Les transactions étendues et quelques Frameworks qui les supportent.

Christophe Ponsen
cponsen@info.fundp.ac.be

Institut d'Informatique, Université de Namur

Résumé Les transactions étendues posent de nombreux problèmes si l'on veut qu'elles respectent les propriétés ACID. Plusieurs Frameworks existent pour tenter de supporter ces transactions tout en apportant des solutions aux problèmes spécifiques qu'elles amènent. Ce papier rappelle ce que sont les transactions étendues, propose l'analyse détaillée d'un Framework les supportant et jette un regard sur un autre.

1 Les transactions

1.1 Les transactions ACID

Il est connu et reconnu que les transactions respectant les propriétés ACID[1] ne conviennent pas pour construire des applications dont certaines actions ou demandes sont relativement longues dans le temps ou encore lorsqu'une application veut donner les propriétés ACID a des actions très indépendentes entre elles. En effet, les contraintes que les transactions doivent respecter pour être ACID sont tellement fortes qu'elles empêchent les développeurs de faire ce qu'ils veulent. On a alors vu apparaître différents méchanismes permettant d'assouplir ces règles tout en donnant parfois l'impression que les contraintes ACID étaient toujours respectées. Parmis ces mécanismes, on peut citer les *nested transactions* qui offrent un méchanisme permettant de hiérarchiser des transactions tout en garantissant que la plus élevée dans la hiérarchie respecte les contraintes ACID. Ces mécanismes apportent de la souplesse, mais aussi la possibilité d'avoir des erreurs dans des transactions sans pour autant que le système en tienne compte et donc, qu'il garde sa cohérence. En effet, dans le cas des *nested transactions*, même si la transaction la plus élevée est ACID, les transactions internes peuvent ne pas être ACID et donc corrompre le système.

Malgré l'apparition des techniques comme les *nested transactions* qui permettent d'assouplir la rigidité des propriétés ACID, certains problèmes ne pouvaient toujours pas être résolus. En effet, on peut considérer les transactions normales et les *nested* comme des entités ayant un temps de vie très court et effectuant de simple changement d'états stables dans le logiciel. Ces transactions ne conviennent absolument pas pour des tâches demandant de très long temps

[1] Atomocity,Consistency,Isolation,Durability

de traitement ou la manipulation d'entités logiques indépendantes. De manière simple, la technique demandant de bloquer les ressources utilisées par une transaction afin de permettre le respect des propriétés ACID ne peut convenir dans le cas de longs traitement (plusieurs heures) si cette ressource est une ressource pouvant (ou devant) être utilisée par une autre tâche importante. De plus, dans le cas ou une transaction qui a déjà duré quelques heures venait à subir un échec, il est plus que probable que tout le travail réalisé lors de cette transaction soit totalement ou en partie perdu, ce qui peut représenter une perte très importante pour une organisation, tant en temps qu'en argent. Enfin, lors d'une transaction utilisant des entités logiques totallement indépendantes, il est parfois utile de pouvoir valider une partie du travail fait sur une entité sans pour autant, si la transaction globale venait à échouer, revenir en arrière pour toutes les entités. Il peut aussi être utile de pouvoir permettre aux entités en présence de connaître à tout moment l'état de la transaction afin d'adapter leur comportement en fonction de cet état.

D'autres exemples montrant l'utilité d'avoir des transactions différentes peuvent se retrouver dans [I. 03a] p.353.

1.2 Les transactions étendues

Les améliorations apportées aux transactions ACID pour leur donner un peu de souplesse ne suffisent donc pas pour soutenir le fonctionnement des tâches évoquées ci-dessus. Ces mêmes tâches ont régulièrement besoin de voir les (ou des) contraintes ACID relâchées afin de permettre leur bon fonctionnement voir leur fonctionnement tout court (i.e. la possibilité de partager une ressource sans pour autant avoir terminé son travail sur cette même ressource, de communiquer avec une entité extérieure à la transaction pour pouvoir compléter celle-ci,...). On peut en effet considérer ces applications, durant leur fonctionnement, comme des transactions étendues non ACID. Beaucoup de recherches ont été menées afin d'établir des modèles de ces transactions mais la plupart des techniques nées de ces modèles n'ont pas connu un grand succès. Ces techniques étaient trop adaptées à un seul modèle (à un seul type de transaction étendue) alors qu'il en existe plusieurs. Aujourd'hui encore, il n'existe pas de solution intégrée permettant de résoudre toutes les difficultés de ce type de transaction.

Les figures 1 et 2 donnent un exemple d'une transaction étendue représentée comme un ensemble de transactions indépendantes mais qui, en cas d'échec de l'une d'entre elle, comme le montre la deuxième figure, entraîne un changement de comportement de la transaction étendue. On pourrait par exemple imaginer un système de souscription à des jeux en fonction des gains récupérés par le jeu précédent. Le joueur demande que tant que l'on gagne, on joue au jeu suivant en misant l'argent gagné au jeu précédent. L'échec pourrait être un jeu où le joueur perd. Dans ce cas, il peut demander un ensemble d'activités pour compenser la perte ou arrêter de jouer. Au final, l'ensemble de la transaction se termine quand

certaines conditions requises sont remplies. Chacune des activités peut être vue comme une transaction élémentaire, mais l'ensemble des activités effectuées fait partie d'une seule transaction étendue dont le comportement (en cas d'échec ou réussite d'une transaction élémentaire) est adaptatif.

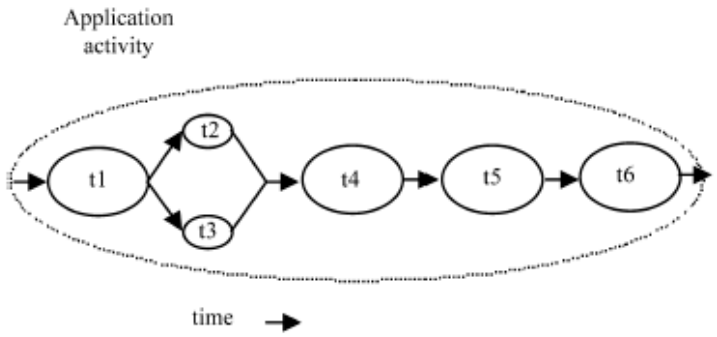

Fig. 1. Example de transaction étendue [I. 03b]

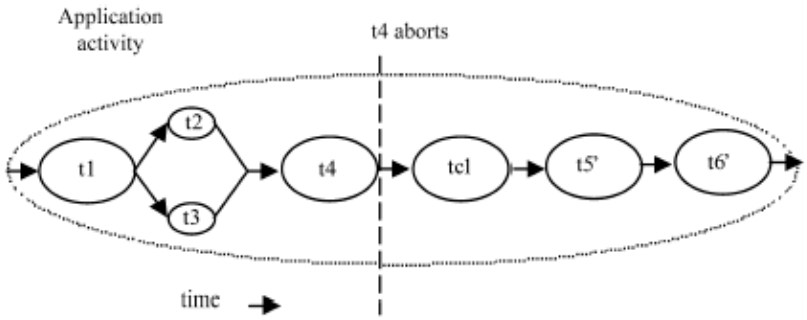

Fig. 2. Example de transaction étendue avec un échec [I. 03c]

1.3 Comment gérer une transaction étendue

Les transactions étendues ont principalement besoin de ne pas être totalement ACID, du moins, à certains moments de leur exécution alors qu'idéalement, la transaction vue dans son ensemble devrait l'être. En effet, la nécessité de ne pas réserver une ressource trop longtemps, de valider le travail déjà effectué au

fur et à mesure, de permettre le déroulement de transactions en parallèle, ..., sont autant de points qui obligeront un développeur à ne pas respecter une ou plusieurs propriétés ACID. On voit donc apparaître la nécessité de gérer les propriétés ACID d'une transaction au niveau de l'application plutôt qu'au niveau du système. Il faut pour se faire que le développeur puisse prendre en compte toutes les opérations effectuées lors d'une transaction étendue et mette en place un ensemble de mécanismes permettant de donner à la transaction les propriétés ACID tout en permettant, pendant son exécution, de ne pas l'être à tout moment. Ainsi, des mécanismes de récupération des erreurs sont les bienvenus, mais ne sont pas toujours nécessaires, des opérations de synchronisations entre objets accédés sont nécessaires, mais pas toujours au même moment, des opérations de rollback sélectifs doivent pouvoir être faites alors qu'une partie seulement de la transaction a été réalisée. Bref, l'ensemble des opérations assurées normalement par le système afin de rendre une transaction ACID doivent être implémentées par le développeur de manière explicite.

1.4 Comment alléger le travail des développeurs

La majorité des théories en informatiques débouchent sur des outils permettant d'exploiter ces théories. Plusieurs techniques de programmation ont déjà été proposée par les chercheurs mais aucune n'a vraiment connu un grand succès car, comme nous l'avons dit ci-avant, chacune d'elle était prévue surtout pour un type de transaction étendue. Pour qu'une technique ou un outil connaisse un grand succès, il fallait essayer de rencontrer plusieurs attentes des développeurs, à savoir par exemple :

- Simplifier l'écriture des méthodes de gestion des propriétés ACID ;
- Offrir un maximum de méthodes permettant de gérer au mieux ces propriétés ACID ;
- Rendre ces méthodes les plus transparentes et les moins contraignantes possible sans quoi, les développeurs préféreront réécrire leurs propres méthodes.

Etant donné que ces transactions se déroulent le plus souvent dans le cadre d'applications distribuées et parce que permettre de gérer ces transactions dans un cadre restreint à une machine est un cas particulier d'un cadre distribué, le mieux serait de voir l'ensemble de ces fonctionnalités regroupées au sein d'un même Framework distribué.

2 Différents Frameworks

A l'heure actuelle, il existe plusieurs Frameworks, certains étant dédiés explicitement aux transactions normales (voir *nested*), d'autres aux transactions étendues. Ce travail propose de voir en détail le Framework inclus dans CORBA et d'en survoler un autre en prêtant attention à leur capacité à supporter les transactions étendues et a leur prise en main.

2.1 l'Activity Service Framework de CORBA

Présentation. Fort de toutes ces observations, CORBA propose un Framework générique permettant de gérer les transactions étendues. Loin d'être une solution complète, ce Framework n'en est pas moins un outil précieux pour soutenir ce genre de transactions. En effet, il propose un mécanisme complet qui, associé à des méthodes que doit fournir le développeur, permet de gérer les transactions étendues.

Principes de base. Le principe de base est on ne peut plus simple mais demande un niveau d'abstraction des concepts de l'application. Le Framework se propose de traiter toute tâche comme une activité, ces activités pouvant elles-même contenir des activités. Les activités s'enregistrent auprès d'un coordinateur qui leurs permet de s'échanger des messages. Chaque activité possède un état qui peut influencer sur les messages envoyés ou reçus. Certains états sont bloquants, d'autres non. Le Framework prend en charge, notemment en utilisant le serveur de transaction de CORBA, certains aspects de sérialisation ou de sauvegarde d'objets pour permettre la restauration de l'état du système en cas d'échec ou de crash sévère, mais pas tous vu que les transactions étendues requierent de relâcher par moment certaines contraintes ACID. Certains objets, certains états ne peuvent être gérés par le Framework directement, notamment les suites d'enchaînements de transactions élémentaires au sein d'une transaction étendue. Le Framework offre diverses possibilités permettant de gérer facilement tout cela, ces possibilités sont expliquées ci-après.

Architecture. Le Framework travaille donc avec des activités, ces activités sont découpées en actions (Actions)[2] qui s'enregistrent auprès d'un coordinateur d'activité (Activity Coordinator). Chaque action peut demander à pouvoir utiliser un ensemble de signaux (Signal Set) en s'enregistrant auprès du coordinateur pour cet ensemble spécifique de messages. On entend par utilisation le fait de pouvoir envoyer des messages ou être notifié quand un message appartenant à ce groupe de signaux apparaît. Le Framework ne propose pas l'échange entre deux actions autrement qu'en déclarant un ensemble de signaux spécifiques aux deux actions uniquement. On retrouve ici le mécanisme complet du pattern publish-subscribe. Le coordinateur joue un rôle central puisque c'est lui qui interagit directement avec les ensembles de signaux pour les envoyer aux actions qui se sont enregistrées.

On peut donc voir l'architecture du service comme le montre la figure 3.

[2] une activité peut-être découpée en plusieurs actions ou être une seule action, ce choix est laissé au développeur

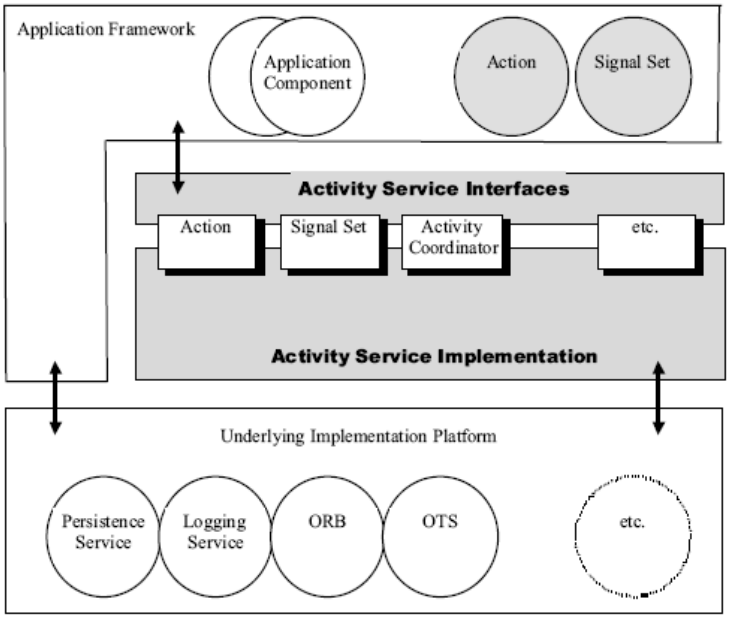

Fig. 3. Architecture du service Activity [I. 03d]

Propriétés du Framework. Seul le développeur a connaissance de la manière dont l'application va utiliser les données, dont les données doivent être bloquées (n'oublions pas que les transactions étendues relâchent souvent la contrainte d'isolation des requêtes) dans certains cas et enfin, comment les activités doivent traiter les échecs. Toutes ces informations pourraient être codées directement dans l'application, mais alors, si une modification devait être faite, tout l'application devrait être reconstruite. Le Framework permet de définir ces propriétés par activité par le biais d'un espace de paires de valeurs. On peut très facilement changer les propriétés de l'application au travers du Framerwork et cela au niveau des activités.

Traitement des échecs et des procédures de recouvrement. Le Framework introduit une manière très originale de traiter les échecs. Plutôt que de devoir définir des procédures spécifiques en cas d'échec d'une activité, il suffit de définir les instructions a effectuer dans une ou plusieurs activités, de les enregistrer auprès du coordinateur avec le bon ensemble de signaux. Cette manière orginale de traiter les échecs permet une gestion complète de ceux-ci sans compliquer le travail du développeur puisque celui-ci n'a plus qu'à définir des ensembles de signaux et des activités.

En ce qui concerne les procédures de recouvrement, le Framework propose un ensemble de procédures automatiques, semi-automatique et d'autres manuelles.

Les procédures automatiques sont les procédures classiques qui s'appliquent à tout objet défini auprès du service sous-jacent de transaction. Cependant, toutes les activités ne peuvent s'enregistrer auprès de se service, de même que certaines transactions étendues qui ne respectent pas les propriétés ACID. Dans ce cas, le Framework permet d'enregistrer une multitude d'informations sur ces objets et laisse le soin au développeur d'écrire les procédures permettant de remettre le système en état à partir des informations qu'il aura demandées au Framework et des informations qu'il aura sauvegardées par le biais de l'application pour les objets que le Framework ne peut gérer seul.

Exemple et conclusion. La figure 4 montre un exemple concret de l'utilisation du Framework. Cet exemple est une implémentation du protocol two-phase commit en utilisant les signaux, les ensembles de signaux et les actions.

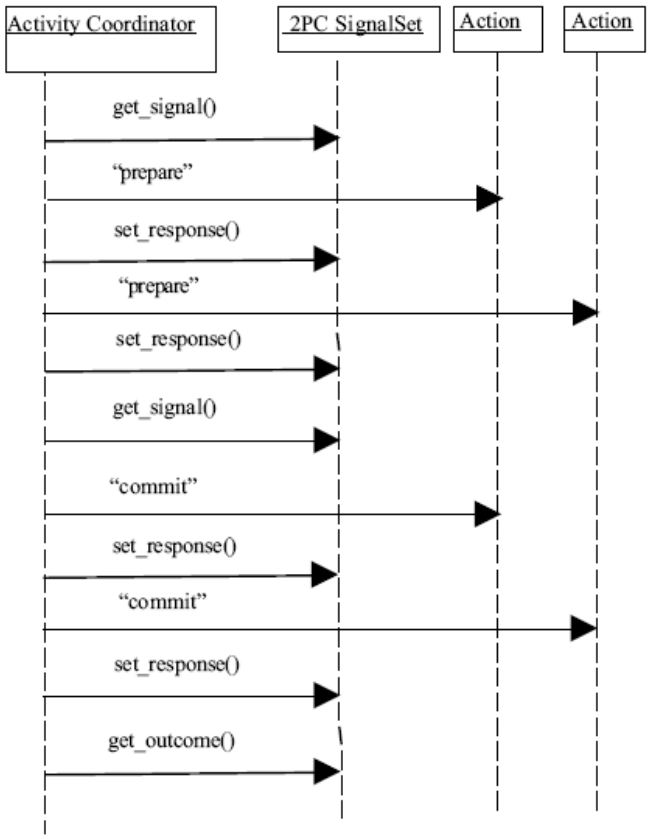

Fig. 4. Example d'implémentation, le Two-phase commit [I. 03e]

L'ensemble des concepts mis en oeuvre ici sont génériques et, s'ils ont été implémentés dans ce service CORBA, ils ont été repris depuis pour être intégrés dans d'autres architectures (J2EE notamment). De plus, on peut réutiliser ces concepts pour écrire ce service et l'utiliser avec un autre middleware que CORBA. En effet, aucun concept développé ici n'est attaché à la technologie CORBA.

Autre élément en faveur de ce Framework, il propose une architecture n'imposant aucun modèle de transaction étendue, mais permet bien de supporter au moins une majorité d'entre eux. Il suffit de modifier les ensembles de signaux pour qu'ils respectent les spécifications des modèles.

Le Framework semble donc supporter pleinement les transactions étendues et s'il ne peut à lui seul gérer le recouvrement en cas d'arrêt brutal du système, il propose suffisamment de méthodes et de mécanismes permettant au développeur de prendre en charge ces cas extrêmes de la manière la plus simple possible.

2.2 Le Framework ACTA

Présentation. ACTA est un Framework théorique et générique permettant de gérer les transactions étendues (appelées complexes dans le Framework). A la différence du service de CORBA, le Framework ACTA n'est pas un Framework de développement, mais un Framework permettant de modéliser et raisonner sur les transactions.

Principes de base. Ce Framework n'ayant pas pour but la réalisation d'une application utilisant des transactions étendues, les concepts mis en oeuvres sont plus orientés recherche et réfléxion que programmation. En effet, ACTA propose de définir les transactions complexes de manière indépendante (et non liée à une application), d'en définir les propriétés et de raisonner sur leurs comportements et leurs interactions.

Architecture. Dans le cadre d'un Framework théorique, on ne parlera pas d'architecture, mais plutôt de dimensions. La figure 5 montre ainsi que le Framework propose de se baser avant tout sur l'ensemble des effets qu'une action au sein d'une transaction peut avoir sur la transaction elle-même ou sur les objets accédés ou utilisés. On voit le Framework comme un analyseur d'intéractions, ce qui permet de raisonner sur ces même interactions.

Le raisonnement sur ces transactions se fait via la structure mathématique du Framework qui permet de définir les transactions via des règles de représentation et d'interactions.

Exemple :
Commit-Dependency : Si une transaction A développe une *commit-dependency*

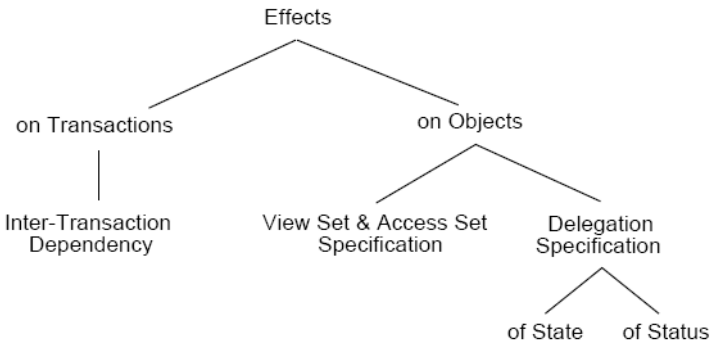

Fig. 5. Dimensions d'ACTA [Pan91a]

vis-à-vis d'une autre transaction (noté $A \rightsquigarrow B$), alors la transaction A ne peut effectuer sont commit tant que la transaction B n'a pas effectué soit un commit soit un abort. Cela n'implique pas que si la transaction B échoue, alors la transaction A doit aussi échouer. *Transitive-commit-dependency* (notée $\overset{*}{\rightsquigarrow}$) est défini par une close transitive de dépendance de commit. Une transaction A a une transitive-commit-dependency avec chaque membre du groupe de transactions formé par la close transitive de dépendance de commit à partir de A. [Pan91b]

On peut déclarer les relations de dépendances par des règles de production. Ainsi, on peut représenter la règle de production permettant d'ajouter un fils à un parent dans une *nested transaction* par :

$$\begin{matrix} P \vdash P \\ T \vdash C \end{matrix} \left\{ \begin{matrix} P \rightsquigarrow C \\ C \rightarrow P \end{matrix} \right\}$$

où : $\vdash$ = symbol de renomination
C = une transaction fille
P = une transaction parente
T = une transaction
A $\rightarrow$ B = A possède un *abort-dependency* sur B

Traitement des échecs et des procédures de recouvrement. Le traitement des échecs n'est pas pris en compte de manière explicite puisqu'il s'agit ici d'un Framework de représentation et de raisonnement. On indique les liens entre transactions, les effets et les états de chacunes et si elles effectuent un commit ou un abort pour terminer, mais pas les actions spécifiques à effectuer dans le cas d'un abort ou d'un commit. Toute action à prendre en cas de commit ou abort est elle-même une nouvelle transaction qui intervient alors dans le schéma.

Quant aux procédures de recouvrement, ces procédures étant intrinsèquement liées à un système et non à la représentation des transactions, elles sont sans ob-

jet dans ce Framework.

Conclusion. Ce Framework est le plus puissant présenté dans cet article car il permet une représentation mathématique complète de toutes les interactions possibles entre transactions complexes au travers des définitions des entités et des règles de représentations et de production permettant de mettre ces entités en relations.

Il est cependant orienté vers la recherche exclusivement car n'est pas implémentable et ne permet pas de créer directement une application à partir des spécifications introduites dans le système.

3 Conclusion

Les deux Frameworks présentés sont deux des Frameworks les plus connus actuellement. Même si ACTA est déjà ancien, il n'en reste pas moins d'actualité et a certainement permis la réalisation des Framework comme CORBA ou celui du J2EE. Si le Framework CORBA se veut un outil d'aide à l'implémentation alors qu'ACTA est un outil d'aide à la recherche et au raisonnement, il semble évident que la combinaison des deux est un moyen sûr de permettre à une application de fonctionner correctement avec des transactions étendues. Il est en effet évident que, si les transactions sont définies dans ACTA, si l'on y considère toutes les interactions possibles ainsi que les règles de production dont l'application va avoir besoin et si on les valide dans ACTA, l'implémentation de ces transactions dans un Framework tel que l'Activity Service de CORBA n'est plus alors que l'écriture du code d'une analyse complète et correcte. Le fait de travailler avec les deux Framework demanderait plus de travail de la part du développeur, mais assurerait une correction et une robustesse sans pareille de l'application.

Références

[I. 03a] I. Houston, M.C Little, I. Robinson, S.K. Shrivastava and S.M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *SOFTWARE-PRACTICE AND EXPERIENCE*, 33 :351–373, 2003.

[I. 03b] I. Houston, M.C Little, I. Robinson, S.K. Shrivastava and S.M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *SOFTWARE-PRACTICE AND EXPERIENCE*, 33 :354, 2003.

[I. 03c] I. Houston, M.C Little, I. Robinson, S.K. Shrivastava and S.M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *SOFTWARE-PRACTICE AND EXPERIENCE*, 33 :355, 2003.

[I. 03d] I. Houston, M.C Little, I. Robinson, S.K. Shrivastava and S.M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *SOFTWARE-PRACTICE AND EXPERIENCE*, 33 :356, 2003.

[I. 03e] I. Houston, M.C Little, I. Robinson, S.K. Shrivastava and S.M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *SOFTWARE-PRACTICE AND EXPERIENCE*, 33 :365, 2003.

[Pan91a] Panayiotis K. Chrysanthis and Krithi Ramamritham. A Unifying Framework for Transactions in Competitive and Cooperative Environnements. *IEEE Office Knowledge Engineering*, 4(1) :6, February 1991.

[Pan91b] Panayiotis K. Chrysanthis and Krithi Ramamritham. A Unifying Framework for Transactions in Competitive and Cooperative Environnements. *IEEE Office Knowledge Engineering*, 4(1) :7, February 1991.

[Pan91c] Panayiotis K. Chrysanthis and Krithi Ramamritham. A Unifying Framework for Transactions in Competitive and Cooperative Environnements. *IEEE Office Knowledge Engineering*, 4(1) :2–21, February 1991.