Traitement de requêtes réparties : des modèles à leur implémentation[1].

N. Boudjlida et Y. Belhamissi

CRIN - Bat. LORIA - Université de Nancy I, B.P. 239

54506 - Vandoeuvre Les Nancy Cedex (France)

E-mail: nacer@loria.fr ou belhamis@loria.fr

Résumé : Cet article présente une expérimentation de la gestion de transactions dans des bases de données distribuées. Nous rappelons, dans un premier temps, le principe de traitement de requêtes réparties, celui de la validation à deux phases (*Two-Phase Commit Protocol*) et les problèmes posés par la panne d'un des sites impliqués dans la transaction répartie. Dans un second temps, nous montrons les possibilités offertes par un système commercial, Sybase en l'occurrence. Nous concluons en montrant les limites de ce système et en énonçant quelques préceptes méthodologiques dans le développement et le contrôle des transactions réparties.

Mots clés : bases de données distribuées, transaction répartie, transparence, two-phase commit protocol, remote procedure call, site coordinateur, fragmentation.

Abstract : This article presents an experimentation in the area of transaction management in distributed database systems. First, we give an overview of distributed transaction management, with an emphasis on the two phase commit protocol and on how to deal with site failures. Then, we'll present how this protocol is implemented in the commercial database management system Sybase. Finally, we'll point out some limits of this implementation and give some design precepts to deal with the development and the control of distributed transactions.

Keywords : distributed databases, distributed transactions, transparency, two-phase commit protocol, remote procedure call, commit site, fragmentation.

1 Introduction

Dans un système de gestion de bases de données (SGBD) réparties, comme dans un système centralisé, un des objectifs de la gestion des transactions est de préserver *l'atomicité* de celles-ci

[1] In Actes des Journées Internationales des Sciences Informatiques, JISI'94, Tunis, Tunisie, Mai 1994.

(*"Politique du tout ou rien"*, c'est-à-dire, validation (*COMMIT*) ou annulation (*ROLLBACK*) de toutes les actions réalisées par la transaction). Dans un contexte réparti, l'exécution d'une transaction peut impliquer différents sites compte tenu de la localisation des données. En effet, une transaction T peut être décomposée en sous-transactions T_i, chaque T_i s'exécutant sur un site S_i. Dans ce cas, il est clair que des validations locales et indépendantes ne sont pas acceptables dans la mesure où une ou plusieurs des transactions T_i peuvent avoir avorté. Dans ce contexte, un des protocoles de validation communément utilisé est le protocole de validation à deux phases (*Two-Phase Commit Protocol*). Le but de cet article est de montrer une implémentation de ce protocole sur un SGBD commercial, Sybase en l'occurrence. Nous rappellerons, d'abord, le principe du traitement des requêtes réparties (paragraphe 2), le fonctionnement du protocole de validation à deux phases, ainsi que les problèmes liés à son implémentation (paragraphe 3). Le paragraphe 4 est dédié à la description du service du SGBD Sybase pour la gestion de transactions réparties. Enfin, nous conclurons par quelques remarques relatives à l'expérimentation menée et au chemin qui reste à parcourir pour aller des modèles de traitement de requêtes réparties à leur implémentation .

2 Traitement des requêtes réparties

Nous appelons requête répartie un traitement qui implique plus d'un site du système distribué. Le paragraphe 2.1 introduit le traitement de telles requêtes et le paragraphe 2.2 l'illustre par un exemple.

2.1 Traitement de requêtes réparties

Dans un SGBD relationnel, le rôle dévolu au sous-système de traitement des requêtes est d'engendrer un plan d'exécution supposé optimisé. Dans un contexte centralisé, les principales techniques, non exclusives, d'optimisation sont :

- celles dites *syntaxiques* [12] qui opèrent par transformation d'une représentation sous forme d'arbre de la requête. Ce type d'approche utilise les propriétés formelles des opérateurs de l'algèbre relationnelle couplées avec des heuristiques qui consistent à appliquer le plus tôt possible les projections (opérations réduisant la taille des relations *"en largeur"*) et les sélections (opérations réduisant la taille des relations *"en hauteur"*).

- celles basées sur des modèles de coûts et utilisant des informations statistiques généralement rangées dans le dictionnaire de la base.

- et enfin, celles dites *sémantiques* [8] qui tentent d'exploiter les contraintes d'intégrité pour simplifier les requêtes.

Dans un cadre réparti, le traitement des requêtes doit prendre en compte les informations relatives à la distribution (fragmentation et localisation) et/ou la duplication des données sur les différents sites. Diverses stratégies sont candidates à l'exécution (jointures parallèles, semi-jointures [2], etc). Néanmoins, afin de satisfaire la propriété de *transparence* de la distribution des données (i.e. l'utilisateur *doit "avoir l'impression"* de travailler sur une base centralisée), il est nécessaire que le système décompose une requête en sous-requêtes en fonction de la localisation des données. Les sous-requêtes peuvent alors être adressées aux sites concernés pour y être exécutées. Le système devra ensuite être à même de contrôler que toutes les sous-requêtes se sont bien achevées, de constituer le résultat final à partir des résultats des sous-requêtes, dans le cas d'une transaction de lecture, et de valider les effets de toutes les sous-requêtes dans le cas d'une transaction de mise à jour.

Ainsi, un *plan d'exécution* [7], dans un contexte réparti, devra comporter les informations nécessaires aux exécutions locales des sous-requêtes et celles nécessaires à leur coordination. L'exemple qui suit illustre le fonctionnement d'un tel type de système.

2.2 Un exemple de traitement de requête répartie

Considérons deux relations R et S ayant un attribut A en commun. Supposons que R et S soient fragmentées horizontalement[2] sans redondance [9] en, respectivement, *R.f1, R.f2, R.f3* et *S.f1, S.f2*, les relations-fragments pouvant être physiquement sur des sites différents. Supposons que les expressions de fragmentation soient[3] :

- $R.f1 = \sigma_{A \leq 50}(R)$, $R.f2 = \sigma_{50 < A \leq 80}(R)$, $R.f3 = \sigma_{A > 80}(R)$,
- et $S.f1 = \sigma_{A \leq 50}(S)$, $S.f2 = \sigma_{A > 50}(S)$.

Soit également la requête SQL :

*SELECT * FROM* R, S
WHERE R.A = S.A

[2]Rappelons que la fragmentation horizontale consiste a "découper" une relation en sous-relations, chaque sous-relation contenant un sous-ensembles de tuples de la relation initiale. Rappelons, également, que la relation initiale est reconstituée, dans ce type de fragmentation, par l'union de ses fragments.

[3]$\sigma, U et \bowtie$ désignent, respectivement, la sélection, l'union et la jointure relationnelles

La démarche consiste à représenter la requête sous forme d'arbre d'opérateurs de l'algèbre relationnelle en y incorporant les expressions de reconstruction des relations fragmentées (figure 1).

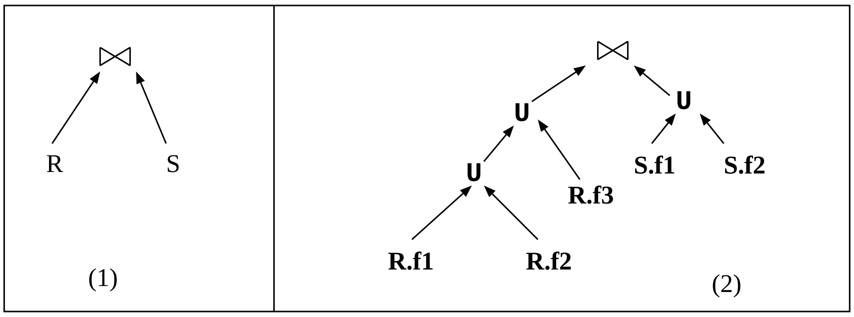

Figure 1 : Arbre initial et arbre avec expressions de reconstruction

Par la suite, les heuristiques mentionnées ci-dessus sont applicables. On appliquera également des règles de simplification liées à la fragmentation. Ainsi,

- un sous-arbre comportant une projection appliquée à un fragment vertical ne comportant pas les attributs de la liste de projection sera éliminé.
- Il en est de même pour les sous-arbres où une condition de sélection ou une expression de jointure est en contradiction avec la ou les expressions de fragmentation.

Dans l'exemple considéré, en utilisant la distributivité de la jointure et de l'union de relations, nous obtenons :
$(R.f1 \bowtie S.f1) \cup (R.f1 \bowtie S.f2) \cup (R.f2 \bowtie S.f1) \cup (R.f2 \bowtie S.f2) \cup (R.f3 \bowtie S.f1) \cup (R.f3 \bowtie S.f2)$.

Puis, nous éliminons les sous-expressions $(R.f1 \bowtie S.f2), (R.f2 \bowtie S.f1), (R.f2 \bowtie S.f2)$ et $(R.f3 \bowtie S.f1)$, en exploitant les contradictions entre les expressions de jointure et les expressions de fragmentation. Nous aboutissons ainsi, à l'arbre de la figure 2.

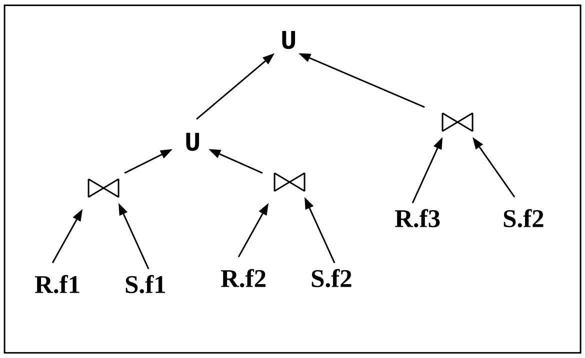

Figure 2 : Arbre optimisé

Un exemple de stratégie d'exécution serait alors d'opter pour des jointures parallèles ou, de jointures parallèles sur le sous-arbre gauche et de semi-jointure sur le sous-arbre droit[4]. Le rôle du système de traitement de requêtes est aussi de coordonner l'exécution. Ce rôle est primordial lorsqu'il s'agit de transactions de mise à jour. En effet, les mécanismes centralisés de gestion des transactions ne sont évidemment plus applicables, dans la mesure l'invalidation d'une sous-requête devra entraîner l'avortement de toutes les sous-requêtes impliquées dans la transaction.

Le paragraphe qui suit montre le principe et le fonctionnement du protocole de validation à deux phases, communément utilisé dans ce contexte.

3 Protocole de validation à deux phases

Le protocole de validation à deux phases (*Two-Phase Commit Protocol*) [6] assure la validation atomique des transactions réparties. Pour qu'une transaction répartie soit validée et que ses effets deviennent permanents, le protocole doit obtenir l'accord de tous les sites participants à la transaction. Un site est privilégié par ce protocole : le *coordinateur* (voir la figure 3).

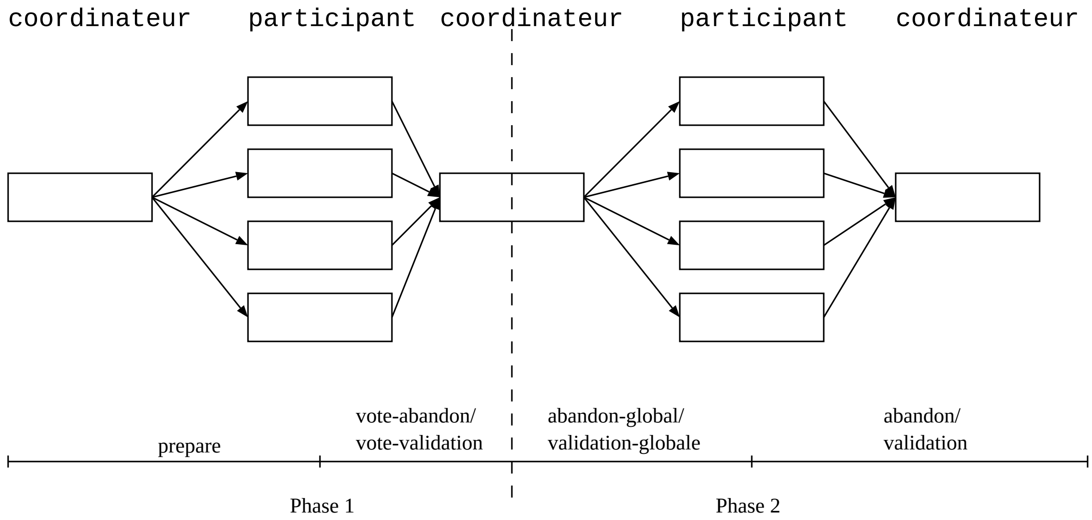

Figure 3 : Le protocole de validation à deux phases

Le site coordinateur journalise le *début de la transaction*. Ensuite, le protocole se déroule comme suit :

- Le site coordinateur envoie un message *prepare* à tous les sites participants et entre dans une phase d'attente.

[4]Il est clair que d'autres facteurs, tels le choix des sites d'exécution [1], peuvent intervenir à ce stade. Mais nous ne les considérerons pas ici

- Lorsqu'un site participant reçoit un message *prepare*, il vérifie s'il peut valider la transaction. Si le test est positif, il envoie un message *vote-validation* au site coordinateur et entre dans un état *prêt*. En revanche, si la transaction ne peut pas être validée, il envoie un message *vote-abandon* au coordinateur et considère que la transaction est annulée car il suffit qu'elle soit abandonnée sur l'un des sites pour qu'elle le soit globalement.

- Quand le coordinateur reçoit les messages de réponse de tous les sites participants, il décide d'annuler la transaction si au moins l'un des messages reçus est *vote-abandon* et de la valider dans le cas où tous les participants ont envoyé des messages *vote-validation*. Le coordinateur envoie des messages *abandon-global* ou bien *validation-globale* selon la décision prise.

- Les sites participants abandonnent ou valident la transaction selon le message reçu et envoie un accusé de réception au site coordinateur.

Pour tolérer les pannes de sites, un protocole de terminaison doit être associé au protocole de validation à deux phases. A chaque site, que ce soit un coordinateur ou un participant, doit être associé un *timeout* (chien de garde). Celui-ci doit être déclenché, sur un site, à chaque fois que la réception d'un message attendu n'a pas lieu au cours d'un intervalle de temps fixé. Les traitements associés au déclenchement des timeouts sont différents suivant le rôle du site dans le protocole. Par exemple, si un site participant tombe en panne après avoir reçu le message *prepare*, mais avant d'avoir envoyé sa réponse au site coordinateur, ce dernier, ne voyant pas arriver le message en question, pourra décider l'annulation globale de la transaction.

Ce mécanisme nécessite des envois de messages entre les différents sites. Cette communication inter-sites est généralement, réalisée par un mécanisme d'appel de procédures distantes (*Remote Procedure Call ou rpc*) [3]. La sémantique de l'appel de procédures distantes est identique à celle de l'appel de procédures locales : l'appelant est bloqué, la procédure appelée est exécutée et les résultats sont rendus à l'appelant qui peut alors continuer son exécution.

L'implémentation du mécanisme d'appel de procédures distantes est illustré par la figure 4. Les paramètres de l'appel sont mis sous la forme d'un ou de plusieurs paquets de messages. Les paquets sont ensuite acheminés, via réseau de communication, vers le site appelé. Ils y sont reçus et "dépaquetés". La procédure appelée est exécutée et les résultats reprennent le chemin inverse jusqu'au site appelant où ils seront délivrés à la procédure appelante.

Ces mécanismes ont fait l'objet de diverses implémentations dans les SGBD commercialisés. Le paragraphe suivant analyse celle offerte par le SGBD Sybase pour en montrer le principe et les limites.

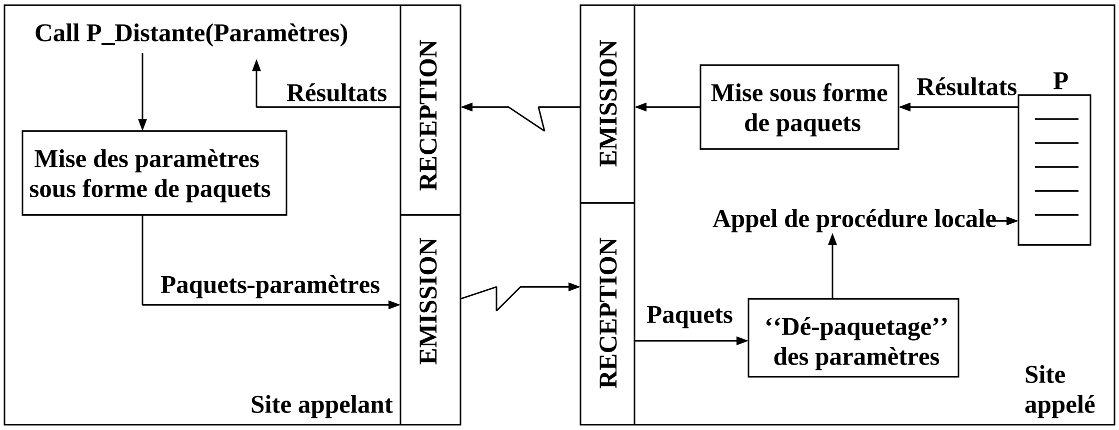

Figure 4 : Principe de l'appel de procédures distantes

4 Gestion des transactions réparties dans Sybase

SYBASE [10] est un système de gestion de bases de données dont l'installation peut comporter un ensemble de serveurs SQL répartis sur plusieurs sites de calcul. Chaque serveur prend en charge un ensemble de bases de données. SYBASE offre une interface entre les applications écrites dans un langage hôte et les serveurs de bases de données. Cette interface, appelé DB-Library [11], se présente comme un ensemble de macros et de fonctions permettant la communication avec un serveur SYBASE à partir d'un programme utilisateur. Il est possible d'exécuter des transactions réparties sur SYBASE, la DB-Library offrant, à cet effet, un ensemble de fonctions permettant la programmation du protocole de validation à deux phases.

4.1 Principe

Le programme utilisateur doit, au départ, ouvrir une session avec chacun des serveurs impliqués dans la transaction. Une transaction répartie est effectuée par l'envoi de requêtes SQL aux différents serveurs par l'intermédiaire de la DB-Library. La validation de la transaction répartie est programmée par l'utilisateur en utilisant les fonctions de la DB-Library qui implémentent le protocole de validation à deux phases.

L'un des serveurs SYBASE doit jouer le rôle du coordinateur de validation, il est appelé *serveur de commit* (figure 5).

- L'application envoie à chaque serveur une requête BEGIN TRANSACTION comprenant des informations d'identification de l'application, de la transaction et du serveur de commit.
- Elle envoie ensuite les requêtes SQL à chacun des serveurs. Ces requêtes sont, par exemple, des opérations de mise à jour de bases de données.

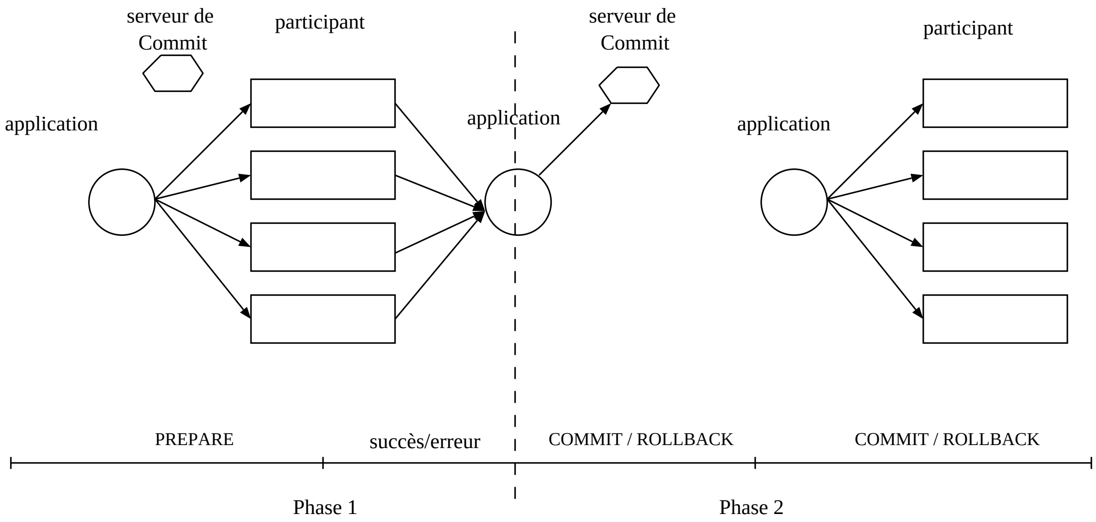

Figure 5 : Le protocole de validation à deux phases dans SYBASE

- L'application entame la première phase du protocole de validation en envoyant une requête PREPARE à chacun des serveurs participants à la transaction.
- Si la commande est exécutée avec succès sur tous les serveurs, lors de la seconde phase, l'application demande au serveur de commit de noter la validation de la transaction.
- L'application doit envoyer pour finir une requête COMMIT TRANSACTION ou bien ROLLBACK TRANSACTION aux serveurs participants qui doivent respectivement valider ou annuler la transaction.

Si une panne survient sur l'un des serveurs ou si l'application échoue avant le PREPARE TRANSACTION alors le serveur coordinateur annule la transaction. Si l'un des serveurs tombe en panne après le PREPARE TRANSACTION, mais avant le COMMIT TRANSACTION alors le serveur en question doit communiquer avec le serveur de commit pour savoir si la transaction doit être validée ou annulée.

Une fois que le site de commit a noté que la transaction doit être validée, elle le sera sur tous les serveurs et quelle que soit l'erreur qui puisse survenir au niveau de l'application ou au niveau des serveurs.

Le rôle de l'application utilisateur est de fournir les requêtes SQL aux différents serveurs en utilisant les routines de la DB-Library prévues à cet effet. Le rôle du serveur de commit est de fournir un emplacement unique où la décision de validation/annulation de la transaction sera maintenue. Les autres serveurs ne rentrent en communication avec le serveur de commit que dans le cas où une erreur survient.

4.2 Exemple

L'extrait de programme C suivant est un exemple d'utilisation de la DB-Library pour programmer une transaction répartie sur deux serveurs SYBASE. L'un des serveurs joue le rôle de serveur de commit. Le but de l'exemple n'est pas de fournir un programme complet mais de montrer les différentes étapes de la validation de la transaction.

La communication avec les serveurs se fait par l'intermédiaire d'une structure de données créée lors de la connexion avec les serveurs (étape non explicitée dans l'exemple). Dans la suite, *dbproc-server1* et *dbproc-server2* seront les structures de données liées respectivement aux serveurs *serveur1* et *serveur2*. De même, la structure *dbproc-commit* sert pour la connexion avec le serveur de commit. Les commandes *dbcmd* et *dbsqlexec* servent, respectivement, à constituer et à exécuter des commandes dans une zone *cmdbuf* situé dans la structure de données *dbproc-server$_i$*.

Une fois les connexions avec les serveurs établies, l'application doit signaler le début de la transaction aux différents serveurs ainsi qu'au serveur de commit (dans ce cas, *serveur2*).

```
/* - DEBUT DE LA TRANSACTION REPARTIE - */
sprintf(cmdbuf, "BEGIN TRANSACTION", xact-string);
/*—— Débuter la transaction avec le serveur 1 */
dbcmd(dbproc-server1, cmdbuf);
dbsqlexec(dbproc-server1);
/*—— Débuter la transaction avec le serveur 2 */
dbcmd(dbproc-server2, cmdbuf);
dbsqlexec(dbproc-server2);
/*—— Débuter la transaction avec le serveur de commit */
scan-xact(dbproc-commit, commid);
```

Les serveurs doivent maintenant exécuter les requêtes SQL qui sont, par exemple, une requête de mise à jour sur une base de données repliquée sur les deux serveurs. La commande *dbsqlexec* rend un résultat par lequel l'application peut savoir si les requêtes ont été exécutées avec succés ou non. S'il y a eu échec sur au moins l'un des serveurs, la transaction est annulée. La fonction *abort-xact* informe le serveur de commit de l'annulation de la transaction. La commande ROLLBACK TRANSACTION est utilisée pour annuler la transaction sur chacun de serveurs et restaurer l'état antérieur au début de la transaction. L'application se termine par la fonction *dbexit*.

```
/* - ENVOI DES REQUETES SQL AUX DIFFERENTS SERVEURS - */
/*—— Envoi des requêtes SQL au serveur 1 */
sprintf(cmdbuf, " REQUETE SQL");
dbcmd(dbproc-server1, cmdbuf);
ret-server1 = dbsqlexec(dbproc-server1);

/*—— Envoi des requêtes SQL au serveur 2 */
sprintf(cmdbuf, " REQUETE SQL");
dbcmd(dbproc-server2, cmdbuf);
ret-server2 =dbsqlexec(dbproc-server2);

/*—— Test d'erreur dans l'exécution des requêtes SQL */
if (ret-server1 == FAIL || ret-server2 == FAIL)
{
/*—— Erreur dans l'exécution de la transaction. */
/*———— Informer le serveur de commit de l'annulation de la transaction*/
abort-xact(dbproc-commit, commid);

/*———— Annuler la transaction sur les différents serveurs */
sprintf(cmdbuf, "ROLLBACK TRANSACTION");
dbcmd(dbproc-server1, cmdbuf);
if (dbsqlexec(dbproc-server1) != FAIL)
remove-xact(dbproc-commit, commid, 1);
dbcmd(dbproc-server2, cmdbuf);
if (dbsqlexec(dbproc-server2) != FAIL)
remove-xact(dbproc-commit, commid, 1);
dbexit();
}
```

Si les requêtes SQL ont été exécutées avec succès sur tous les serveurs, la première étape du protocole de validation débute par l'envoi des requêtes PREPARE TRANSACTION. Le retour de cette requête doit indiquer si le serveur est prêt à valider la transaction. Si une erreur survient à ce niveau, le même traitement d'annulation de transaction que précédemment est exécuté.

```
/*- ENVOI DES MESSAGES PREPARES - */
sprintf(cmdbuf, "PREPARE TRANSACTION");
```

```
dbcmd(dbproc-server1, cmdbuf);
dbcmd(dbproc-server2, cmdbuf);
ret-server1 = dbsqlexec(dbproc-server1);
ret-server2 = dbsqlexec(dbproc-server2);
if (ret-server1 == FAIL || ret-server2 == FAIL)
{
/*—— Erreur dans la phase de préparation : invalider la transaction. */
/*———— Séquence de traitement d'annulation */
}
```

Tous les sites sont prêts à valider la transaction, l'application demande au serveur de commit de noter que la transaction est validée. S'il y a erreur au cours de cette étape, l'application doit exécuter le traitement d'annulation.

```
/*- VALIDATION DE LA TRANSACTION SUR LE SERVEUR DE COMMIT - */
if (commit-xact(dbproc-commit, commid) == FAIL)
{
/* Le serveur de commit n'a pas pu enregistrer l'état de validation de la transaction. */
/*———— Séquence de traitement d'annulation */
}
```

Le serveur de commit a notifié que la transaction est valide, il reste à l'application d'en informer les serveurs participants en leur envoyant la requête COMMIT TRANSACTION. Le serveur de commit maintient le nombre total de serveurs participants à la transaction, l'application doit envoyer une requête *remove-xact* pour lui signifier de réduire ce nombre de 1 à chaque fois que la transaction se termine sur un serveur.

```
/* - VALIDATION GLOBALE DE LA TRANSACTION - */
sprintf(cmdbuf, "COMMIT TRANSACTION");
dbcmd(dbproc-server1, cmdbuf);
if (dbsqlexec(dbproc-server1) != FAIL)
remove-xact(dbproc-commit, commid, 1);
if (dbsqlexec(dbproc-server2) != FAIL)
remove-xact(dbproc-commit, commid, 1);
```

La transaction répartie se termine avec succès, ses effets sont maintenant permanents sur tous les serveurs. L'application clôture le protocole de validation sur le serveur de commit à l'aide de la fonction *close-commit*.

```
/* - FIN DE LA VALIDATION - */
close-commit(dbproc-commit);
dbexit();
```

5 Discussion

Il est clair, suite à l'exposé qui vient d'être fait, que les principes de transparence de la distribution et de la localisation des données ne sont nullement atteints dans l'utilisation du SGBD Sybase dans un contexte réparti. La première différence que l'on peut noter entre le principe général du protocole de validation à deux phase, présenté dans le paragraphe 3, et son implémentation dans Sybase est le fait qu'il n'existe pas d'outil implémentant le protocole dans sa totalité. La Db-library propose plutôt un ensemble de services permettant la programmation d'un tel protocole par l'utilisateur au sein de ses programmes transactionnels. L'application acquiert donc, un rôle dans l'exécution du protocole de validation à deux phases. En somme, elle se doit de jouer elle-même le rôle de coordinateur.

Par ailleurs, les fonctions de la DB-Library ne permettent qu'une implémentation centralisée du protocole de validation à deux phases. Le schéma de communication du protocole ne prévoit une communication entre les serveurs participants et le serveur de commit qu'en cas d'erreur. Dans le cas normal, l'application est la seule à communiquer avec le serveur de commit. Le nombre de messages échangés reste tout de même identique au cas général car tous les messages qui sont envoyés par le serveur coordinateur dans le cas général sont pris en charge par l'application.

En outre, toutes les opérations du protocole de validation sont séquentielles. En effet, il n'existe pas de diffusion de messages qui pourrait diminuer le coût des communications ni de traitement parallèle sur les différents serveurs.

Le protocole de terminaison utilisé n'est pas explicité dans la documentation technique et la DB-Library n'offre pas de fonctions dédiées à la gestion des timeouts. De telles fonctions sont souhaitées pour adapter les protocoles de terminaison de la validation à deux phases au type de transaction exécutée.

Malgré ces quelques insuffisances, il nous semble néanmoins que la stratégie d'implémentation comme celle présentée ici (ensemble de services) est plus réaliste que celle consistant à implémenter “en dur” le protocole de validation à deux phases. La deuxième possibilité aurait contraint l’utilisateur à utiliser un schéma général du protocole quel que soit le type de transaction exécutée. Une implémentation, comme celle de Sybase, pourrait, à notre avis, être augmentée par des services de gestion des timeouts afin d’adapter le protocole de terminaison au type de transaction utilisée et, également, par des services de reprise négociée, c’est à dire, un support automatisé qui permettraient aux administrateurs des bases de données locales de se concerter lors de reprises en cas de panne ou d’annulation des effets de sous-transactions, d’autant que ce type de mécanisme est nécessaire dans les nouvelles applications des bases de données [5, 4].

Références

[1] Y. Belhamissi. Scheduling in an Object-Based System : The Gothic Approach. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems, ECOOP’92*, Utrecht, The Netherlands, June 1992.

[2] P.A. Bernstein and D.M. Chiu. Using semi-joins to solve relational queries. *ACM*, 28(1):25–40, January 1981.

[3] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Trans Comput. Syst.*, 2(1):39–59, February 1984.

[4] N. Boudjlida. Bases de Données Avancées : Réparties, Etendues, Déductives, Orientées Objets. Support de cours, Université de Nancy 1, Octobre 1993.

[5] P. Richard C. Delobel, C. Lécluse. *Des Systèmes Relationnels aux Systèmes à Objets*. Inter-Editions, 1991.

[6] J.N Gray. *Operating Systems : An Advanced Course*, chapter Notes on Data Base Operating Systems In Operating System, pages 393–481. R. Bayer, R.M Graham, and G. Seegmuller, New York, springer verlag edition, 1979.

[7] M.T Ozsu and P. Valduriez. Distributed database systems: Where are we now? *Computer*, pages 68–78, August 1991.

[8] M.T Ozsu and P. Valduriez. *Principles of Distributed Database systems*, chapter 6, pages 148–171. Prentice-Hall International Inc., 1991.

[9] S.Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *ACM SIGMOD Int. Conf. On Management of Data*, pages 128–136, Orlando, June 1982.

[10] SYBASE Technical Publication, Emeryville, CA. *Commands Reference Manual for SYBASE Server for UNIX*, 1992.

[11] SYBASE Technical Publication, Emeryville, CA. *Open Client DB-Library/C Reference Manual*, 1992.

[12] E. Wong and K. Youssefi. Decomposition : A strategy for query processing. *ACM Trans Database Syst.*, 1(3):233–241, September 1976.