNested
    END ;
    BEGIN_C
        /* appel fonction système setcurrent_tr */
    END_C ;
END Validation ;

METHOD Abandon ;
BEGIN
    SELF.AbandonEffectif ;
    BEGIN_C
        /* appel fonction système setcurrent_tr */
    END_C ;
END Abandon ;
```

V.3.1.4 L'atomicité des transactions abandonnées

L'algorithme d'abandon mis en œuvre par la méthode *Abandon* présentée ci−dessus garantit l'annulation des effets de la transaction, si les conditions suivantes sont vérifiées :

- toutes les sous−transactions sont terminées (validées ou abandonnées) ;
- le mécanisme qui met en œuvre l'insertion d'un enregistrement dans un objet historique est fiable.

La première condition est nécessaire pour assurer l'annulation des effets des sous−transactions validées. Elle est toujours vraie si les restrictions de programmation des méthodes transactionnelles (III.3.2.1) sont respectées.

La deuxième condition est nécessaire pour assurer la correction de l'algorithme d'abandon (elle garantit que la liste d'intentions est complète). Dans le cas des transactions non−réparties, l'utilisation du mécanisme d'appel de méthode pour la mise à jour de l'historique est fiable, car les objets atomiques utilisés par une transaction, ainsi que les enregistrements qui leur sont associés, se trouvent sur le même site que l'objet transaction et son historique.

V.3.2 Résistance aux pannes d'application

Une erreur de programmation, telle qu'une division par zéro ou l'utilisation d'une référence invalide, entraîne l'arrêt de l'application. On s'intéresse ici au cas où une telle défaillance se produit pendant l'exécution d'une transaction. Nous faisons l'hypothèse que l'événement peut apparaître seulement pendant l'exécution du traitement effectif réalisé pour le compte de la transaction, et non pas pendant l'exécution du code transactionnel (nous considérons que ce code est fiable).

L'analyse comparative entre le cas de panne d'application et celui d'abandon programmé d'une transaction (V.3.2.1) à démontré que le mécanisme des listes d'intentions n'est pas suffisant pour remédier aux pannes d'application. Cette analyse permet d'identifier les mécanismes à rajouter pour masquer ce type de panne.

V.3.2.1 Les caractéristiques des pannes d'application

Une panne d'application dans le système Guide se manifeste par la mort du contexte courant d'exécution et l'interruption de l'exécution de l'application. De même que dans le cas d'une panne de transaction, lorsqu'un tel événement se produit pendant l'exécution d'une application, les effets de la transaction doivent être annulés. Un autre point commun avec les pannes de transaction : il suffit d'annuler ses effets seulement au niveau de la mémoire d'exécution (il n'y a rien à défaire au niveau de la mémoire de stockage). Cependant, les pannes d'application ont des traits spécifiques :

− Dans le cas des pannes de transaction, le traitement de l'annulation (exécution de la méthode *Abandon*) fait partie intégrante de l'application. Ceci n'est pas le cas pour les pannes d'application.

− La détection de la panne. Dans le cas des pannes de transaction, la décision d'abandon est prise par l'application. Dans le cas des pannes d'application, la panne doit être détectée et le traitement de l'annulation déclenché par une entité active extérieure à l'application, appelée *Gérant de Transactions*.

− L'annulation d'une sous−transaction en cas de panne transactionnelle n'entraîne pas l'abandon de ses ancêtres. Dans le cas d'une panne d'application qui survient pendant l'exécution d'une sous−transaction, la transaction entière (ou globale) est abandonnée. Étant donné qu'au moment de la panne plusieurs composantes de la transaction peuvent être actives, l'historique de la transaction se trouve éparpillé entre les listes d'intentions associées à ces différentes composantes. Si par exemple, une panne d'application survient pendant l'exécution de la sous−transaction T2 représentée dans la Fig. 5.3, l'historique de la transaction globale est enregistré dans les listes d'intentions H et H2.

− Une panne d'application peut avoir comme conséquences la perte d'informations constituant l'historique d'une transaction. En effet, la mort d'un contexte implique la disparition de la mémoire d'exécution des grappes non−partagées couplées dans ce contexte. Si une grappe contenant un objet historique disparaît, l'annulation des effets enregistrés dans cet historique n'est plus possible.

Nous avons mis en place les mécanismes suivants, afin de rendre les transactions Guide résistantes aux pannes d'application :

− Les Gérants de Transactions (GT) (V.3.2.2). Un GT est une entité active, présente sur chaque site Guide, qui gère des informations concernant les transactions en cours d'exécution sur le site afin de pouvoir détecter les transactions affectées par les pannes et d'annuler leur effets.

− Des mécanismes de notification par envoi de messages qui informent le GT sur les événements significatifs concernant les transactions. Ces mécanismes permettant au GT de rassembler des informations sur les transactions sont présentés en V.3.2.3.

− Des mécanismes qui empêchent la perte d'informations constituant l'historique d'une transaction, présentés en V.3.2.4 et V.3.2.5.

V.3.2.2 Le Gérant de Transactions

Chaque GT gère une structure de données, nommée *TransList*, qui enregistre les informations suivantes concernant toute transaction active sur son site :

- *Sup* : identificateur de la transaction mère.
- *InfList* : liste des filles actives de la transaction.
- *CtxList* : liste des contextes d'exécution de la transaction.

La mort d'un contexte est détectée par le démon Guide, qui signale l'événement au Gérant de Transactions (GT) local. Si le contexte disparu fait partie de l'ensemble des contextes d'exécution d'une transaction, alors le GT a détecté une transaction défaillante et déclenche son annulation conformément à l'algorithme présenté dans la suite.

Soit P la transaction défaillante et T la racine de son hiérarchie (si P est racine ou plate, T=P). Pour annuler les effets de T, le GT utilise l'algorithme d'abandon présenté en V.3.1.3 afin d'annuler les effets de chacune de ses composantes actives. Les composantes doivent être traitées dans le sens "feuilles −> racine", ceci afin de tenir compte du cas où un même objet atomique ayant été modifié par plusieurs composantes de la transaction, plusieurs copies−avant associés à l'objet sont conservées. L'objet doit être restauré dans ce cas, sur la base de son état initial. Cet état correspond à la première copie−avant effectuée, qui est enregistrée dans l'historique de la composante la plus haute de la hiérarchie. Par exemple, dans le cas où il faut annuler la transaction T représentée en Fig. 5.3, l'état final de l'objet O doit être O_0, qui est enregistré en H.

Algorithme d'annulation

Soit T la racine de la transaction dont les effets doivent être annulés. Alors Annule(T) effectue :

- Pour tout transaction F appartenant à l'*InfList* de T, appeler Annule(F).
- Effectuer T.Abandon.

Le GT met à jour la *TransList* sur la base des informations qu'il reçoit dans les messages de notification. Nous présentons dans la suite les événements significatifs (qui doivent être signalés au GT), le contenu des messages envoyés et les actions du GT lors de la réception de ces messages.

V.3.2.3 Les messages de notification

Ces messages sont de trois sortes :

- *Lancement d'une transaction*. La méthode *Début* appelle une primitive système qui envoie au GT un message contenant les informations suivantes : la référence de la nouvelle transaction, la référence de sa mère et l'identificateur du contexte courant d'exécution.

Sur la réception de ce message, le GT met à jour la *TransList* : une nouvelle entrée est créée et la transaction est rajoutée à l'*InfList* de la mère.

- *Terminaison d'une transaction.* Les méthodes *Validation* et *Abandon* appellent une primitive système, qui envoie au GT un message pour l'informer sur la terminaison de la transaction courante.

Le GT enlève l'entrée correspondant à la transaction de la *TransList* et met à jour l'*InfList* de la mère (s'il existe une mère).

• *Extension d'une transaction.* Une transaction s'étend dans un nouveau contexte lors de l'extension de l'activité courante (lors de la création d'un représentant d'activité dans un nouveau contexte C, cf. II.3.1) ou lorsqu'une extension d'activité s'exécutant dans un contexte C change de transaction courante[(2)]. Un message est alors envoyé au GT, pour l'informer de l'extension de T dans le contexte C.

Le GT rajoute alors l'identificateur du contexte C à la *CtxList* de T.

V.3.2.4 Fiabilisation de l'historique

Le problème des pertes d'historique est illustré par l'**Exemple 2** :

Exemple 2. Soit T une transaction s'exécutant dans un contexte C qui meurt suite à une panne (voir Fig. 5.4). Supposons que les objets T et H (l'historique associé à T) sont placés dans la grappe G1 et que l'objet atomique O utilisé pour le compte de T se trouve dans la grappe G2. Compte tenu de la politique de placement des objets, l'enregistrement de restauration associé à O est aussi placé dans la grappe G2[(3)].

Soit Gi une grappe couplée dans le contexte C. Si, au moment de la panne, la grappe Gi n'est pas partagée avec d'autres applications s'exécutant sur le site (comme c'est le cas pour G1), alors les modifications effectuées par T au niveau de cette grappe sont perdues. Dans le cas contraire, la grappe reste couplée dans la mémoire (c'est le cas pour G2, qui reste couplée dans le contexte C' d'une autre application) et les modifications apportées par T au niveau de cette grappe restent présentes dans la mémoire.

La disparition de la grappe G1 de la mémoire implique la perte de l'historique H, donc l'impossibilité d'annuler les modifications effectuées au niveau de l'objet O.

---

(2) Cf. à IV.5.2.2, une activité transactionnelle est caractérisée par la transaction pour le compte de laquelle elle s'exécute, appelée transaction courante. Une extension d'activité change de transaction courante lorsqu'on lui demande d'exécuter une méthode pour le compte d'une autre transaction. Cette nouvelle transaction devient alors sa transaction courante.

(3) Un objet est placé lors de sa création dans la grappe de l'objet qui effectue la création, (*ref. Chap II).*

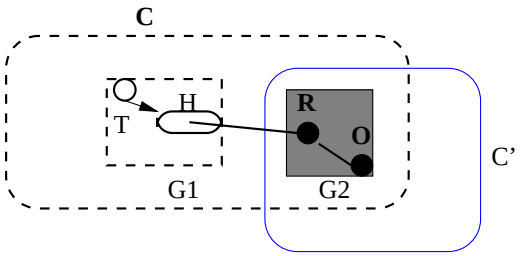

Fig. 5.4 : La mort d'un contexte lors d'une panne d'application

Afin d'éviter les pertes d'historique, nous avons mis en œuvre un mécanisme qui force le couplage des grappes contenant des objets transaction et historique, dans un deuxième contexte appelé *contexte transactionnel* (CT). CT appartient à un propriétaire privilégié, qui est le GT. Le contexte CT est créé lorsque l'application lance pour la première fois une transaction. Par la suite, chaque fois qu'un objet transaction est créé dans une grappe, on force le couplage de cette grappe dans le CT. De cette manière, si le contexte d'exécution meurt, la grappe reste couplée dans le CT, donc disponible pour le GT. Considérons encore l'**Exemple 2** : G1 ne disparaît plus de la mémoire après la mort de C (voir Fig. 5.5).

Ce mécanisme de fiabilisation a donc nécessité la modification du mécanisme de couplage : nous avons introduit une exception à la règle par défaut, qui stipulait qu'une grappe peut être couplée dans un contexte seulement si la grappe et le contexte ont le même propriétaire (cette règle est nécessaire pour assurer l'isolation des objets appartenant à des propriétaires différents, cf. II.3.1). Le nouveau mécanisme permet le couplage d'une grappe dans un contexte appartenant au GT.

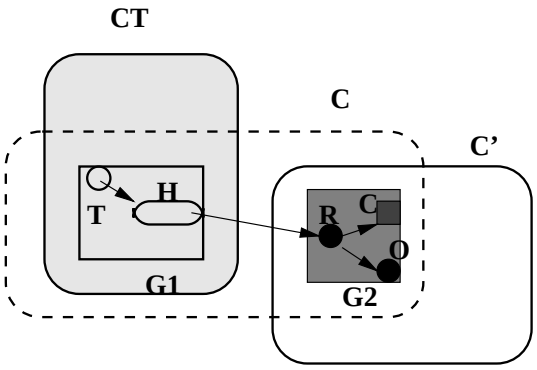

Fig. 5.5 : Utilisation des contextes spéciaux pour la fiablisation des historiques

Éléments de mise en œuvre

Afin de coupler la grappe G d'un objet transaction T dans le contexte CT, la création de T est suivie par l'appel à une méthode nommée *Insertion* sur un objet appartenant au GT. Ceci produit le changement du contexte d'exécution ; l'exécution de la méthode *Insertion* a lieu dans CT, car l'objet cible a comme propriétaire le GT. À son tour, *Insertion* invoque une méthode sur l'objet transaction T. Normalement, l'exécution de cette méthode a lieu dans le contexte où la grappe G a été initialement couplée. Le nouveau mécanisme de couplage force le couplage de G dans le contexte CT ayant comme propriétaire le GT.

V.3.2.5 Mise en œuvre des copies−avant

Nous avons poursuivi deux objectifs pour la mise en œuvre des copies−avant : l'efficacité et la résistance aux pannes d'application. Concrètement, il a fallu assurer que les copies−avant des objets atomiques qui restent couplés en mémoire après une panne(4) soient préservées.

Les copies−avant ne sont pas des objets, mais des zones de mémoire manipulées au niveau des couches basses du système Guide. Pour résister aux pannes d'application, la zone mémoire nécessaire pour une copie−avant est réservée dans la grappe de l'objet atomique (dans la Fig. 5.5, C représente la copie−avant de l'objet O). En conséquence, si la grappe de l'objet reste en mémoire d'exécution après une panne d'application, l'enregistrement de restauration et la copie−avant restent disponibles.

La réservation de la zone nécessaire à une copie, ainsi que la copie elle−même, sont effectués par une fonction système, nommée *doCopy* et appelée par la méthode *Contrôle* dans le cas où un enregistrement de restauration est créé pour l'objet (cf. V.3.1.2). L'adresse de la copie est stockée dans l'enregistrement de restauration.

La restauration d'un objet atomique est mise en œuvre par une fonction système nommée *doRestore*, appelée lors de l'exécution de la méthode *AbandonEffectif*.

V.3.3 Résistance aux pannes de site

Lors d'une panne de site, le contenu de la mémoire d'exécution est perdu. Soit une transaction en cours d'exécution sur un site qui tombe en panne. Étant donné que les effets de la transaction, ainsi que son historique, sont entièrement gardés dans la mémoire d'exécution du site, la transaction devra être abandonnée, sauf si elle se trouvait au cours de la deuxième phase du protocole de validation. Dans ce dernier cas, ses modifications sont déjà journalisées, et la transaction peut être validée sur la base des information contenues dans le journal.

---

(4) Ceci est possible dans les cas des objets partagés, car ceux−ci sont couplés dans des contextes appartenant à des applications différentes.

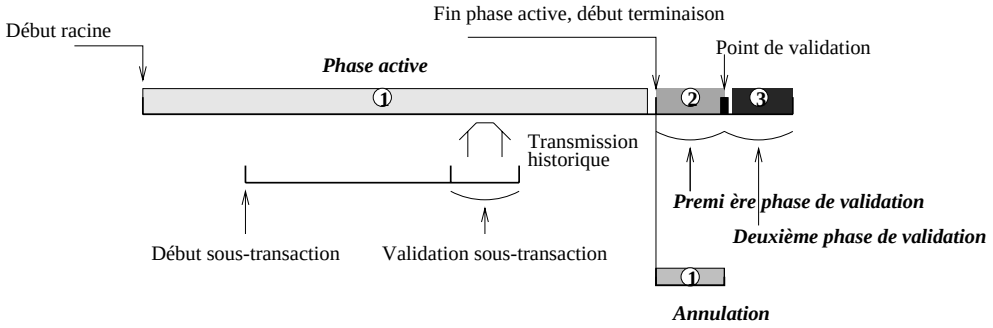

Fig. 5.6 : Les phases d'une transaction emboîtée

L'abandon d'une transaction interrompue en phase active ou en phase d'annulation (cas **1** dans la Fig. 5.6), ou encore au cours de la première phase de validation (cas **2** dans la Fig. 5.6), se fait implicitement, sans l'intervention du STG. Ceci est dû aux causes suivantes :

− le fait que les effets d'une transaction sont localisés : les objets modifiés ainsi que les verrous posés par une transaction se trouvent uniquement sur le site d'exécution de la transaction, dans sa mémoire d'exécution ; or, le contenu de la mémoire d'exécution est perdu.

− la politique de mise à jour différée de la la mémoire stable (cf. V.2.1).

L'interruption d'une transaction qui se trouve dans la deuxième phase de validation (cas **3** dans la Fig. 5.6) met en cause autant l'atomicité de la transaction (car le STG a commencé la mise à jour de la mémoire stable avant la panne), que la permanence (étant donné que la transaction a dépassé le point de validation, les effets de la transaction doivent intégralement se retrouver au niveau de la mémoire stable). Les deux aspects sont simultanément traités par le mécanisme de reprise, non décrit dans le cadre de cette thèse, qui met à jour la mémoire stable sur la base des informations journalisées lors du redémarrage du site.

En conclusion, en dehors du mécanisme mettant en œuvre le protocole de validation à deux phases, dont le rôle est de fournir les données nécessaires à la validation des transactions interrompues pendant la deuxième phase de validation, aucun autre mécanisme n'est nécessaire pour le support de l'atomicité des transactions non−réparties interrompues par une panne de site.

V.4 L'atomicité des opérations appliquées aux objets atomiques

Les événements qui peuvent produire l'interruption de l'exécution d'une méthode sont : une panne d'application, ou une panne de communication lors d'une invocation de méthode à distance. La panne de site n'est pas significative, car les effets de la méthode sont annulés implicitement au niveau de la mémoire d'exécution et il n'y a rien a annuler au niveau de la mémoire stable (cf. V.2.4).

L'atomicité des opérations appliquées aux objets atomiques est implicite (dans le sens où le STG ne gère pas d'informations pour la restauration des objets utilisés hors transactions) ; elle est entièrement basée sur le mécanisme de couplage/découplage des grappes, comme expliqué par la suite.

V.4.1 Utilisation du mécanisme de couplage/découplage des grappes

Le problème à résoudre est illustré au moyen de l'exemple suivant :

Exemple 3. Soit O un objet atomique appartenant à une grappe G. Une application non−transactionnelle invoque la méthode M sur O. Supposons que G est partagée avec une autre application. Considérons que l'objet est marqué (le mécanisme de contrôle de la concurrence autorise l'exécution de M), et que l'exécution de M, commencée, est interrompue par une panne avant sa terminaison.

Tant que la grappe G reste couplée en mémoire d'exécution, si le marquage de O est en mode lecture, alors d'autres applications (transactionnelles ou non) peuvent utiliser O seulement en lecture ; si le marquage est en mode écriture, l'objet O ne peut plus être utilisé (cf. aux règles de marquage et de verrouillage du IV.4), ce qui est souhaitable, car O pourrait être dans un état incohérent (modifié partiellement). L'isolation de l'objet O est garantie par le fait que la marque se trouve dans la même grappe que l'objet.

Lorsque plus aucune application n'utilise G, la grappe est découplée de la mémoire. Le mécanisme de découplage est capable de déterminer si tous les contextes qui ont couplé la grappe se sont terminés normalement, sur la base d'un mécanisme de comptabilisation des demandes de couplage/découpage (le compteur associé à la grappe est nul si tous les contextes dans lesquels la grappe a été couplée se sont terminés normalement ; dans le cas contraire, au moins un des contextes est mort de manière accidentelle, suite à une panne). S'il n'y a pas eu de panne, la grappe est sauvegardée de manière atomique dans la mémoire de stockage. Dans le cas contraire (celui qui nous intéresse) la grappe n'est pas sauvegardée. Par conséquence, les effets de la méthode M (marquage et éventuellement modification) sont automatiquement annulées. L'objet O se trouve en mémoire de stockage dans un

état cohérent. Cet état est retrouvé par toute méthode utilisant l'objet après le re-couplage de G en mémoire d'exécution.

Il faut remarquer ici le fait que les effets d'une transaction validée T ayant modifié la grappe G ne sont pas perdus, grâce au mécanisme de validation à deux phases. Par contre, si une méthode non−transactionnelle R' a modifié un objet atomique O' appartenant à G, alors les effets de R' sont perdus. Ceci respecte la sémantique des objets atomiques utilisés hors transaction : l'atomicité est garantie, mais la permanence des effets non.

V.4.2 Conclusions

La cohérence des objets atomiques utilisés hors transaction est garantie sans aucun mécanisme supplémentaire. En conséquence, le surcoût d'utilisation des objets atomiques hors transaction est dû seulement au contrôle de la concurrence.

Cette stratégie est très efficace en absence de pannes. Dans le cas contraire, elle présente le désavantage que l'objet atomique reste marqué, donc indisponible, tant que la grappe reste couplée en mémoire d'exécution. Il serait donc intéressant d'étudier la mise en œuvre d'un mécanisme supplémentaire, qui permettrait de récupérer l'image cohérente d'un objet atomique à partir de la mémoire de stockage dans le cas où celui−ci resterait indisponible durant un certain temps. Ce mécanisme pourrait être associé avec un mécanisme de délais de garde, à utiliser dans le cas des appels à distance.

V.5 Conclusion

Dans ce chapitre, nous avons présenté les mécanismes qui garantissent l'atomicité des transactions non−réparties multi−contextes, en présence de pannes de transaction, pannes d'application (logicielles) et des pannes de site. L'atomicité est basée sur la mise œuvre des listes d'intention construites dans la mémoire d'exécution pendant la phase active des transactions, ainsi que des journaux−après, construits en mémoire de stockage pendant la phase de validation des transactions racines. L'annulation des transactions suite aux pannes d'application (disparition du contexte d'exécution) a nécessité la mise en œuvre des entités actives appelées Gérants de Transaction sur les sites Guide. Les informations nécessaires à l'annulation sont entièrement gardées en mémoire d'exécution, la principale difficulté étant de garantir la non−perte d'informations lors des pannes de contexte.

Dans ce chapitre nous avons aussi montré que les mécanismes de couplage/découplage fournis par le système Guide, en conjonction avec le mécanisme de

contrôle de concurrence du STG, garantissent l'atomicité des opérations non−transactionnelles appliquées aux objet atomiques.

Les fonctions du GT seront étendues dans le chapitre suivant à la garantie de l'atomicité des transactions réparties.

Chapitre VI

Atomicité des transactions réparties

VI.1 Introduction

Les mécanismes présentés dans le chapitre précédent ne peuvent pas garantir l'atomicité des transactions réparties, car ils ne prennent pas en compte les pannes de communication et ils offrent un support insuffisant pour les pannes de site.

− Les pannes de communication mettent en cause la fiabilité des historiques et des protocoles de terminaison qui les utilisent. Nous avons montré en V.3 que la construction des historiques et le fonctionnement des protocoles de terminaison sont basés sur le mécanisme d'appel de méthode. L'appel de méthode à distance (entre deux objets qui se trouvent sur des sites différents) est implanté en Guide par un mécanisme d'envoi de messages. Or, le système Guide n'offre aucune garantie sur la fiabilité des messages envoyés sur le réseau.

− Lorsqu'une transaction non−répartie s'exécute sur un site qui tombe en panne, ses effets sont entièrement annulés sans que le STG intervienne (cf. V.3.3). Ceci n'est pas vrai dans le cas d'une transaction répartie : suite à une panne de site, les effets de la transaction sont annulés uniquement sur le site défaillant, et c'est le STG qui doit annuler les effets sur les autres sites sur lesquels la transaction s'est étendue. D'autre part, une panne de site peut conduire à la perte de l'historique dans le cas où celui−ci se trouvait dans la mémoire du site défaillant.

Les problèmes qui découlent de la répartition peuvent être groupés en deux catégories :

- La perte d'informations qui constituent l'historique d'une transaction répartie, due aux pannes de communication ou aux pannes de site.
- L'adaptation des protocoles de terminaison définis dans le chapitre précédant pour la prise en compte de ces pannes.

Pour fiabiliser l'historique des transactions réparties, nous avons introduit le concept de *représentant de transaction*. L'objectif poursuivi est de décentraliser la gestion des effets des transactions réparties. Ceci permet de rendre le STG résistant aux pannes de matériel, tout en conservant le principe d'annulation basé sur l'utilisation des listes d'intentions introduit en V.2.2.

Pour garantir l'atomicité des transactions réparties, le STG doit connaître leurs sites d'exécution. La gestion des informations sur la répartition des transactions ne peut être effectuée par les objets spéciaux, car ceci nécessiterait l'utilisation des appels de méthode à distance. Ce sont les *gérants de transaction* (GT) qui effectuent la gestion des informations de répartition (VI.3) et qui coordonnent la terminaison des transaction réparties (VI.4).

VI.2 Utilisation des représentants

Nous appelons *site visité* un site dont la mémoire contient un objet atomique utilisé par la transaction. L'ensemble des sites visités et le site de création de la transaction constituent l'ensemble des *sites de travail*.

L'utilisation des représentants permet de répartir la liste d'intentions (LI) d'une transaction sur l'ensemble de ses sites de travail. Ceci permet de résoudre le problème de la perte de l'historique en cas de panne du site de création. Il reste que si un site de travail tombe en panne, l'historique local d'une transaction T qui a visité le site est perdu. Cependant, cet historique n'est plus nécessaire, car les effets de T locaux au site défaillant disparaissent de la mémoire d'exécution. Le protocole de terminaison se charge ensuite d'annuler les effets de T sur les sites restés opérationnels.

D'autre part, la répartition de la liste d'intentions d'une transaction sur les sites de travail permet d'enregistrer ses effets dans la LI locale au site où ces effets se produisent. Ceci permet d'éviter les pertes d'information qui pourraient être causées par des pannes de communication.

L'historique d'une transaction répartie est constitué par deux sortes d'informations :

- Celles qui servent à retrouver les effets de la transaction sur chacun des sites de travail (ces informations sont gérées au moyen des LI locales).
- Des informations concernant la répartition de la transaction et permettant de connaître l'ensemble des sites de travail (ces informations sont gérées par les GT).

La gestion de l'historique des transactions réparties est fiable si les LI locales sont complètes et si la gestion des informations de répartition est fiable. Cette section présente les mécanismes qui permettent la gestion des LI complètes, dans un contexte à transactions emboîtées. La gestion fiable de la répartition est présentée dans VI.3.

VI.2.1 Mise en œuvre des représentants

Les représentants sont implantés par des objets transaction (instances de la classe *Transaction*). Cette solution présente plusieurs avantages.

− La LI locale d'une transaction peut être mise en œuvre par l'objet historique associé à son représentant local.

− Chaque représentant peut effectuer la terminaison de la transaction localement à son site. Les représentants peuvent valider ou annuler les effets de la transaction sur la base des informations enregistrés dans leur LI, conformément aux algorithmes déjà définis en V.3.1.2 et V.3.1.3.

− Cette approche facilite l'extension du fonctionnement des GT pour la gestion de la répartition. En partant du fait que les GT sont conçus pour gérer des informations concernant les transactions, il résulte que les représentants doivent être enregistrés dans la structure de données gérée par le GT de leur site, et que des informations supplémentaires doivent être associés aux représentants pour la gestion de la répartition (voir VI.3.3).

VI.2.2 Conséquences sur la structure du STG

Une transaction répartie est donc mise en œuvre par un ensemble d'objets transaction : l'objet créé par l'application et l'ensemble des représentants, qui sont cachés à l'application. Au niveau de l'application, une transaction est identifiée par la référence de l'objet transaction qu'elle a créé.

Afin de simplifier la présentation, nous utilisons dans la suite le terme de *représentant* pour tout objet transaction mettant en œuvre une transaction donnée.

Pour différencier les représentants d'une transaction, les GT leur associe un rôle pouvant être celui de :

- *coordinateur*, pour les représentants mis en œuvre par des objets transaction créés par une application,
- *subordonné*, pour les représentants crées par le STG.

Nous disons qu'une transaction est *représentée* sur un site, lorsqu'elle a un représentant sur ce site.

Sur chacun des sites de travail d'une transaction il y a une activité transactionnelle qui exécute les appels de méthode pour le compte de celle−ci (à un instant donné, une seule de ces activités peut être active). Nous avons montré dans IV.5.2 que les activités transactionnelles sont caractérisées par l'identificateur de la *transaction courante* (i.e. par la référence d'un objet transaction). Dans le cas des transactions réparties, les activités transactionnelles sont caractérisées par la *référence du représentant coordinateur*.

Par conséquent, la répartition des transactions n'induit aucun changement dans la mise en œuvre du concept de transaction courante. Le contrôle de la concurrence est fait de la même manière que dans le cas des transactions non−réparties : lorsqu'une transaction T utilise un objet atomique qui se trouve sur un site visité, le propriétaire du verrou posé sur l'objet est toujours T, et pas le représentant local.

La figure Fig. 6.1 met en évidence les composantes qui concourent à l'atomicité d'une transaction répartie. On distingue :

* Les représentants (T, R1, R3 et M), qui sont des objets transaction, donc des entités passives en Guide,

* Les GT des sites de travail de la transaction (GTi, i∈[1,3]), qui sont des entités actives. Les représentants sont enregistrés auprès de ces GT.

* Le mécanisme de contrôle de la concurrence, qui participe au maintien de l'atomicité de la transaction en effectuant la mise à jour de la LI locale avec des enregistrements correspondant aux objets atomiques que la transaction utilise.

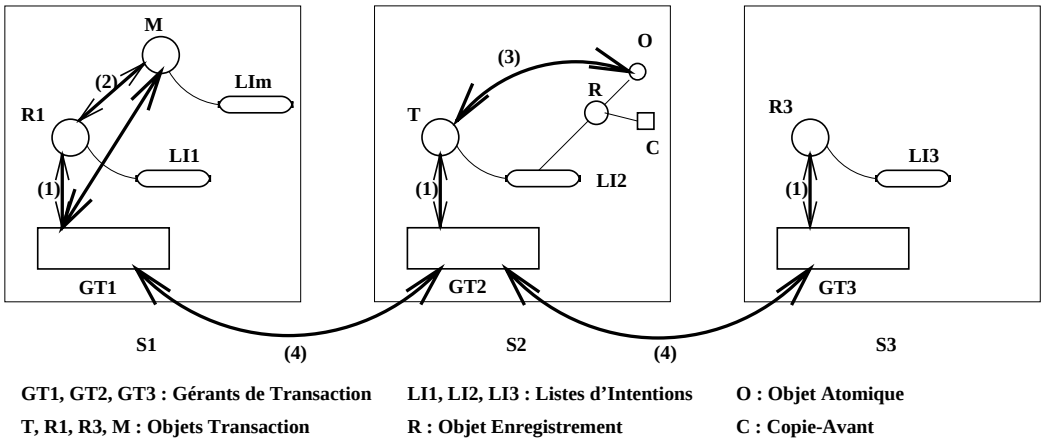

Fig. 6.1 : Les composantes qui assurent l'atomicité d'une transaction répartie

Dans la Fig. 6.1 la transaction T, créée sur le site S2 s'étend sur les sites S1 et S3, R1 et R3 étant ses représentants respectifs. La mère de T est représentée par M sur le site S1. Les traits en gras relient les composantes dont les interactions assurent l'atomicité de T.

On distingue deux catégories d'interactions entre composantes : locales et à distance. Les *interactions à distance* sont limitées à l'interaction des GT (et marquées par (4) dans la Fig. 6.1). Les *interactions locales* sont de plusieurs types :

(1) entre un représentant et le GT local, au cours du déroulement du protocole de terminaison,

(2) entre deux représentants en relation d'affiliation directe (l'un est descendant de l'autre), lors de la validation du descendant.

(3) entre un objet atomique et le représentant de la transaction qui l'utilise. À cette occasion, l'objet est enregistré dans la LI locale (dans la figure Fig. 6.1, O est un objet atomique utilisé par T sur le site S2).

VI.2.3 Différentes stratégies pour la création des représentants d'une transaction emboîtée

Dans le cas des transactions emboîtées, les composantes d'un arbre transactionnel peuvent avoir des sites de création différents. Les objets transaction créés par l'application constituent l'*arbre conceptuel.* Les représentants de transaction forment sur les différents sites de travail des sous−arbres qui constituent les

projections locales de l'arbre conceptuel. Le mode de constitution de ces projections dépend de la stratégie de création des représentants.

Un représentant de transaction est créé lorsque celle−ci utilise pour la première fois un objet atomique sur un site visité. Il s'agit dans ce cas d'une extension *directe* de la transaction. Cependant, un représentant peut aussi être créé comme conséquence de l'extension d'une autre composante de son arbre transactionnel. Il s'agit dans ce cas d'une extension *indirecte*.

Un exemple d'extension indirecte est l'extension de la mère suite à l'extension de la fille. Ceci est nécessaire pour éviter les pertes d'historiques lors de la validation de la fille.

Selon l'événement qui déclenche l'extension indirecte d'une transaction, nous pouvons concevoir plusieurs approches d'extension indirecte :

− *Extension indirecte immédiate (EI)*. L'extension d'une sous−transaction sur un site S implique l'extension immédiate de sa mère sur S (sauf dans le cas où celle−ci s'était déjà étendu sur S de manière directe). Par récursivité, l'extension d'une sous−transaction conduit à la reconstitution de sa hiérarchie sur le nouveau site visité. Par conséquent, la racine de l'arbre transactionnel est représentée sur tous les sites où ses inférieurs sont représentés.

− *Extension indirecte différée (ED).* L'extension d'une transaction mère sur un site visité par une de ses filles est différée jusqu'à la validation de sa fille. Une transaction est alors représentée sur les sites directement visités et sur les sites visités par ses filles validées.

− L'*extension indirecte mixte (EM)* est une combinaison des deux premières. L'extension d'une transaction sur un site visité peut engendrer la création immédiate de représentants pour d'autres composantes de son arbre transactionnel conceptuel, afin de respecter la règle qui suit :

Règle d'extension mixte. Dans le cas où deux transactions Tx et Ty qui se trouvent en relation d'affiliation (l'une est supérieur de l'autre) sont représentées sur un site S, alors les transactions appartenant au chemin qui unit Tx et Ty doivent être représentées sur S.

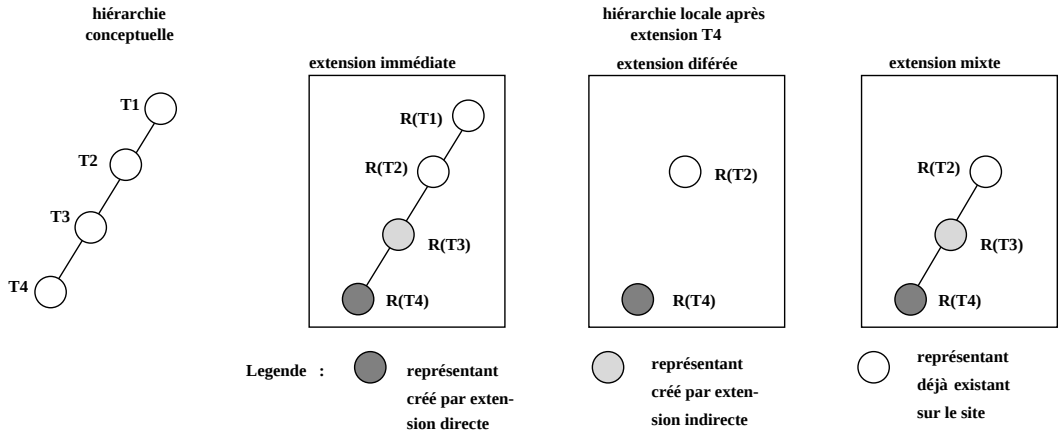

Fig. 6.2 : Illustration des différentes approches d'extension indirecte

La Fig. 6.2 illustre la projection d'une hiérarchie de transactions sur un site S selon les différentes approches d'extension indirecte. Les projections locales sont représentées après l'extension de la transaction T4, en supposant que lors de cette extension T2 est déjà représentée sur le site.

− Dans le cas de EI, lors de l'extension de T4 il existent déjà sur le site S deux objets transaction : R(T2), créé par extension directe, et R(T1) qui apparaît suite à l'extension de T2. Par conséquent, la création de R(T4) nécessite l'extension de T3, i.e. création de R(T3), pour que la hiérarchie de T4 soit entièrement reconstruite sur le site.

− Dans le cas de la deuxième approche (ED ci−dessus), après l'extension de T2 et T4 on retrouve sur S leurs représentants respectifs.

− Enfin, selon l'approche EM, l'extension de T2 sur S conduit seulement à la création de R(T2), alors que l'extension de T4 implique la création de R(T4) et de R(T3). Ceci résulte du fait que T4 et T2 sont en relation d'affiliation et que T3 se trouve sur le chemin qui unit T2 avec T4.

Éléments de mise en œuvre.

La relation d'affiliation entre les différents représentants est maintenue au moyen de la structure de données *TransList* gérée par le GT, qui a été définie dans V.3.2.2. Pour tout objet transaction X enregistré auprès d'un GT, *X.Sup* donne la référence du supérieur local, et *X.InfList* sert à gérer les filles locales. Par exemple, R(T3).Sup = R(T2) et R(T3).InfList = {R(T4)}.

Pour déterminer si deux représentants sont en relation d'affiliation, il est nécessaire d'associer une information supplémentaire aux objets transaction enregistrés dans la *TransList* : la référence de la racine de l'arbre conceptuel. Par

exemple, dans le cas où T1 est la racine, alors R(T4).Racine = R(T2).Racine = T1. Ayant cette information, on peut parcourir les supérieurs d'une des transactions (par exemple T4), ce qui permet de constater que T2 est un de ses supérieurs.

VI.2.4 Analyse comparative des différentes stratégies

L'approche d'extension indirecte différée (ED) semble la plus efficace, car elle permet d'éviter l'extension d'une transaction sur des sites visités par des sous−transactions qui seront finalement abandonnée.

Cependant, l'approche ED ne peut pas être implantée en Guide, car elle ne permet pas de résister aux pannes d'application multi−contexte. Nous avons montré en V.3.2.2 que, pour annuler les effets d'une transaction affectée par une panne d'application, le GT annule les effets de chacune de ses composantes actives, en les traitant dans le sens "feuilles −> racine". L'approche ED ne permet pas d'établir la relation d'affiliation entre des objets transaction appartenant à une même hiérarchie et se trouvant sur un même site. Considérons la hiérarchie transactionnelle présentée dans la Fig. 6.2 et supposons qu'il est nécessaire d'abandonner la transaction globale suite à une panne. Dans le cas de l'approche ED, le GT ne détient pas assez d'informations pour déterminer que R(T2) est un supérieur de R(T4). Cet inconvénient est écarté par l'approche d'extension mixte, car la relation d'affiliation est maintenue par le GT. Le GT peut alors abandonner les effets locaux de la transaction globale en annulant les effets de R(T4), de R(T3) et de R(T2).

Nous avons finalement adopté l'approche d'extension immédiate (EI) parce qu'elle permet de simplifier les protocoles de terminaison. Considérons par exemple la validation des sous−transactions. Dans le cas de l'approche EI la validation d'une sous−transaction sur un site de travail est effectuée toujours en local, alors que dans le cas de l'approche EM cette opération peut nécessiter l'extension de la mère sur le site. Si l'extension de la mère échoue à cause d'une panne de communication, la validation de la sous−transaction n'est pas possible. Par conséquent, la validation des sous−transactions nécessite une mise en œuvre en deux phases. En adoptant l'approche EI, nous avons pu restreindre le protocole de validation des sous−transactions à une seule phase ; ce protocole est rendu résistant aux pannes grâce à un mécanisme de *terminaison retardée* présenté en VI.4.2.

VI.3 Gestion de la répartition

Pour des raisons de simplicité, nous avons choisi une approche centralisée pour la gestion de la répartition. C'est le GT du site de création d'une transaction, appelé *coordinateur*, qui doit connaître les GT des sites visités par la transaction, appelés *participants*. D'autre part, il est nécessaire que les participants connaissent

l'identité du coordinateur. Nous pouvons noter qu'un GT peut être le coordinateur d'une ou plusieurs transactions en même temps que participant à d'autres transactions.

Les GT gèrent la répartition au moyen d'informations associées aux représentants. Selon le rôle d'un représentant, ces informations peuvent être :

- Pour un représentant coordinateur : la liste des GT participants, *PartList*.
- Pour un représentant *subordonné* : un couple formé par l'identité du GT coordinateur et la référence du représentant coordinateur, couple désigné par *[GT−coordinateur, T−coordinateur]*.

Considérons la transaction répartie T, représentée dans la Fig. 6.1. Le gérant GT2 attribue à l'objet transaction T le rôle de coordinateur et lui associe la liste de participants {GT1, GT3}. Le gérant GT1 attribue à l'objet R1 le rôle de subordonné et lui associe le couple [GT2, T].

Cette section décrit les mécanismes qui, pour une transaction donnée, permettent à son GT coordinateur de constituer la liste des participants, et aux participants de connaître le coordinateur de la transaction à laquelle ils participent. Ces informations, qui définissent la répartition d'une transaction, sont en effet critiques. Pour que l'atomicité d'une transaction soit garantie, la liste des participants associée au représentant coordinateur doit être complète. Les mécanismes qui assurent une gestion fiable des informations de répartition en faisant appel à l'approche d'extension indirecte immédiate (EI) sont présentés dans la suite en prenant en considération les aspects suivants :

− L'identification de l'événement qui conduit à la création du représentant d'une transaction T sur un site visité S.

− La mise à jour des informations de répartition au niveau du GT participant.

− La mise à jour des informations de répartition au niveau du GT coordinateur.

VI.3.1 Mécanisme d'extension directe

L'extension directe de la transaction T sur le site S se produit lors de la première utilisation, pour le compte de T, d'un objet atomique O se trouvant sur S. Le représentant R(T) est créé suite à l'interaction du GT avec le mécanisme de contrôle de concurrence. Cet interaction se produit lorsque la méthode *Contrôle* demande au GT l'identificateur du représentant local de la transaction, afin de pouvoir mettre à jour l'historique local de la transaction. Comme un tel représentant n'existe pas encore, le GT crée un objet transaction et l'enregistre dans la *TransList* en tant que subordonné de T. Les autres informations associés au représentant, tel que le couple

[GT−coordinateur, T−coordinateur], sont disponibles sur le site (voir les éléments de mise en œuvre). Ainsi le GT devient participant de T.

La mise à jour des informations de répartition au niveau du GT coordinateur implique un échange de messages entre le participant et le coordinateur. Le participant (soit GTp) envoie un *message d'extension* au coordinateur (soit GTc) pour lui signaler l'extension de T sur son site. Si le message d'extension est reçu par le GTc, il ajoute le GTp à la *PartList* de T et renvoie un *message d'acquittement d'extension.* Le GTp répète le message d'extension jusqu'à la réception d'un acquittement ou jusqu'à l'expiration d'un délai de garde. Si le GTc n'a pas pu être contacté, l'extension de T sur S n'est pas autorisée. Dans ce cas, le mécanisme de contrôle de concurrence refuse l'utilisation de l'objet atomique pour le compte de T.

Éléments de mise en œuvre

Nous avons montré en VI.2.2 que les activités transactionnelles sont caractérisées par la transaction courante, i.e. la référence du représentant coordinateur, qui est enregistrée dans le descripteur d'activité. Lors de l'extension directe d'une transaction T, le GT du site visité peut obtenir cette référence et initialiser *T−coordinateur*, en accédant au descripteur de l'activité qui exécute la transaction. La mise à jour de la *TransList* nécessite aussi l'initialisation de *GT−coordinateur.* Par conséquent, nous avons ajouté au descripteur d'activité un champ supplémentaire afin d'y enregistrer l'identité du GT coordinateur[(1)]. Dans le cas de l'approche EM, un troisième champ est nécessaire pour garder la référence de la transaction racine.

Le changement de la transaction courante a comme effet de bord la mise à jour du descripteur d'activité. Lors du lancement d'une nouvelle transaction, la référence du représentant coordinateur et l'identité du GT coordinateur (l'identité du GT local au site de création) remplacent les anciennes valeurs enregistrées dans le descripteur d'activité. Les anciennes valeurs doivent être sauvegardées, afin de pouvoir les utiliser pour remettre à jour le descripteur d'activité lors de la terminaison de cette transaction. Ces informations sont sauvegardées par GT local dans l'entrée qui a été créée dans la *TransList* pour la nouvelle transaction. Il s'ensuit que les GT associent aux représentants ayant le rôle de coordinateur la référence de la transaction mère et l'identité du GT coordinateur de la mère, désignés par le couple *[GTm−coordinateur, Tm−coordinateur]*.

---

(1) Il résulte que cette information est vehiculée par les messages effectuant les appels de méthode à distance.

VI.3.2 Mécanisme d'extension indirecte

Soit T' une composante de l'arbre transactionnel dont T fait partie. L'extension de T' sur le site S peut être la conséquence de l'extension de T sur S (dans le cas des approches d'extension EI et EM), ou bien la conséquence de la validation de T (seulement dans le cas de l'approche d'extension EM).

VI.3.2.1 Extension indirecte de T' suite à l'extension de T

L'événement qui conduit à l'extension de T' est la création d'un représentant de T sur S. Afin de prendre en compte l'extension de T' au niveau de la gestion de la répartition, il faut d'abord déterminer qui est T'.

− Dans le cas de l'approche EI, T' est la mère de T. Son extension est nécessaire seulement si elle n'est pas encore représentée sur S. L'extension de T' peut engendrer l'extension de la mère de T'. Le processus s'arrête lors de l'extension de la racine, ou lorsqu'un supérieur est déjà représenté sur S.

− Dans le cas de l'approche EM, l'extension de T' est nécessaire seulement s'il existe une transaction T^* tel que T et T^* sont affiliées, et T^* est représentée sur S. Soit Tx = Inférieur(T, T^*) et Ty = Supérieur(T, T^*). Alors T' est le supérieur de Tx. L'extension de T' est suivie par l'extension des supérieurs de T' qui sont inférieurs de Ty. Pour simplifier la présentation nous supposons que T = Tx, donc T' est mère de T.

Pour mettre à jour les informations de répartition sur S, ainsi qu'au niveau du site de création de T', il est nécessaire de connaître l'identité du GT coordinateur de T'. Étant donné que T' est mère de T, cette information est enregistrée par le GT coordinateur de T (cf. VI.3.2). Celui−ci peut donc envoyer ces informations dans le message qui acquitte l'extension de T sur S.

Pour illustrer l'interaction des GT considérons une hiérarchie de transactions composée de T, T' et T'', représentées sur un site S (voir la Fig. 6.3). Le site de création de T est S2, celui de T' est S1, et celui de T'' est S0 (non représenté). Le couple [GT1, T'] est associé à T dans la *TransList* gérée par GT2. Le couple [GT0, T''] est associé à T' dans la *TransList* gérée par GT1.

Lors de l'extension de T sur le site S, GTs envoie à GT2 le message *extension(T, GTs)*. Sur réception du message, GT2 répond par un message d'acquittement *acquit−ext(T, T', GT1)*. Les informations ainsi envoyées au GTs rendent possibles l'extension de T' (donc la notification de GT1), qui sera suivie de l'extension de T'', conformément au même schéma.

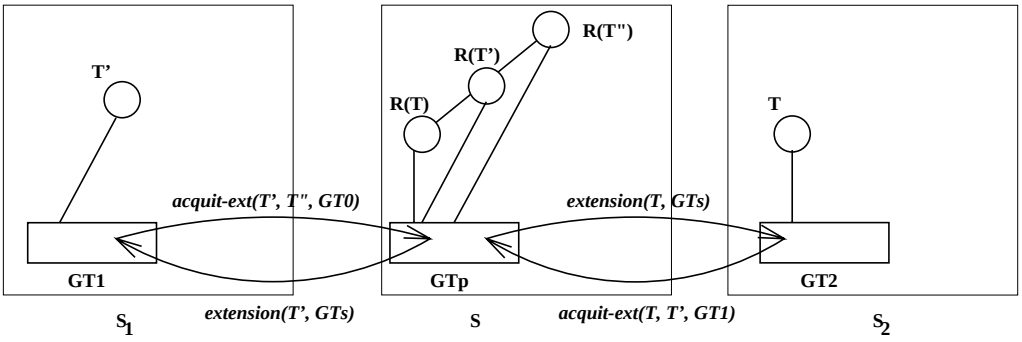

Fig. 6.3 : Interaction des composantes dans le cas de l'extension indirecte

VI.3.2.2 Extension indirecte de T' suite à la validation de T

L'événement qui conduit à l'extension de T' est la réception sur le site S d'un message demandant la validation de T. L'extension a lieu si T' n'est pas encore représentée sur S, T' étant dans ce cas la mère de T.

Nous avons vu que l'extension de T' sur le site S nécessite la connaissance de son coordinateur sur S. L'identité de son coordinateur est envoyée sur S par le coordinateur de T, dans le message qui demande la validation de T.

VI.3.3 Le point sur l'architecture du STG

La Fig. 6.4 présente une synthèse de l'interaction entre les GT et les autres composantes nécessaires à la gestion fiable de l'historique d'une transaction répartie T.

L'objet T est enregistré auprès du GTc lors du lancement de la transaction T. L'extension directe de T sur un site visité est dues à l'interaction avec le mécanisme de contrôle de concurrence, qui doit enregistrer les effets du T dans la LI associée à R(T).

L'extension de T (directe ou indirecte) implique la mise à jour des informations de répartition au niveau du GTc. Ceci est assuré par l'envoi des messages d'extension. Les informations contenues dans les messages d'acquittement d'extension rendent possible l'extension indirecte des supérieurs de T. De même, dans le cas de l'approche EM, les informations contenues dans les messages de demande de validation rendent possible l'extension indirecte de la mère de T sur un site visité par T.

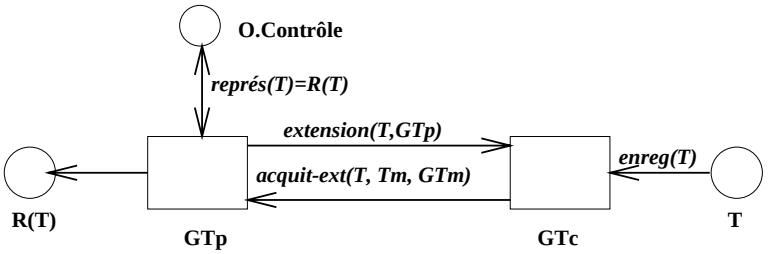

Fig. 6.4 : Interaction des composantes dans les différents cas d'extension

La structure de données *TransList* obtenue après l'intégration des besoins liés à la gestion de la répartition :

- *Sup*, le supérieur local
- *InfList*, la liste des inférieurs locaux
- *Rôle*, peut être *coordinateur* ou *subordonné.*

  Si *Rôle = coordinateur*

  – *PartList*, la liste des GT participants

  – [*GTm−coordinateur*, *Tm−coordinateur*], le GT coordinateur de la mère de T et la référence de la mère de T.

  Si *Rôle = subordonné*

  – [GT−coordinateur, T−coordinateur], le GT coordinateur et la référence de T.
- *Racine*, la référence de l'objet transaction qui est la racine de l'arbre conceptuel. Utilisée seulement dans le cas de l'approche EM.

VI.4 Protocoles de terminaison

Les protocoles de terminaison spécifiés dans ce chapitre prennent en compte la nouvelle approche de gestion de l'historique, basée sur l'utilisation des représentants. La stratégie d'extension indirecte considérée est celle de l'extension immédiate (EI).

Les pannes de matériel peuvent avoir une incidence sur l'exécution de la transaction, mais aussi sur le déroulement du protocole de terminaison. Par exemple, la panne d'un site S peut provoquer l'interruption d'une transaction si elle se produit pendant l'exécution de la transaction sur ce site. D'autre part, la panne peut se produire après que la transaction ait fini le traitement sur le site S. Dans ce cas, si la transaction arrive au point de terminaison, alors la panne doit être prise en compte par le protocole de terminaison. De manière similaire, une panne de communication

peut mener au blocage d'une transaction, ou peut empêcher le déroulement du protocole de terminaison.

Nous commençons par présenter le principe de fonctionnement des protocoles de terminaison, en faisant abstraction des éventuelles pannes qui pourraient affecter leur fonctionnement (VI.4.1). Dans un deuxième temps, nous allons rajouter les fonctions nécessaires pour rendre ces protocoles résistants aux pannes de communication et de site (VI.4.2).

VI.4.1 Principe de fonctionnement

Le rôle d'un protocole de terminaison est de garantir l'exécution d'un même algorithme de terminaison sur tous les sites de travail de la transaction. L'algorithme de terminaison à exécuter peur être soit décidé par l'application, dans le cas de terminaison normale, soit dépendant de l'état de la transaction, dans le cas d'une terminaison accidentelle. Dans ce deuxième cas, c'est le STG qui doit détecter la panne, choisir le mode de terminaison et déclencher le protocole correspondant.

La mise en œuvre des protocole de terminaison a été influencée par l'approche de mise en œuvre de l'historique : approche centralisé pour la gestion de la répartition et l'utilisation des représentants pour la gestion locale des LI. Le traitement correspondant à la terminaison effective d'une transaction sur les différents sites de travail est effectué par les représentants de transaction, conformément aux algorithmes de terminaison définis en V.3.1. La coordination des représentants se fait par l'interaction des GT.

La terminaison normale est initiée sur le site de création de la transaction par l'appel à la méthode *Validation* ou *Abandon* sur l'objet coordinateur. En dehors du traitement de validation ou d'abandon effectué par le représentant coordinateur, le coordinateur (GTc) effectue aussi le changement de la transaction courante. En outre, il demande aux participants (les GTp) d'effectuer la terminaison de la transaction sur leurs sites (il diffuse un message de *demande de validation* ou de *demande d'abandon*). Sur la réception d'un tel message, tout GTp déclenche le traitement de terminaison effectif (par l'appel de la méthode *Validation* on *Abandon* sur l'objet transaction mettant en œuvre le représentant).

Dans l'hypothèse que la terminaison n'est pas interrompue par une panne, l'abandon d'une transaction ou la validation d'une transaction emboîtée peuvent se dérouler dans une seule phase. Cependant, la validation d'une transaction racine nécessite deux phases, car la construction du journal−après se fait en coopération avec un mécanisme extérieur au STG, le mécanisme de journalisation étant implanté au niveau du service de stockage de Guide. Par conséquent, lors de la validation d'une transaction racine, le coordinateur envoie un message de *demande de*

préparation aux participants et attend un *message d'acquittement* de leur part, afin de pouvoir prendre la décision de déclencher une deuxième phase de validation ou d'annulation.

Ce protocole de base est étendu dans la suite pour la prise en compte des terminaisons accidentelles.

VI.4.1.1 Extension pour la prise en compte des pannes d'application

Considérons le cas où l'exécution de la transaction T est interrompue par une panne d'application qui se produit sur un site S. Conformément à la spécification donnée dans V.3.2.2, ce type de panne est détecté par le GT local, qui prend la décision d'abandonner la transaction globale en abandonnant la racine de T, soit T*. Ceci garantit l'annulation des effets de toutes les composantes actives(2) de la transaction. Le fonctionnement des GT doit être étendu afin que l'abandon de T* ait lieu sur tous les sites de travail. Cette extension est basée le fait que T* est représentée sur tous les sites contenant au moins un représentant d'une composante active de la transaction (il existe une projection de l'arbre conceptuel sur tous les sites de travail visités par T* et ses inférieurs). Il suffit alors de demander à son coordinateur l'annulation de T*.

Le fonctionnement des GT à la détection de la panne d'une transaction T peut être décrit comme suit :

Soit GTi le gérant du site S.

− Si T* a été créée sur ce site, alors GTi est le coordinateur de T*. Il diffuse alors un message de *demande d'annulation* aux participants de T*.

− Sinon, T* est représentée sur S ; il existe alors une entrée correspondant à T* dans la *TransList*. Cette entrée contient l'identité du coordinateur de T* (GTc). Le GTi envoie un message de *demande d'annulation* au GTc qui diffuse le message aux participants.

− Sur réception d'un message de demande d'annulation d'une transaction, tout GT déclenche l'algorithme d'annulation défini en V.3.2.2.

Dans le cas de l'approche EM, l'algorithme est beaucoup plus compliqué, car T* peut ne pas être représentée sur le site où se produit la panne, donc le GTi ne connaît pas le GTc de T*. D'autre part, il peut y avoir d'autres sites, contenant des représentants des composantes actives de la transaction sur des sites où T* n'est pas représentée ; ces sites ne sont alors pas connus par le GTc de T*.

---

(2) Les composantes actives sont des sous−transactions qui ne sont pas encore validées ou abandonnées.

VI.4.1.2 Extension pour la prise en compte des pannes de matériel

L'exécution d'une transaction peut être interrompue par la panne du site courant d'exécution, ou suite à un appel de méthode à distance, lorsque le site distant est tombé en panne. Par ailleurs, en effectuant un appel de méthode à distance, une transaction peut rester bloquée à cause d'une panne de communication. Étant donné que l'application ne peut plus continuer dans ce cas, il est nécessaire d'abandonner la racine de T. Par conséquent, l'annulation des effets d'une transaction interrompue par une panne matérielle ce fait de la même manière que lors d'une interruption provoquée par une panne d'application. Il ne reste qu'à étendre le fonctionnement des GT pour la détection des transactions interrompues par les pannes matérielles.

Nous avons spécifié un mécanisme pour la détection des transactions défaillantes qui est basé sur l'interaction avec le mécanisme de contrôle de concurrence. Lors de la détection d'un conflit d'accès, le mécanisme de contrôle de concurrence envoie un message au GT local (soit GTi) en lui indiquant la référence de la transaction qui provoque le conflit. Lors de la réception du message, le GTi essaie de déterminer l'état de la transaction en interrogeant les participants de la transaction :

- Si le GTi est le coordinateur de T, alors il diffuse un message d'interrogation d'état aux participants.
- Si le GTi est un participant, il envoie le message d'interrogation d'état au GTc. Si celui−ci est en vie, il diffuse le message d'interrogation aux participants.
- Un message d'interrogation est répété jusqu'à l'obtention d'un acquittement contenant l'état de la transaction (*active* ou *connaît−pas*), ou jusqu'à l'expiration d'un délai de garde. Si le délai de garde est dépassé, ou si un site répond par *connaît−pas*, le GT déclenche l'abandon de la transaction globale.

Les messages d'*interrogation d'état* permettent au coordinateur de détecter si un site visité est tombé en panne, mais permettent aussi la détection de la panne du site coordinateur par les participants. En effet, le GT d'un site défaillant ne répond pas, ou ne connaît pas la transaction (dans le cas ou le site est revenu après une panne). Cependant, ce mécanisme ne permet pas toujours la détection des transactions bloquées à cause d'une panne de communication, car la panne qui a provoqué l'interruption de l'application a pu disparaître avant que le processus d'interrogation soit déclenché. La prise en compte de ce type de panne nécessite soit des modifications au niveau du mécanisme mettant en œuvre les appels de méthode à distance, soit l'utilisation de délais maximaux pour l'exécution des transactions.

VI.4.1.3 Nouveau point sur l'architecture du STG

La Fig. 6.5 présente la synthèse de l'interaction entre les GT d'une transaction et l'interaction des GT avec les objet transaction au cours du déroulement des différents protocoles de terminaison.

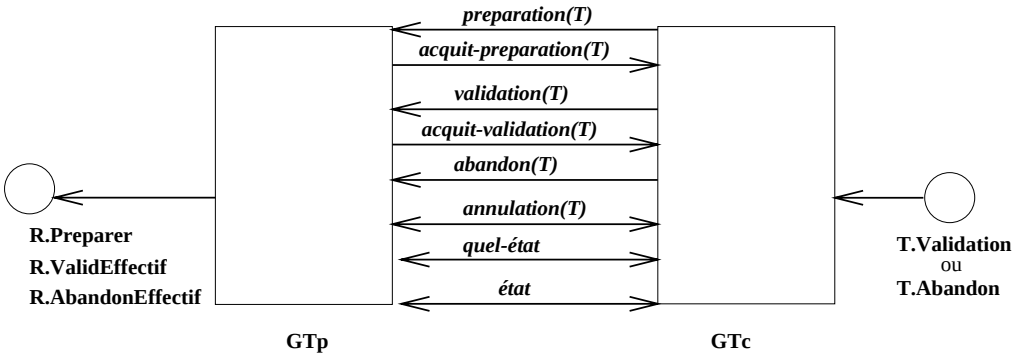

Fig. 6.5 : Interaction des composantes pendant le déroulement des protocoles de terminaison

VI.4.2 Fiablilisation de la terminaison

Pour rendre le protocole décrit ci−dessus résistant aux pannes, il est nécessaire d'affiner la gestion de l'*état* des transactions. Tout objet transaction présent dans la *TransList* d'un GT a un état associé. L'état du représentant coordinateur est initialisé par *active* lors du lancement de la transaction. L'état d'un représentant est initialisé par la même valeur lors de sa création.

Jusqu'à maintenant, nous avons considéré que les GT enlevaient les transactions terminées de la *TransList* ; dans le cas des sous−transactions, elles étaient enlevées de la liste des filles actives associées à leur supérieur local. On pouvait dire qu'une transaction terminée est "oubliée" après sa terminaison. Lorsqu'un GT est interrogé sur l'état d'une transaction T, il répond par *active*, si T est présente dans la *TransList*, ou par *connaît−pas* dans le cas contraire.

Dans la suite, nous présentons les états des représentants d'une transaction, ainsi que les transitions d'état au cours du déroulement du protocole de terminaison.

VI.4.2.1 Pannes de communication

Terminaison d'une sous−transaction

La résistance aux pannes de communication des protocoles de terminaison des sous−transactions est basée sur la terminaison des supérieurs − ou *terminaison retardée* − et sur l'utilisation des messages d'interrogation.

Soit S un site visité par la sous−transaction T, GTp le gérant de ce site et GTc le coordinateur de T. Supposons que le GTp ne reçoit pas le message de demande de validation envoyé par le GTc. T reste alors dans la liste des filles actives de son supérieur local. Elle sera détectée lors de la terminaison de sa mère. Si la mère est abandonnée, T est abandonnée. Si la mère est validée, le GTp va d'abord déterminer le mode de terminaison de sa fille (T) en interrogeant le GTc sur l'état de T. La réponse du GTc doit permettre la validation de T. Il résulte que le GTc ne peut pas oublier la transaction T après sa validation locale.

La terminaison d'une sous−transaction est retardée au plus tard jusqu'à la terminaison de sa racine.

Par conséquent, afin de résister aux pannes de communication nous avons modifié le comportement des GT de la manière suivante :

− Un GT qui reçoit la demande d'abandon d'une transaction T doit effectuer l'abandon des filles actives (si elles existent) en plus de l'abandon local de T.

− Un GT qui reçoit la demande de validation d'une transaction T doit effectuer la terminaison de ses filles actives (si elles existent) avant de pouvoir effectuer la validation locale de T.

− Après la validation locale d'un représentant, un GTp peut oublier la transaction après avoir envoyé un message d'acquittement de validation.

− Cependant, un GTc ne peut pas oublier la transaction T avant d'avoir reçu le message d'acquittement de la part de tous les participants. En attendant, l'état de représentant coordinateur passe dans *en−validation* (voir Fig. 6.8).

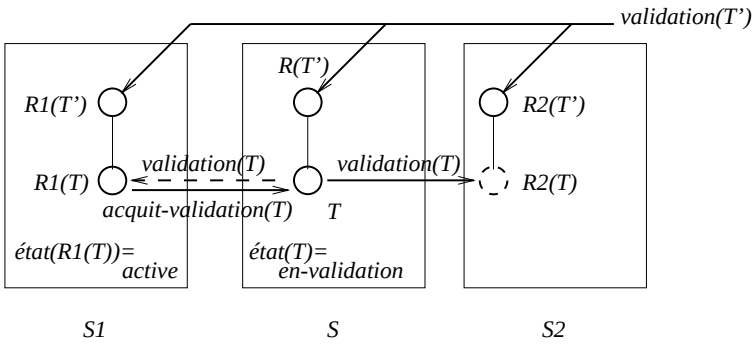

Fig. 6.6 : Validation tardive d'une sous−transaction

La Fig. 6.6 présente l'état de T sur son site de création S et sur les sites visités S1, S2, en considérant que le message de demande de validation n'arrive pas sur S1. La validation de T sur S1 a lieu lorsque le message de demande de validation de T', la

mère de T, arrive sur ce site. Le GT1 peut valider T après avoir interrogé le coordinateur sur l'état de T.

Lors de la validation de T' sur le site S, la transaction T est déjà validée localement, mais elle peut être encore présente dans la *TransList*, en état *en−validation* (elle apparaît dans la liste des filles actives de T'). Le GTc peut alors valider les effets de T' sur S, mais il doit garder T' dans la *TransList*, en état *en−validation*, jusqu'à la disparition de T (jusqu'à ce que la liste de filles actives de T' devienne vide).

Les transitions de l'état d'une sous−transaction validée sur un site visité sont présentés dans la Fig. 6.7. Celles correspondant à une sous−transaction validée sur le site de création le sont dans la Fig. 6.8.

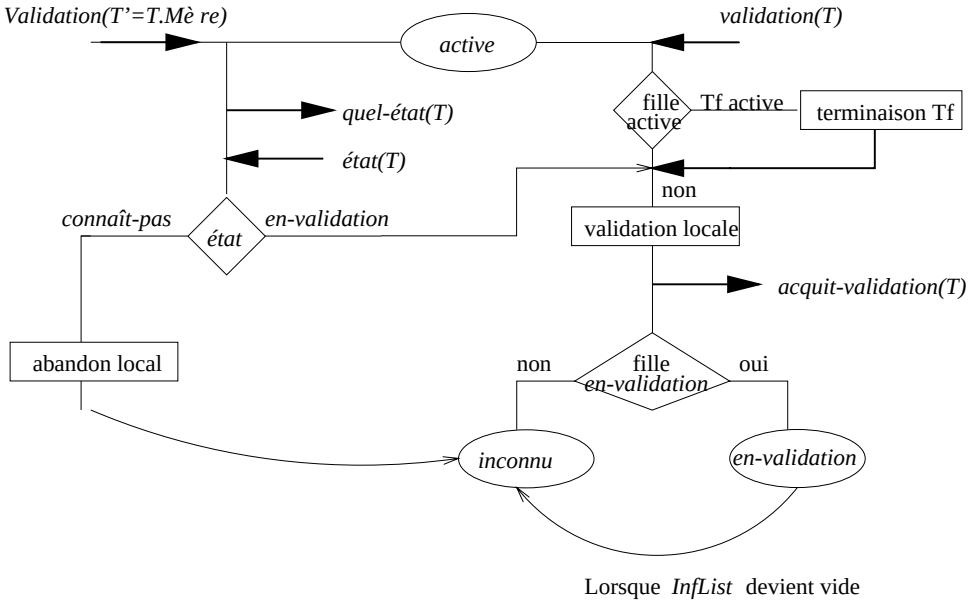

Fig. 6.7 : Validation d'une sous−transaction T sur un site visité

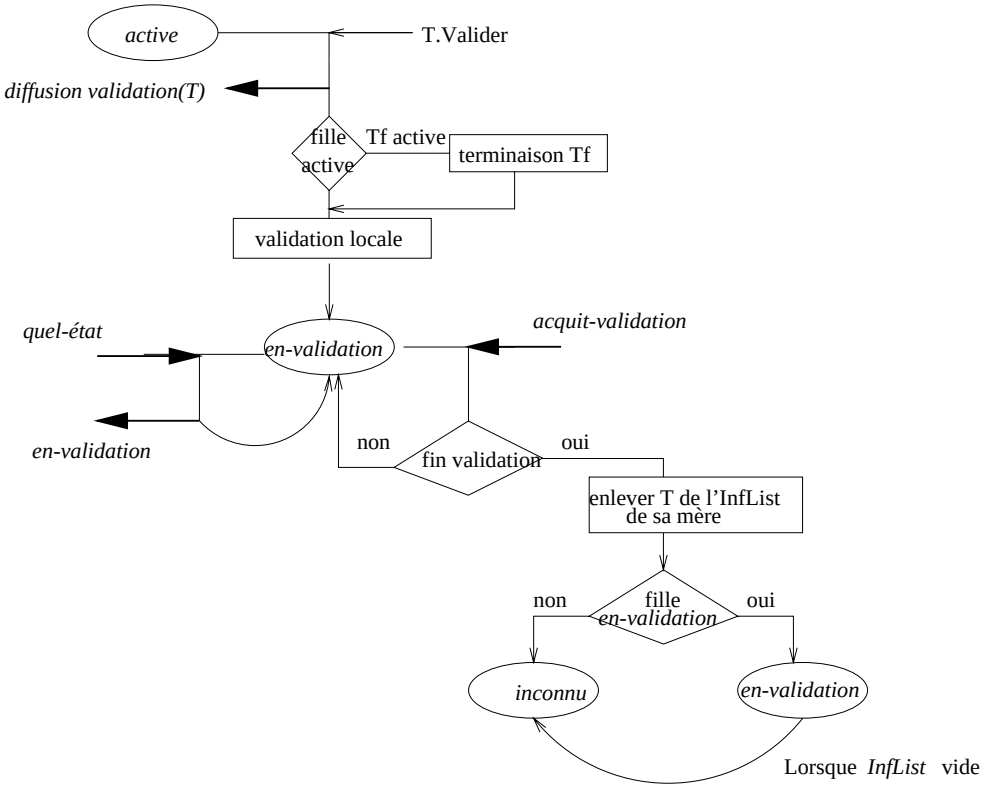

Fig. 6.8 : Validation d'une sous−transaction T sur le site de création

Terminaison d'une transaction racine

La validation d'une transaction racine est mise en œuvre par un protocole classique se déroulant en deux phases. La première phase est bloquante. L'état de l'objet coordinateur passe dans l'état *en−validation* seulement si le GTc reçoit une réponse positive (*prête−à−valider*) de la part de tous les participants. Dans le cas contraire, il diffuse un message de demande d'abandon et oublie T. Si un GTp répond *prête−à−valider*, l'état du représentant est passé en *préparé*. Supposons que le message envoyé par le GTc pour demander la validation ou l'abandon de T, dans la deuxième phase de terminaison, se perd. Le GTp peut alors déterminer le mode de terminaison de T en interrogeant le site coordinateur sur l'état de T. Si le GTc répond *en−validation*, alors T sera validée sur le site visité et un message d'acquittement de validation est renvoyé au GTc. Le GTc garde la transaction dans *TransList*

jusqu'à ce que tous les GTp aient acquitté la validation. Si le GTc répond *connaît−pas*, T est abandonnée.

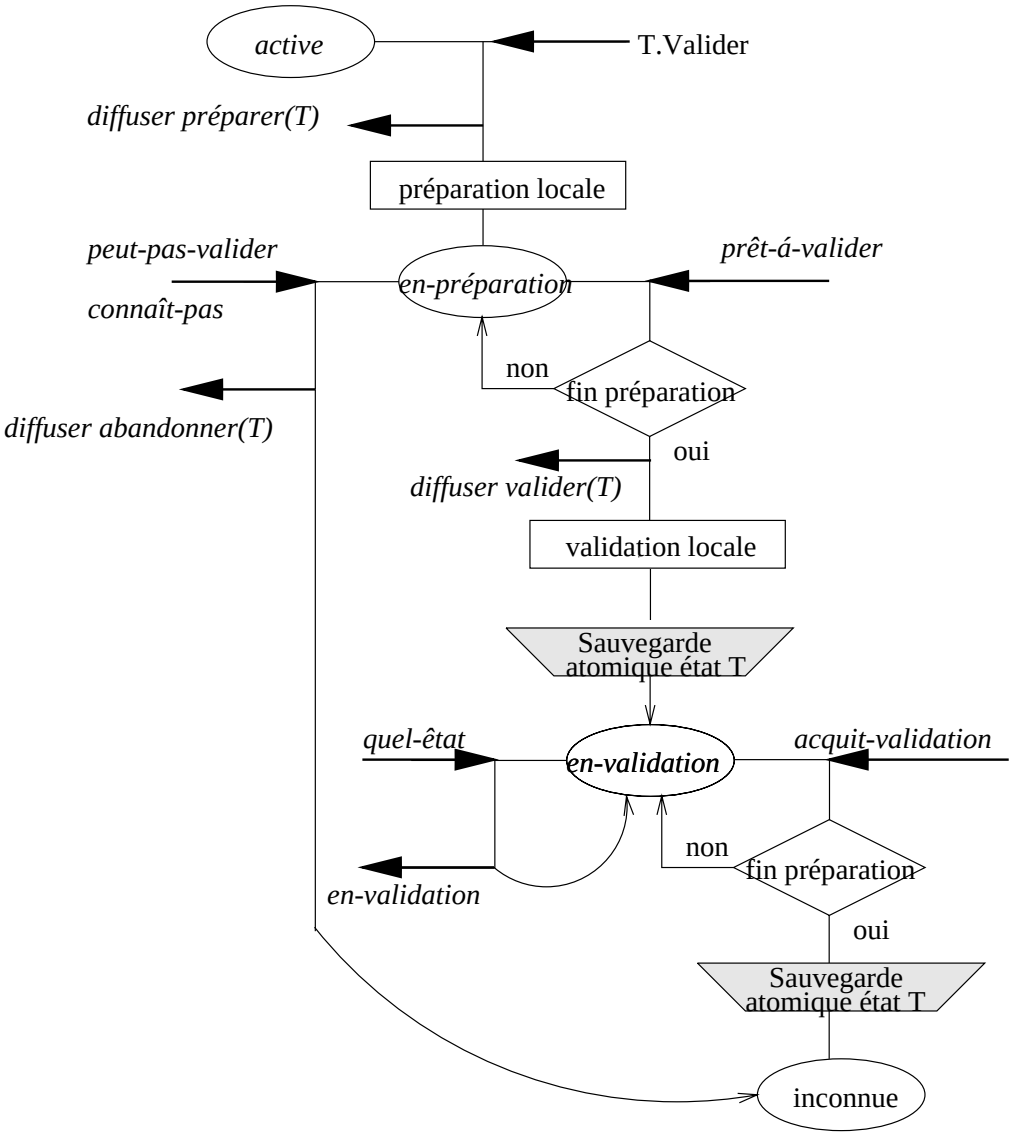

Fig. 6.9 : Transitions d'état d'une transaction racine sur le site coordinateur

VI.4.2.2 Pannes de site

Le problème des pannes de site se pose seulement dans le cas de la validation des transactions racine. En effet, une panne de site implique la perte de la *Trans-List* ; lors du redémarrage du site, le GT local ne connaît plus les transaction qui y ont été enregistrées. Cependant, le GT doit "se rappeler" les transactions créées sur le site qui étaient en état *en−validation*. Une extension est donc nécessaire dans le fonctionnement des GT pour sauvegarder de manière atomique les transactions se trouvant dans cet état.

VI.5 Conclusion

Dans ce chapitre nous avons présenté les mécanismes qui garantissent l'atomicité des transactions réparties en présence des différentes catégories de pannes considérées (V.1). Ces mécanismes sont de deux sortes : ceux implantés par les classes spéciales et ceux mis en œuvre par les Gérants de Transaction (GT).

Le fonctionnement des GT, décrit dans le chapitre V, a été étendu afin d'assurer une gestion fiable des informations de répartition et des informations décrivant les effets des transactions (listes d'intentions).

L'interaction entre les GT et les instances des classes spéciales, pendant le déroulement d'une transaction répartie, a comme but de rassembler les informations qui constituent son historique. Dans la suite, au cours de la terminaison (normale ou due à une panne) de la transaction, l'interaction des composantes garantit son atomicité globale. Les interactions peuvent être locales ou elles peuvent avoir lieu entre composantes qui se trouvent sur des sites distants. L'interaction à distance est limitée aux composantes des GT, qui communiquent via un protocole résistant aux pannes de site et de communication.

Le protocole de communication des GT comporte un algorithme original qui assure l'extension d'une hiérarchie de transactions emboîtées sur les différents sites visités par les transactions composantes. L'approche d'extension mise en œuvre par cet algorithme présente l'avantage de permettre la validation des sous−transactions réparties dans une seule phase. Le protocole de validation des transactions racine se déroule de façon classique, en deux phases.

Chapitre VII

Évaluation

VII.1 Introduction

Afin d'évaluer le Service Transactionnel Guide (STG) de manière qualitative nous présentons une étude comparative avec d'autres prototypes de recherche qui intègrent les concepts de *transaction* et d'*objet atomique*. Nous ne sommes actuellement pas encore en mesure de proposer une évaluation quantitative.

L'analyse comparative est réalisée en prenant en considération trois aspects essentiels de notre travail :

1. L'*approche à base d'objets* pour la mise en œuvre des transactions et des objets atomiques. Nous allons comparer notre proposition avec celles de Avalon/C++ et d'Arjuna.
2. Le *modèle transactionnel* adopté. En partant du modèle des transactions emboîtées défini par Moss, nous rappelons les extensions proposées dans la littérature qui permettent d'augmenter le parallélisme au sein d'une transaction emboîtée. Nous considérons les systèmes étudiés qui intègrent certaines de ces extensions dans leur modèle transactionnel, en mettant en évidence le principe de mise en œuvre des extensions.
3. Le *modèle d'architecture*. Une partie importante du travail que nous avons effectué est représentée par l'étude des conséquences de l'utilisation des transactions emboîtées et des objets atomiques sur l'architecture du service transactionnel. Nous comparons cette architecture avec celle proposé par Moss et avec celle du système Clouds.

VII.2 Évaluation de l'approche à base d'objets

Avalon/C++, Arjuna et le STG sont assez proches dans la manière d'utiliser l'approche à base d'objets. Tous les trois fournissent des classes pré−définies qui rendent possible l'utilisation des objets atomiques et des transactions dans les programmes d'application. Ces classes constituent une interface d'utilisation, mais aussi une approche de mise en œuvre des fonctions nécessaires pour garantir les

propriétés des objets atomiques et des transactions. Dans les trois systèmes, l'héritage est un moyen préconisé pour la mise en œuvre de programmes fiables. Les classes pré−définies fournissent des règles de synchronisation et des propriétés concernant la tolérance aux pannes, dont les objets manipulés par les applications peuvent hériter.

Avalon/C++ a été conçu pour offrir une interface orientée−objet à un système transactionnel déjà existant, le système Camelot. Ceci n'est pas le cas pour Arjuna, dont la hiérarchie des classes fournit toutes les fonctions nécessaires à la définition et à l'exécution des programmes distribués utilisant des transactions et des objets atomiques persistants, sur des plates−formes UNIX. Le STG se situe dans un contexte encore différent, car les classes spéciales visent à améliorer la fiabilité des applications dans un système réparti offrant déjà le support pour la programmation avec des objets persistants.

Les classes *recoverable* et *atomic* en Avalon/C++ correspondent aux classes *StateManager* et *LockManager* en Arjuna, et à la classe *Atomic* fournie par le STG. En Avalon et en Arjuna ces classes mettent en œuvre le support pour la persistance et le contrôle de la concurrence. Étant donné qu'en Guide la persistance des objets est déjà assurée par le système, nous avons regroupé le support pour le contrôle de la concurrence et pour la restauration des objets atomiques dans la classe *Atomic.*

Le système Arjuna et le STG doivent fournir le support pour la cohérence globale des objets atomiques utilisés au sein d'une transaction. Ce support est fourni par les classes *AtomicAction*, respectivement *Transaction.* La mise en œuvre des listes d'intention gérées par les objets de type *Transaction* en Guide est fortement inspirée d'Arjuna. Cependant, en Guide les informations concernant la répartition des transactions ne sont pas gérées par les objets transaction. La mise en œuvre est plus "classique", la gestion de l'information étant assurée par des entités spécialisées, les *Gérants de Transaction*.

VII.3 Évaluation du modèle transactionnel

L'analyse comparative du modèle transactionnel adopté par le STG avec ceux adoptés par différents systèmes supportant les transactions emboîtées est effectuée en prenant en considération :

- le degré de parallélisme intra−transactionnel supporté,
- les autres formes de parallélisme prises en compte par le modèle transactionnel.

VII.3.1 Parallélisme au sein d'une transaction emboîtée

Le modèle défini par Moss [41] prévoit une seule forme de parallélisme au sein d'une transaction, celle des sous−transactions filles. Les sous−transactions n'entrent jamais en conflit avec leurs ancêtres, car les objets sont utilisés par les transactions feuilles uniquement. Par conséquent, une transaction est autorisée à verrouiller un objet déjà verrouillé par un de ses ancêtres dans un mode conflictuel (les transactions mères détiennent des verrous suite à la validation de leurs filles, par le biais d'un mécanisme appelé *héritage ascendant*). Le même modèle est implanté dans Argus [38] et dans Arjuna [45] mais d'une manière différente par rapport à la mise en œuvre proposée par Moss : en Argus et Arjuna les transactions mères peuvent directement utiliser des objets, mais elles sont suspendues durant l'exécution de leurs filles.

Des systèmes comme Camelot [19] et Hermes/ST [21] ont étendu le parallélisme intra−transactionnel, en permettant le parallélisme entre mères et filles. Le parallélisme entre une transaction et ses filles est basé sur la *rétention* de verrous : une fille peut verrouiller un objet déjà verrouillé dans un mode conflictuel par un de ses supérieurs seulement si le verrou posé sur l'objet est en état *retenu* (dans ce cas, le supérieur n'a pas le droit d'utiliser l'objet). Les verrous retenus par une transaction sont obtenus par *héritage ascendant* (ce sont des verrous acquis par les filles validées). Ce mécanisme a été adopté par le STG.

Une solution alternative est proposée dans le système Clouds [16], où les sous−transactions peuvent s'exécuter en parallèle avec leurs supérieurs, sans aucun contrôle ou restriction. Une transaction peut par exemple voir les effets de ses filles non−validées. Ceci est contraire à la définition donnée par Moss, qui stipule qu'une sous−transaction doit être isolée par rapport à ses supérieurs afin que les erreurs qui surviennent au sein d'une sous−transaction ne puissent se répercuter sur ses supérieurs. Allchin propose en [2] de structurer les transactions de manière à ce que les traitements sur les objets partagés soient effectués seulement par les filles dans le cas où l'isolation entre la mère et ses filles est absolument nécessaire.

Afin de permettre un meilleur partage des objets au sein d'une hiérarchie transactionnelle, le système LOCUS [43] met en œuvre un modèle permettant à une transaction d'offrir des verrous à sa hiérarchie. Ceci revient à étendre le champ d'utilisation de la technique de *rétention* de verrous. Une transaction peut ainsi arriver à retenir des verrous non seulement suite à la validation de ses filles, mais aussi en renonçant explicitement au droit d'utiliser des objets qu'elle a verrouillé.

Cette technique est appelée *héritage descendant* par les auteurs de [29]. LOCUS étant un système de gestion de fichiers répartis, les opérations de *prise de verrous* et d'*offre de verrous* sont effectuées dans un cadre spécifique : lors de l'ouverture, respectivement de la fermeture des fichiers [43].

Le STG fournit l'*offre de verrous* comme opération utilisable explicitement lors de la programmation des applications transactionnelles. Le STG va encore plus loin dans l'utilisation de l'héritage descendent : les transactions Guide peuvent offrir un verrou de manière à garder le contrôle sur le mode d'utilisation des objets qu'elles renoncent à utiliser. Par exemple, lorsqu'une transaction offre le verrou en écriture posé sur l'objet O, elle peut restreindre le mode dans lequel les inférieurs utilisent O en lecture (la transaction peut empêcher le verrouillage de O en écriture). Cette technique est appelée *dégradation de verrous* en [29].

Il est enfin possible d'obtenir du parallélisme au sein d'une transaction sans avoir besoin des créer de sous−transaction. Le système Hermes/ST permet la création de processus légers s'exécutant pour le compte d'une même transaction [33]. Afin de garantir la cohérence des objets partagés, le système garantit l'ordonnancement de ces processus. Le STG garantit un comportement similaire pour les activités concurrentes s'exécutant pour le compte d'une même transaction, cependant la mise en œuvre est différente. En Hermes/ST, le mécanisme de contrôle de la concurrence assure l'enchaînement de ces processus par la détection de leur terminaison. En Guide, le mécanisme de contrôle de concurrence ne prend pas en compte les événements de type fin d'activité. À la place, lors de la terminaison d'une requête transactionnelle qui ne crée pas de transactions, le verrou posé sur l'objet cible passe en *retenu*. Suite au changement d'état du verrou, le verrouillage de l'objet re−devient possible lorsqu'il est demandé par une requête s'exécutant pour le compte de la même transaction, qu'elle soit ou non concurrente, avec celle qui vient de se terminer. Il faut noter que dans le système Hermes/ST une transaction ne peut s'exécuter en parallèle avec un supérieur (une sous−transaction T peut verrouiller un objet dans un mode qui génère un conflit avec un de ses supérieurs T', si le processus exécutant T' est suspendu pendant l'exécution du processus exécutant T).

VII.3.2 Autres formes de parallélisme

À notre connaissance, le seul système prenant en compte les exécutions non−transactionnelles est Hermes/ST. De même qu'en Guide, si une transaction et une requête non−transactionnelle entrent en conflit, alors elles sont sérialisées. Deux cas sont possibles. Si le conflit est généré par la transaction, alors elle doit attendre la fin de la requête non−transactionnelle ; dans le cas contraire, la requête

non−transactionnelle est bloquée jusqu'à l'abandon de la transaction ou jusqu'à la validation de sa racine.

Cependant, en Hermes/ST les requêtes non−transactionnelles concurrentes ne sont pas sérialisées. À l'opposé, le STG assure la sérialisation des requêtes non−transactionnelles concurrentes appliquées à des objets atomiques, afin d'assurer la cohérence locale de ces objets.

Dans le système Arjuna, où de même qu'en Guide, la création des transactions et des objets atomiques est effectuée explicitement par les programmes utilisateurs, les objets atomiques sont censés être verrouillés pour le compte d'une transaction (la transaction courante). Les auteurs ne précisent pas les conséquences d'un appel de méthode invoqué sur un objet atomique en dehors d'une transaction.

− si le verrouillage n'était pas permis dans ce cas, alors il s'ensuivrait que les objets atomiques ne peuvent être utilisés hors transaction.

− si le verrouillage était fait pour le compte d'une transaction par défaut, l'atomicité des opérations appliquées aux objets atomiques hors transaction ne serait pas garantie.

VII.4 Évaluation de l'architecture

Nous avons choisi de comparer l'architecture du STG avec celle du prototype fourni par Moss et avec celle du système Clouds. Ce choix est justifié par le fait que Moss a étudié en premier l'impact des transactions emboîtées sur l'architecture d'un système transactionnel distribué. Par ailleurs, l'exemple du système Clouds est intéressant, car il offre un modèle d'architecture complètement réparti et symétrique.

Les trois systèmes ont des caractéristiques communes :

− présence sur chaque site d'un gérant transactionnel (GT) ;

− les GT coopèrent (communication par des messages asynchrones), afin d'assurer l'atomicité des transactions réparties et emboîtées ;

− l'architecture est symétrique (le fonctionnement de chaque GT est le même sur chaque site).

− ils mettent en œuvre le support nécessaire pour la validation locale des sous−transactions réparties : les ancêtres d'une sous−transaction sont représentés sur tous les sites visités par la sous−transaction. Cependant, les trois systèmes ont adopté des approches d'extension indirecte différentes.

Nous analysons les choix de conception des GT et les conséquences de ces choix sur le fonctionnement des protocoles de terminaisons.

VII.4.1 Comparaison avec l'architecture proposée par Moss

De même qu'en Guide, le protocole de communication des GT conçu par Moss est indépendant du système de communication mettant en œuvre les applications réparties.

VII.4.1.1 Spécificités

Les transactions de Moss sont mono−site : lorsqu'une transaction doit effectuer un traitement sur un site distant, son GT crée une sous−transaction, et c'est la sous−transaction ainsi créée qui prend en charge le traitement sur le site distant. Il s'ensuit que le *site de création* d'une sous−transaction peut être différent du site où elle est censée s'exécuter, appelé *site cible*. Les transactions peuvent manipuler des objets uniquement sur leur site cible. Sur les autres sites (appelés sites visités), elles ne peuvent le faire qu'indirectement, à travers leurs inférieurs. Cependant, de même que dans notre modèle, les transactions ont des représentants sur les sites visités, afin que la *terminaison locale* des transactions soit possible.

Gestion de la répartition

Moss a élaboré une approche de gestion centralisée avec extension indirecte différée (cf. VI.2.3) :

• Pour toute transaction mère, son GT gère une liste des sites visités. Cette liste est mise à jour à l'occasion de la création d'inférieurs directs (des filles) et lors de la validation des inférieurs.

− Étant donné que la création d'une sous−transaction distante est décidée par le GT de sa mère, le site cible dune fille est inséré dans la liste des sites visités par la mère, sans que cela nécessite un échange de messages entre les GT. Par conséquent, le GT d'une transaction mère connaît les sites de ses filles actives, mais ne connaît pas les sites des inférieurs actifs de ses filles.

− Afin de compléter la liste des sites visités associée à une transaction mère par les sites de ses inférieurs validés, le GT de toute sous−transaction transmet la liste des sites visités par celle−ci au GT de sa mère, lors de la validation de la sous−transaction.

• Pour toute sous−transaction, son GT reconstitue la hiérarchie de celle−ci, afin de pouvoir effectuer sa validation locale, si besoin est. Cette reconstitution est faite sur la base des informations contenues dans l'identificateur de la sous−transaction. Un identificateur de transaction est obtenu par la concaténation des identificateurs de ses supérieurs avec des informations propres à la transaction, telles que le site de création et le site cible. La reconstitution de la hiérarchie d'une sous−transaction est faite lors de l'exécution d'une première requête pour le compte de celle−ci (c'est à ce

moment que le GT du site cible prend connaissance de l'existence de la sous−transaction et l'enregistre dans ses structures de données).

Exemple. Soit T une transaction mère ayant une fille T1 qui s'exécute sur le site S1. Soit GT1 le gérant du site S1. Lors de la création de T1, le site S1 est inséré dans la liste des sites visités par T, mais T1 n'est pas encore connue par le supérieurs de T. La référence de T1 contient l'identification des sites cibles de ses supérieurs, ce qui permet au GT1 de reconstituer la hiérarchie de T1 lors du premier traitement effectué pour le compte de celle−ci. Si T1 arrive à la validation, le GT1 envoie un *message de validation* contenant la liste des sites visités par T1 au GT de T. Sur réception du message de validation, la liste reçue est fusionnée avec celle de T.

VII.4.1.2 Répercussions sur les protocoles de terminaison

Lors de la validation d'une transaction, la liste des sites visités par ses inférieurs peut être incomplète suite à une panne de communication. Dans l'exemple ci−dessus, S1 peut manquer de la liste des sites visités par la mère de la transaction T, si le GT de celle−ci ne reçoit pas le message de validation de T.

Par conséquent, lors de la validation d'une transaction T, son GT effectue la validation locale et envoie les messages de validation suivants :

− aux GT de sites visités par les inférieurs de T. Le message contient, en plus de l'identificateur de T, la liste des inférieurs validés de T. Moss démontre que ces messages rendent possible la terminaison des inférieurs se trouvant sur des sites qui n'ont pas reçu le message de validation ou d'abandon qui leur était destiné.

− au GT de sa mère, pour lui communiquer la liste des sites visités par T ;

De même, lors de l'abandon d'une transaction T, le message d'abandon doit être envoyé aux GT des sites visités et au GT de sa mère.

Outre les messages de terminaison (de validation et d'abandon), la prise en compte des pannes se fait aux moyen des message d'interrogation d'état, qui peuvent être envoyés par les GT des transactions mères vers les GT des filles, ou vice−versa. L'interrogation d'une mère sert a vérifier que ses filles ont effectivement démarré sur leur site cible, ainsi qu'à détecter les filles abandonnées, ou les filles dont le site est tombé en panne. L'interrogation d'une fille sert à détecter les situations dans lesquelles le message de terminaison envoyé par sa mère est perdu, ou bien elle est devenue orpheline (sa mère est morte).

VII.4.1.3 Conclusion

La stratégie d'extension des transactions proposée par Moss est beaucoup plus simple que la notre, car elle n'entraîne pas l'interaction des GT lors de la création d'une sous−transaction. Cependant, cette stratégie permet la création de sous−transactions à l'initiative des GT. Ceci est contradictoire avec l'objectif que

nous nous sommes fixés lors de la conception du STG : à savoir que les sous−transactions doivent être un moyen souple pour la structuration des applications (leur utilisation correspond aux besoins des applications, pas à des contraintes imposées par le service).

D'autre part, malgré le fait que l'approche d'extension adoptée par nous semble plus lourde, elle permet l'élaboration de protocoles de terminaison plus simples pour les sous−transactions. Par ailleurs, dans l'architecture de Moss, le nombre des sous−transactions créées peut être important, ce qui alourdit par la suite les protocoles de terminaison.

VII.4.2 Comparaison avec l'architecture proposée en Clouds

De même qu'en Guide, Clouds permet la mise en œuvre de transactions "vraiment" distribuées, dans le sens qu'une transaction peut directement utiliser des objets se trouvant sur un site distant. L'objectif est d'éviter la création de sous−transactions pour les appels à distance.

VII.4.2.1 Spécificités

Le modèle d'architecture proposé par Allchin [2] est caractérisé par un couplage fort entre la gestion des transactions et le mécanisme de communication qui met en œuvre les appels de méthode à distance. Ceci est dû au fait que les concepteurs du Clouds ont prévu dés le début l'intégration des facilités transactionnelles dans le système, afin de faciliter le développement des applications fiables.

À l'opposé, le STG a été construit sur un système distribué existant ; par ailleurs l'architecture proposée par Moss est censée être suffisamment générique pour qu'on puisse la mettre en œuvre dans un système distribué quelconque. Ceci justifie dans notre cas, et dans celui de Moss, un choix d'architecture où la gestion des transactions ne s'appuie pas sur la communication au sein des applications.

Gestion de la répartition

L'approche de gestion est centralisée avec extension indirecte différée. Les caractéristiques essentielles du mécanisme permettant de reconstituer la répartition des transactions au niveau de leur site de création sont :

- sur chaque site où une transaction est représentée, on lui associe une liste de sites visités (gérée par le GT local).
- lors d'un appel de méthode à distance effectué pour le compte d'une transaction, le message de retour contient la liste des nouveaux sites visités par la transaction. Les nouveaux sites sont insérés dans la liste gérée localement.
- lors de la validation d'une sous−transaction, sa liste de sites visités est fusionnée avec la liste locale de sa mère.

Grâce à ce mécanisme, si une transaction arrive au point de terminaison, alors la liste des sites visités gérée sur son site de création contient tous les sites qu'elle a directement visités, ainsi que les sites visités par ses inférieurs validés. Cependant, cette liste peut ne pas être complète, car le modèle de programmation autorise la création de sous−transactions asynchrones. Par exemple, si une méthode M, exécutée sur un site distant pour le compte d'une transaction T, crée une sous−transaction asynchrone T1, alors M peut se terminer avant la validation de T1. Dans ce cas, le message retournant le résultat de M arrive sur le site d'origine avant la validation de T1. Par conséquent, il est possible que T arrive au point de validation sans que que le GT de son site de création n'ait pas encore pris connaissance des sites visités par l'intermède de T1. Ceci n'est pas en contradiction avec avec l'approche d'extension indirecte différée, car tant que la sous−transaction T1 n'est pas validée, les sites qu'elle a visités ne doivent pas apparaître dans la liste des sites visités par sa mère T.

En conclusion, lors de la validation d'une transaction, le GT de son site de création connaît tous les sites visités par la transaction et ses inférieurs validés, mais peut ne pas connaître des sites visités par ses inférieurs qui sont encore en cours d'exécution (si de tels inférieurs existent).

VII.4.2.2 Répercussions sur les protocoles de terminaison

Le problème de la connaissance éventuellement incomplète des sites visités par une transaction est résolu par la mise en œuvre d'un protocole de validation à deux phases pour les sous−transactions aussi bien que pour les transactions racines.

La phase de préparation a pour rôle de compléter la liste des sites visités. Le GT coordinateur envoie un message de préparation aux GT des sites visités connus. Lorsqu'un message de demande de préparation arrive sur un site visité, le GT participant attend la terminaison de filles actives avant d'effectuer la validation

locale. Chaque fille qui valide transmet au représentant de la mère sa liste de sites visités. Le GT participant envoie au coordinateur, dans son message d'acquittement, la liste des sites visités par toutes les filles validées.

Il faut noter que la phase de préparation sert aussi à la détection des transactions orphelines (transactions dont la mère a été abandonnée, mais qui continuent encore leur exécution).

La validation des sous−transactions en deux phases pose le problème des sous−transactions bloquées dans l'état *préparée*. Par exemple, l'ordre de validation ou d'abandon peut ne pas arriver sur un site visité par une sous−transaction à cause d'une panne de communication, ou à cause de la panne du site coordinateur. Étant donné que le GT du site visité ne peut prendre une décision unilatérale pour valider ou abandonner la sous−transaction, une solution possible pour débloquer la situation est l'abandon la transaction mère. L'abandon de la mère impose l'abandon de la sous−transaction sur tous les sites, ce qui garantit son atomicité. Dans le cas où l'application veut avoir le contrôle sur le mode de terminaison d'une transaction mère, elle peut créer une sous−transaction tampon entre la transaction mère et ses filles distantes.

VII.4.2.3 Conclusion

La solution adoptée en Clouds pour la gestion de la répartition est intéressante dans le cas où le support transactionnel peut être intégré avec le mécanisme de communication du système. Dans le cas où le modèle de programmation ne permettrait pas l'exécution de sous−transactions asynchrones, le protocole de validation des sous−transactions pourrait être simplifié (réduit à une seule phase), ce qui permettrait d'éviter les problèmes cités ci−dessus.

VII.4.3 Synthèse

Le tableau suivant résume l'analyse comparative des trois architectures présentées, en considérant les caractéristiques suivantes : le modèle d'architecture, le mécanisme utilisé pour rassembler les informations de répartition, l'approche de gestion de la répartition, l'approche d'extension indirecte, et le protocole de validation des sous−transactions.

	Moss	Clouds	STG
Modèle d'architecture réparti	non	oui	oui
Utilisation d'un mécanisme de communication indépendant	oui	non	oui
Approche gestion répartition centralisée	oui	oui	oui
Approche d'extension indirecte	différée	différée	immédiate
Validation sous−transactions	une phase	deux phases	une phase

Conclusion

1 Rappel des objectifs

L'objectif initial de la thèse a été d'améliorer le système Guide en y adjoignant des mécanismes nécessaires au maintien de la cohérence des objets et des applications. Nous nous sommes très rapidement orientés vers le concept de *transaction,* qui a été adopté par un bon nombre de systèmes répartis comme un moyen pratique pour la programmation des applications réparties fiables. En effet, l'exécution des applications en tant que transactions permet aux programmeurs de ne plus se préoccuper des pannes lors de la conception des applications réparties, car celles−ci sont masquées grâce à la propriété d'atomicité des transactions. Un autre avantage des transactions est la protection contre les accès concurrents incontrôlés. Sur ce plan, le langage Guide offrait déjà la possibilité d'associer aux objets des conditions d'activation permettant la synchronisation des flots d'exécution concurrents qui les partagent. En comparaison, le modèle de correction assuré par la sérialisation des exécutions concurrentes est plus puissant, car il offre les moyens pour garantir la cohérence globale d'un ensemble d'objets. Cependant, le modèle transactionnel classique est trop restrictif pour les applications visés par Guide (basées sur la coopération par partage d'objets). Nous nous sommes par conséquent fixés pour objectif la définition d'un modèle transactionnel plus souple.

Nous avons aussi, dès le début, préconisé l'utilisation des objets atomiques pour renforcer la fiabilité des applications Guide. Grâce à leurs propriétés, les objets atomiques facilitent la mise en œuvre des transactions, ainsi que la prise en compte de la sémantique des objets dans la politique de synchronisation et de récupération après pannes.

Cependant, nous nous sommes proposés de définir un modèle transactionnel qui permet de dissocier les deux concepts. Le but était de permettre l'utilisation des objets atomiques en dehors des transactions, mais aussi l'utilisation des objets non−atomiques par les transactions.

Le modèle transactionnel ainsi obtenu doit servir à la mise en œuvre des outils nécessaires au développement et à l'exécution des applications réparties fiables à base d'objets. Ces outils, qui constituent le Service Transactionnel Guide, doivent avoir les propriétés suivantes :

− Leur visibilité doit être assurée au niveau applicatif, mais sans que ceci nécessite des changements au niveau du langage de programmation. L'approche à base d'objets apporte une réponse à ce besoin.

− Les outils doivent être souples, afin de minimiser les contraintes imposées par leur utilisation et de permettre la prise en compte des besoins des utilisateurs sur le plan du degré de cohérence désiré.

− Ils doivent faciliter la programmation des applications complexes. Étant donné que l'utilisation des transactions emboîtées facilite la structuration des applications, nous avons pris la décision de définir un modèle transactionnel avec support pour les transactions emboîtées.

La mise en œuvre des outils permettant l'utilisant des transactions et des objets atomiques nécessite des mécanismes spécifiques. Un des objectifs de la thèse a été l'identification des mécanismes nécessaires pour la fiabilisation des applications. Nous nous sommes employés à identifier les mécanismes déjà existants en Guide, qui devaient être modifiés, ainsi que les nouveaux mécanismes à rajouter dans le système.

2 Bilan du travail réalisé

2.1 Le modèle

Nous avons défini un modèle transactionnel avec support pour les transactions emboîtées qui répond aux objectifs de souplesse fixés.

Ce modèle permet de dissocier les objets atomiques des transactions. Les programmeurs peuvent choisir entre plusieurs degrés de cohérence pour les objets manipulés par leurs applications : pas de cohérence garantie par le service transactionnel pour les objets ordinaires (avec la possibilité d'utiliser les conditions de synchronisation du langage), cohérence locale pour les objets atomiques utilisés hors transaction, et cohérence globale pour les objets atomiques utilisés au sein d'une transaction.

Le modèle défini étend le modèle des transactions emboîtées proposé par Moss, tout en gardant l'isolation des composantes comme garantie de correction. Les extensions apportées font des transactions emboîtées un outil très souple permettant la coopération entre les composantes d'une transaction. Dans notre modèle, la coopération par partage d'objets est possible pas uniquement entre des composantes qui se trouvent au même niveau de leur hiérarchie transactionnelle,

mais aussi entre composantes se trouvant sur des niveaux différents (grâce aux mécanismes d'héritage ascendant et descendant), et encore entre des activités concurrentes s'exécutant au sein d'une même transaction.

De plus, le modèle prend en compte les exécutions concurrentes entre transactions et requêtes non−transactionnelles, ainsi que la concurrence entre requêtes non−transactionnelles.

Notre contribution au domaine consiste en l'intégration dans un modèle homogène de toutes les formes possibles de parallélisme au sein d'une transaction, le parallélisme entre transactions et requêtes non−transactionnelles, ainsi que la concurrence entre requêtes non−transactionnelles. Le modèle est concrétisé par un ensemble de règles de verrouillage et de marquage, qui définissent l'utilisation des objets atomiques à l'intérieur et à l'extérieur des transactions. Le choix entre les règles devant être appliquées est effectué par le système de manière automatique, celui−ci étant capable de déterminer le contexte d'utilisation de tout objet atomique.

2.2 Le Service Transactionnel Guide

Le Service Transactionnel Guide (STG) a été mis en œuvre dans le cadre d'un système existant qui fournit le support pour la programmation à base d'objets persistants et répartis. Il s'agit du système Guide−2, opérationnel sur un réseau de PC 80386 et 80486. Ces machines tournent sous le système d'exploitation OSF−1, construit au dessus du micro−noyau Mach 3.0.

Pour faciliter l'utilisation du service, ainsi que son intégration dans le système, nous avons adopté une approche de mise en œuvre à base d'objets. Cette approche a aussi l'avantage de rendre possible la prise en compte de la sémantique des objets et des besoins des applications sur le plan de la cohérence.

Le STG est caractérisé par une architecture en deux couches. Au niveau supérieur se trouvent les classes spéciales qui constituent l'interface du service et qui mettent en œuvre certaines des fonctions liées au contrôle de la concurrence et à la restauration des objets atomiques. Par exemple, la classe *Atomic* permet la création d'objets atomiques utilisables au sein des transactions ou en dehors des transactions. La classe *Transaction* permet l'utilisation des transactions, avec support pour les transactions emboîtées. Les classes spéciales reposent sur des mécanismes implantés au niveau du noyau Guide.

Si la définition du modèle transactionnel a été faite selon une approche de conception descendante, nous avons du adopter une approche de conception

ascendante pour les mécanismes système, afin de prendre en compte les contraintes imposées par l'architecture du système Guide. Parmi les mécanismes système réalisés nous pouvons mentionner :

- Support pour la mise en œuvre du concept de *transaction courante*. Ce mécanisme, situé au niveau de la machine d'exécution du noyau Guide, permet de définir la portée d'une transaction, participant ainsi au support des transactions emboîtées. Grâce à ce mécanisme, lors de l'invocation d'un objet atomique, le mécanisme de contrôle de la concurrence peut déterminer si l'objet est utilisé hors transaction, où, dans le cas contraire, l'identité de la transaction pour le compte de laquelle l'objet est utilisé.
- Support pour la mise en œuvre du concept d'*activité courante*. Ce mécanisme est nécessaire pour garantir la cohérence locale des objets atomiques partagés par des activités concurrentes s'exécutant au sein d'une transaction.
- Support pour la mise en œuvre de la restauration de l'état initial des objets atomiques modifiés par des transactions qui sont abandonnées à la demande du programme, où dont la racine n'est pas validée à cause d'une panne logicielle ou matérielle. Ce mécanisme est mis en œuvre au niveau de la machine à grappes du noyau Guide.
- Support pour la résistance aux pannes logicielles. Nous avons apporté des modifications au niveau du serveur de stockage du système Guide afin de permettre le couplage des grappes transactionnelles dans un contexte spécial, appelé *contexte transactionnel*. Ce contexte, créé pour toute application transactionnelle, permet l'exécution des algorithmes d'annulation dans le cas d'une panne de contexte.

La gestion des transactions est réalisée par deux sortes de composantes. D'une part nous avons des composantes passives, mises en œuvre par des instances de la classe *Transaction*, et d'autre part des composantes actives − les *Gérants Transactionnels* (GT). Les composantes passives sont chargées de la gestion des effets des transactions, et les composantes actives de la détection des pannes et du bon déroulement des protocoles de terminaison.

Nous avons suivi une démarche progressive dans la conception des GT. Dans un premier temps, nous avons considéré le cas des transactions non−réparties. Dans ce cadre, notre contribution est d'avoir identifié les problèmes et d'avoir proposé des solutions pour la tolérance aux erreurs de programmation dans un système où la communication entre applications se fait par le partage des unités de gros grain

(unités regroupant un ensemble d'objets), lorsque le partage se fait par le couplage de ces unités dans plusieurs espaces d'adressage virtuel.

Dans un deuxième temps, nous avons considéré le cas des transactions réparties et étendu le fonctionnement des GT pour la gestion des informations de répartition des transactions emboîtées. Nous avons étudié les mécanismes nécessaires à la terminaison locale des transactions emboîtées. À cette occasion, nous avons identifié plusieures approches d'extension permettant la reconstitution des hiérarchies transactionnelles sur les sites visités par les différentes composantes d'une transaction. Nous avons aussi étudié l'influence de l'approche d'extension sur les protocoles de terminaison. L'approche d'extension que nous avons adopté est assez contraignante, car elle impose la gestion des informations de répartition complètes à tous les niveaux d'une hiérarchie transactionnelle (l'extension d'une sous−transaction sur un nouveau site est suivie par l'extension immédiate de tous ses supérieurs, avec mise à jour des informations de répartition correspondantes). Son avantage est de permettre la mise en œuvre des protocoles de terminaison en une seule phase pour les transactions emboîtées.

3 Perspectives

Les perspectives à court terme pouvant être dégagées de mon travail de thèse visent à l'évaluation quantitative du modèle d'architecture et aux expérimentations nécessaires pour déterminer à quel point le modèle de concurrence est approprié lors de la conception des applications réelles. Ce dernier point est mis en défaut par l'arrêt du projet Guide.

Par conséquent, dans le long terme, il serait intéressant de porter les classes spéciales sur une autre plate−forme qui fournit un support pour la programmation des applications réparties à base d'objets, par exemple sur une plate−forme avec support pour la programmation en Java. Ce portage devrait comporter deux aspects : la traduction des classes dans le langage de programmation à objets spécifique, et l'étude des besoins au niveau système. C'est ce deuxième point qui devrait poser des problèmes plus difficiles. La partie la plus intéressante devrait être la ré−évaluation de la conception des Gérants Transactionnels. Il faudrait voir, par exemple, si les contraintes imposées par la nouvelle architecture ne mettent pas en cause les choix d'extension des transactions réparties emboîtées. Des changements dans ce sens auraient des conséquences au niveau de la conception des protocoles de terminaison.

Parmi les perspectives à long terme, nous avons identifié les directions d'étude suivantes pour renforcer la souplesse du service :

- Mise en œuvre de nouvelles classes permettant la définition des objets avec des propriétés spécifiques. Par exemple, une classe spéciale pourrait fournir uniquement le support pour la restauration des objets en cas de panne. En effet, le contrôle de concurrence peut ne pas être nécessaire pour des objets accédés uniquement à travers d'autres objets atomiques.
- La définition d'une politique de synchronisation spécifique peut nécessiter l'utilisation d'opérations compensatrices pour l'annulation des effets des transactions abandonnées ou défaillantes. Ceci implique des changements dans la gestion des effets des transactions. Il faudrait donc étudier les besoins liés à ce type de journalisation au niveau des classes spéciales et du système.

Références bibliographiques

[1] M.J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian et M. Young, ‘‘Mach: a new Kernel Foundation for Unix Development’’, *Proc. of the USENIX 1986 Summer Conference*, pp. 93−112, Juillet 1986.

[2] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, (GIT−ICS−83/23), Georgia Institute of Technology, Atlanta, Georgia 30332, September 1983.

[3] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville et G. Vandôme, ‘‘Architecture and Implementation of Guide, an Object−Oriented Distributed System’’, *Computing Systems (Usenix)*, vol.4(1), pp. 31−67, 1991.

[4] R. Balter, S. Lacourte et M. Riveill, ‘‘The Guide Language’’, *The Computer Journal*, 37(6), pp. 519−530, Décembre 1994.

[5] P. A. Bernstein et N. Goodman, ‘‘Concurrency Control in Distributed Database Systems’’, *ACM Computing Surveys*, 13(2), pp. 185−221, Juin 1981.

[6] P.A. Bernstein, V. Hadzilacos et N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison−Wesley, Massachusets, 1987.

[7] C. Boksenbaum, M. Cart, J. Ferrié et J−F. Pons, ‘‘Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems’’, *IEEE Transactions on Software Engineering*, 13(4), pp. 409−419, Avril 1987.

[8] M. Cart et J. Ferrié, ‘‘Integrating Concurrency Control into an Object−Oriented Database System’’, *Advances in Database Technology − EDBT ’90 (Proceedings of the International Conference on Extending Database Technology)*, édité par F. Bancilhon C. Thanos D. Tsichritzis, pp. 363−377, LNCS 416, Venice Italy, March 1990.

[9] M. Cart, J. Ferrié et H. Richy, ‘‘Contrôle de l’exécution de transactions concurrentes’’, *Techniques et Science Informatique*, 8(3), pp. 225−240, Août 1987.

[1 0] P.Y. Chevalier, *Persistence et disponibilité dans les systèmes répartis*, Thèse de doctorat, Université Joseph Fourier, Grenoble 1, Octobre 1994.

[1 1] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte et X. Rousset de Pina, ‘‘Experience with Shared Object Support in the Guide System’’, *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems*, San−Diego, Septembre 1993.

[1 2] P.Y. Chevalier, D. Hagimont, J. Mossière et X. Rousset de Pina, ‘‘Le système réparti à objets Guide’’, *Techniques et science informatique*, 15(6), pp. 801−830, Juin 1996.

[1 3] R.C. Chen et P. Dasgupta, ‘‘Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation’’, *Proceedings of the Ninth International Conference on Distributed Computing Systems*, Newport Beach CA, June 1989.

[1 4] P.K. Chrysanthis et K. Ramamrithan, ‘‘ACTA: a framwork for specifying and reasoning about transaction behavior’’, *Readings in Database Systems*, édité par Michael Stonebraker, pp. 335−350, Morgan Kaufmann Publishers, San Francisco, California, 1994.

[1 5] G. Coulouris, J. Dollimore et T. Kindberg, *Distributed Systems Concepts and Design*, Addison−Wesley, 1994.

[1 6] P. Dasgupta, R.J. LeBlanc Jr., M. Ahamad et U. Ramachandran, ‘‘The Clouds Distributed Operating System’’, *IEEE Computer*, 24(11), pp. 34−44, November 1991.

[1 7] D.L. Detlefs, M.P. Herlihy et J.M. Wing, ‘‘Inheritance of synchronization and recovery properties in Avalon/C++’’, *IEEE Computer*, 21(12), pp. 57−69, December 1988.

[1 8] G.N. Dixon, G.P. Parrington, S.K. Shrivastava et S.M. Wheater, ‘‘The Treatement of Persistent Objects in Arjuna’’, *The Computer Journal*, 32(4), pp. 323−332, August 1989.

[1 9] J.L. Eppinger, L.B. Mummert et A.Z. Spector, *Camelot and Avalon−A Distributed Transaction Facility*, Morgan Kaufmann, San Mateo, California, 1991.

[2 0] K.P. Eswarn, J.N. Gray, R.A. Lorie et I.L. Traiger, ‘‘The Notions of Consistency and Predicate Locks in a Database System’’, *Communications of the ACM*, 19(11), pp. 624−633, Novembre 1977.

[2 1] M. Fazolare, B. G. Humm et R. D. Ranson, ‘‘Concurrency Control for Distributed Nested Transactions in Hermes/ST’’, *Proceedings of the 1993 International Conference On Paralel and Distributed Systems*, National Taiwan Univerisy, Taipei, Taiwan, Republic of China, Decembre 1993.

[2 2] J.N. Gray, ‘‘Notes on database operating systems. Operating Systems−An advanced Course’’, *Lecture Notes in Computer Science 60*, édité par R. Bayer, R.M. Graham et G. Seegmueller, pp. 393−481, Springer−Verlag, Berlin, 1978.

[2 3] J.N. Gray, P. M. Jones, M. Blasgen, B. G. Lindsay, R. A. Lorie, T. Price, F. Putzolu et I. L. Traiger, ‘‘The Recovery Manager of the System R Database Manager’’, *ACM Computing Surveys*, 13(2), pp. 223−242, Juin 1981.

[2 4] J.N. Gray et A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[2 5] M. Guerni, *L’impact des objets typés sur le modèle transactionnel à effets différés*, Thèse de doctorat, Université de Montpellier II, Sciences et Techniques du Languedoc, Octobre 1995.

[2 6] D. Hagimont, *Adressage et protection dans un système réparti*, Thèse de doctorat, Institut National Polytechnique de Grenoble, Octobre 1993.

[2 7] D. Hagimont, P.Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossière et X. Rousset de Pina, ‘‘Persistent Shared Object Support in the Guide System: Evaluation & Related Work’’, *9th ACM Conference on Object−Oriented Systems, Languages and Applications (OOPSLA)*, Octobre 1994.

[2 8] T. Harder et A. Reuter, ‘‘Principles of Transaction−Oriented Database Recovery’’, *ACM Comput. Surv.*, 15(4), pp. 287−317, Decembre 1983.

[2 9] T. Harder et K. Rothermel, ‘‘Concurrency Control Issues in Nested Transactions’’, *VLDB Journal*, 2(1), pp. 39−74, 1993.

[3 0] R. Haskin, Y. Malachi, W. Sawdon et G. Chan, ‘‘Recovery Management in QuickSilver’’, *ACM Transactions on Computer Systems*, 6(1), pp. 82−108, February 1988.

[3 1] M. P. Herlihy et J. M. Wing, ‘‘Avalon: Language Support for Reliable Distributed Systems’’, *Proceedings of the Seventh International Symposium on Fault−Tolerant Computing*, pp. 89−94, July 1987.

[3 2] M. P. Herlihy et W. Weihl, ‘‘Hybrid Concurrency Control for Abstract Data Types’’, *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Août 1986.

[3 3] B.G. Humm, ‘‘An Extended Scheduling Mechanism for Nested Transactions’’, *Proceedings of The Third International Workshop on Object Orientation in Operating Systems*, édité par L−F Cabrera et N. Hutchinson, pp. 125−134, IEEE Computer Society Press, Decembre 1993.

[3 4] H. F. Korth, ‘‘Locking Primitives in a database system’’, *Journal of the ACM*, 30(1), pp. 55−57, Janvier 1983.

[3 5] H. T. Kung et J. T. Robinson, ‘‘On optimistic methods for concurrency control’’, *ACM Transactions on Datbase Systems*, 6(2), pp. 213−226, Juin 1981.

[3 6] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin et X. Rousset de Pina, ‘‘Design and Implementation of an Object−Oriented Strongly Typed Language for Distributed Applications’’, *Journal of Object−Oriented Programming*, 3(3), pp. 11−22, Octobre 1990.

[3 7] B. Lampson, ‘‘Atomic Transactions’’, *Distributed Systems: Architecture and Implementation*, édité par Goos and Hartmanis, pp. 246−265, Springer Verlag, Berlin, 1981.

[3 8] B. Liskov, ‘‘The ARGUS language and system. Distributed systems−Methods and tools for specification: An advanced course’’, *Lecture Notes in Computer Science 190*, édité par M. Paul and H.J. Siegert, pp. 343−430, Springer−Verlag, Berlin, 1985.

[3 9] B. Liskov et R. Scheifler, ‘‘Guardians and Actions: Linguistic Support for Robust, Distributed Programs’’, *ACM Transactions on Programming Languages and Systems*, 5(3), pp. 381−404, Juillet 1983.

[4 0] C. Malta, *Les systèmes transactionnels pour environements d’objets : Principes et mise en œuvre*, Thèse de doctorat, Université de Montpellier II, Sciences et Techniques du Languedoc, Octobre 1993.

[4 1] J.Eliot B. Moss, ‘‘Nested Transactions and Reliable Distributed Computing’’, *Proceedings of the Symposium on Reliability in Distributed Sofrware and Database Systems*, pp. 33−39, July 1982.

[4 2] J.Eliot B. Moss, *Nested Transactions : An approach to Reliable Distributed Computing*, MIT Press, 1985.

[4 3] E.T. Mueller, J. D. Moore et G. J. Popek, ‘‘A Nested Transaction Mechanism for LOCUS’’, *Proceedings of the 9th Symposium on Operating System Principles*, pp. 71−85, ACM/SIGOPS, Bretton Woods, 1983.

[4 4] G. D. Parrington, ‘‘Reliable Distributed Programming in C++ : The Arjuna Approach’’, *Second Usenix C++ Conference*, pp. 37−50, San Francisco, Avril 1990.

[4 5] G. D. Parrington et S. K. Shrivastava, ‘‘Implementing Concurrency Control in Reliable Distributed Object−Oriented Systems’’, *Proceedings of the Second European Conference on Object−Oriented Programming*, édité par ECOOP88, pp. 233−249, Oslo, Norway, August 1988.

[4 6] R. J. Peterson et J. P. Strickland, ‘‘LOG Write−Ahead Protocols and IMS/VS Logging’’, *Proceedings of the Second ACM SIGACT News−SIGMOD Symposium on Principles of Database Systems*, pp. 216−243, Mars 1983.

[4 7] P. M. Schwarz et A. Z. Spector, ‘‘Synchronizing Shared abstract types’’, *ACM Transactions on Computer Systems*, 2(3), pp. 223−250, Août 1984.

[4 8] S. K. Shrivastava, G. N. Dixon, G. D. Parrington., ‘‘An Overview of the Arjuna Distributed Programming System’’, *IEEE Software*, pp. 66−73, January 1991.

[4 9] J.S.M. Verhofstad, ‘‘Recovery Techniques for Database Syatems’’, *ACM Computing Surveys*, 10(2), pp. 167−195, Juin 1978.

[5 0] B. Walker et G. Popek, ‘‘The LOCUS Distributed Operating System’’, *Proc. 9th ACM Symposium on Operating System Principles*, pp. 49−70, Bretton Woods, Octobre 1983.

[5 1] B. Walter, ‘‘Nested transactions with multiple commit points: An approach to the structure of advanced database appliacations’’, *Proceedings of the Tenth International Conference on VLDB*, Singapore, 1984.

[5 2] W. E. Weihl, ‘‘Data−dependent concurrency control and recovery’’, *Proceedings of the 2nd ACM SIGACT−SIGOPS Symposium on Principles of Distributed Computing*, pp. 63−124, Montreal, Août 1984.

[5 3] W. E. Weihl, *Specification and Implementation of Atomic Data Types*, Thèse, Massachusetts Institute of Technology, Mars 1984.

[5 4] W. E. Weihl et B. Liskov, ‘‘Implementation of Resilient, Atomic Data Types’’, *ACM Transactions on Programming Languages and Systems*, 7(2), pp. 244−269, Avril 1985.

[5 5] J. M. Wing et D. C. Steere, ‘‘Specifying Weak Sets’’, *Proceedings of the International Conference on Distributed Computing Systems*, June 1995.

[5 6] Z. Wu, R. J. Stroud, K. Moody et J. Bacon, ‘‘The design and implementation of a distributed transaction system based on atomic data types’’, *Distributed Systems Engineering*, 2(1), pp. 50−64, March 1995.

Introduction

Chapitre I

Transactions et objets atomiques : utilisation et mise en œuvre

Chapitre II

Guide

Chapitre III

Les caractéristiques générales du Service Transactionnel Guide

Chapitre IV

Le contrôle de la concurrence

Chapitre V

L'atomicité

Chapitre VI

Atomicité des transactions réparties

Chapitre VII

Évaluation

Conclusion