データベースシステム　100－12
（１９９４．１０．１４）

分散トランザクション制御における代行コミットメント方式

田坂光伸　丹羽智弓　新田淳
(株) 日立製作所 システム開発研究所

分散トランザクションのコミットメント処理には、アプリケーションプロセスの占有時間や、メッセージ通信回数の増大といった問題がある。本稿では、分散トランザクション制御に、コミットメント処理を専門に”代行”するプロセスを導入する。この代行プロセスは、複数トランザクションのコミット処理を並行して行い、それらのメッセージ通信を一括して処理する。これにより、アプリケーションプロセスの利用効率を向上するとともに、通信オーバヘッドを削減し、分散トランザクション処理のスループット向上を狙う。

Agent Commitment for Distributed Transaction Control

Mitsunobu Tasaka　Chiyumi Niwa　Jun Nitta
Systems Development Laboratory, Hitachi Ltd.

The commitment control for distributed transactions increases time for the transactions to occupy application processes, and overhead to transfer multiple messages. This paper introduces ”agent” that is a specialized process to make the commitment only. The agent process concurrently does commitments of multiple transactions and makes a message transfer for a collection of the transactions. It aims to increase the throughput of distibuted transaction processing by better utilization of application processes and decrease of overhead for communications.

1 はじめに

分散環境において構築されたオンライントランザクション処理(OLTP)システムにおいては、分散トランザクションのコミットメント処理のオーバヘッドが、そのスループットに多大な影響を及ぼす。分散トランザクションのコミット処理には、一般に、アボート仮定2フェーズコミット(2PC)[1]と呼ばれるコミットプロトコルが用いられる。

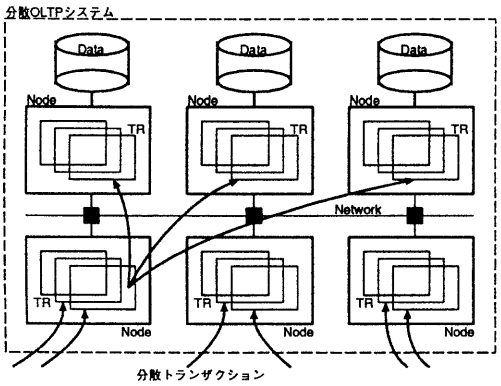

図1 分散OLTPシステム

分散OLTPシステムは、図1に示すように、複数のノードから構成される。分散OLTPシステムには、複数の分散トランザクションが並行して投入される。分散OLTPシステムの内部で、分散トランザクションは、複数のトランザクション(TR)に分割され、複数ノードに分散して処理される。

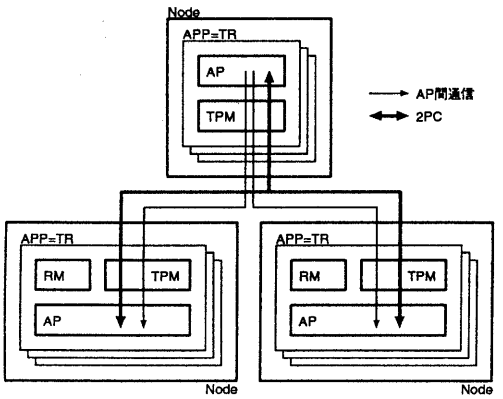

図2 DTP

本稿では、分散OLTPの形態として、X/Openが規定したDTP(Distributed Transaction Processing)[2]を想定する。DTPでは、複数の分散するアプリケーションプログラム(AP)により構成され、個々のAPがデータベース管理システム(DBMS)のようなリソースマネジャ(RM)をアクセスする[2]。ここで、図2のように、個々のAPを処理するプロセスそれぞれがトランザクション(TR)と"結合"し、AP間の通信に従ってTRの親子関係が形成される。トランザクション処理モニタ(TPM)[3]は、AP間通信機能を提供し、分散トランザクション制御を行なう。つまり、前記のようにAP間通信により形成されたTRの親子関係に従って、分散トランザクションのコミット処理を行なう。以降では、APを処理するプロセスのことをAPプロセス(APP)と呼ぶ。

2PCは、分散トランザクションに属するTR間の親子関係に従ってメッセージ通信を行ないながら、コミット処理を進めるものである。図3は、2PCにおけるメッセージ通信の様子を示す。ここで、分散トランザクションの最上位に位置するTRをルートTRと呼ぶ。

多くのTPMが用いるコミット処理方式は、APPが、それが結合するTRのコミット処理をも行なうというものである[4]。しかし、この方式による2PCには、以下のような問題点がある。

課題1 コミット処理中のメッセージ待ちにより、APPがアイドルな状態のまま占有される時間が増大し、分散OLTPシステムのスループットを劣化させる。

課題2 ノード間のメッセージ通信回数が増大し、分散OLTPシステムのスループットを劣化させる。

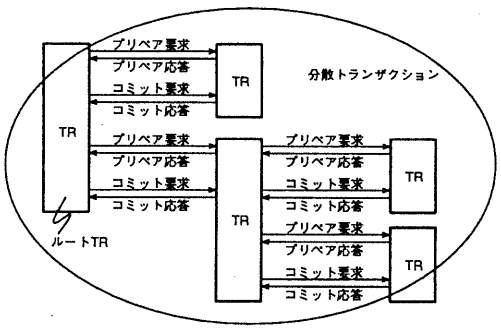

図3 2フェーズコミット

本稿では、TPMが行なう分散トランザクション制御に、以下のような処理方式を導入する。

(1) 各ノードに、TRのコミット処理を専門に行なうプロセス(CMP)を導入する。CMPは、APPに代わって、当該APPが結合していたTRのコミット処理を行なう。

(2) CMPは、複数のTRを並行処理するとともに、それらについてのメッセージ通信などのコミット処理を、一括して行なう。

CMPのようなコミット専用プロセスの概念は、X/OpenのDTPにも存在している[5]。また、商用のTPMの中には、トランザクションログの出力を専門に担当するプロセスを設けているものもある[4]。このプロセスは、複数のAPPにより処理されたTRが出力していたログを一括して出力する。これらに対して、本稿では、CMPによる代行コミットの制御方式を提案し、メッセージ通信などについて、複数TRの一括処理を導入することにより、分散OLTPシステムのスループットの向上を図る。

本稿では、代行コミットの実現方式についての定性的な考察を述べる。以下、2節で、APPからCMPへの代行の制御方式について述べた後、3節で、CMPが行なう複数TRの一括処理について論じる。

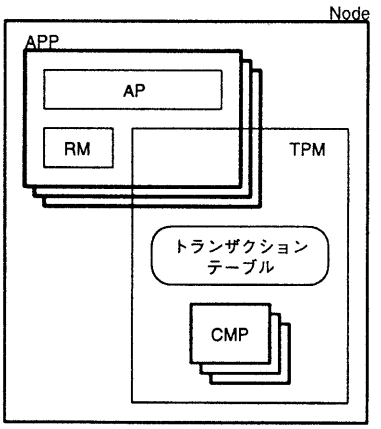

図4　システム構成

2　代行コミット

CMPは、APPの代わりに、当該APPが結合していたTRのコミット処理を行なう。これにより、コミット処理が開始されると、APPを、それまで結合していたTRから解放できるようになる。つまり、APPは、純粋なAP部分の処理を行なうだけで良くなり、それまで結合していたTRがコミット処理に入ると、他のTRと結合して、再びAP部分の処理を行なう。すなわち、ある一定時間の間に、一つのAPPが処理できるTR数を増加させることできる。

図4は、代行コミットを導入したTPMのシステム構成の概略である。個々のノードは、複数のCMPを備える。CMPは、複数のTRを、マルチスレッドにより並行処理するが、一般に、起動できるスレッド数には上限があるため、複数プロセスを設けることにしている。トランザクションテーブルは、当該ノードで動作しているTRを登録しておく管理表であり、共有メモリに置かれる。APPは、自らと結合するTRをトランザクションテーブルに登録する。CMPは、トランザクションテーブルを参照して、コミット処理を担当するTRに関する情報を得る。

以下、代行コミットにおいて、コミット処理を行なうべきTRをCMPへ割当て、APPを解放する処理の方式について述べる。

2.1　代行コミットの契機

2PCにおいて、あるTRのコミット処理の開始時点は、当該TRに対するプリペア要求が、当該TRの存在するノードに送信されて来た時である。そこで、あるTRの代行コミットを、当該TRへのプリペア要求メッセージの受信を契機に開始する。

ここで、同一の分散トランザクションに属する複数のTRが、同一ノードに存在する場合がある。この場合、分散トランザクションのコミット処理は、同時期に行なわれるのであるから、一つのTRに対してプリペア要求が送信されてくれば、他のTRについてもコミット処理を開始して構わないように思える。しかし、問題は、それほど単純ではない。

分散トランザクションのコミット処理は、ルートTRに相当するAPが、コミットを要求することにより始まる。ここで、下位のTRにプリペア要求が伝達されていくが、プリペア要求は、単にコミット処理の契機を示すだけでなく、当該TRの中ではこれ以上、APによる処理を要求されないことを示している。言い替えれば、一般に、プリペア要求が送信されてくるまでは、当該TRの中で、APによる処理が終ったか(コミット処理に移って良いか)を知ることができない。

上記の制約を無視できるかどうかは、AP間で行なわれる通信の形態に依存する。図5は、代行コミット開始の契機とAP間通信の形態の関係について示す。

DTPにおいて用いられるAP間通信には、大きく分けて、Request/Response型とConversational型がある[6]。Request/Responseは、一般に、RPC(Remote Procedure Call)と呼ばれる形態であり、あるAPが、他のAPの持つプロシージャを呼出すものである。Request/Response型には、同期型と非同期型がある。同期型では、呼出し元APは、呼出し先APでのプロシージャが完了し、応答を受けとるまでブロックする。対して、非同期型では、呼出し元APは、応答を待たない。一方、Conversational型では、二つのAPが、send/receiveにより、互いに通信を行なう。本稿では、便宜上、Request/Responseの同期型を同期RPC、Request/Responseの非同期型、及び、Conversational型を非同期RPCと呼ぶことにする。

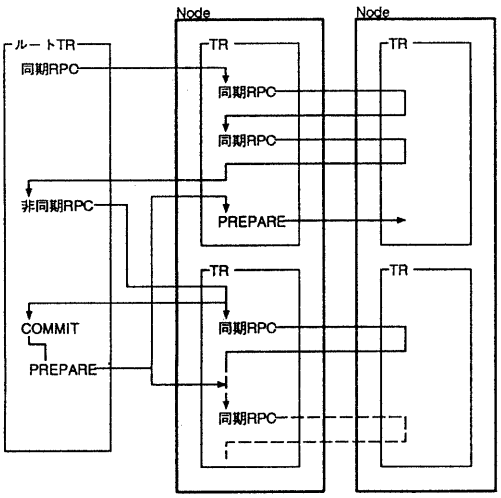

図 5　AP 間通信形態との関係

さて、分散トランザクションに属する全ての TR の中で、全ての AP が同期 RPC のみを用いた場合には、前述の制約を無視できる。何故なら、ルート TR の AP がコミットを要求した時点で、全ての TR において、AP による処理の終了が保証されるからである。つまり、この場合には、ある TR に対するプリペア要求が送信されてくれば、同一の分散トランザクションに属する他の TR についても、コミット処理を開始して構わない。

しかし、分散トランザクションに属する TR の中に、非同期 RPC を用いたものがある場合には、事情が異なる。ルート TR の AP がコミットを要求しても、AP による処理が続行中の TR が存在するかもしれない。つまり、ある TR に対するプリペア要求が送信されてきても、他の TR については、さらに新たな RPC を受けとる可能性があるため、そのコミット処理を開始することができない。

2.2　CMP への割当て

CMP と TR の対応は固定ではない。ある TR のコミット処理は、オンライン中のある一時期に必要になるにすぎず、CMP は、複数の TR のコミット処理を繰り返さねばならないのは明白である。そこで、代行コミットにおいては、コミット処理を行なうべき TR を、CMP へ割当てる処理が必要になる。

CMP に、コミット処理の代行を要求する方式として、図6に示すように、以下の三方式が考えられる。

方式案 1 従来の方式と同様に、APP がプリペア要求を受信し、当該 APP が、CMP へ、自らが結合していた TR の代行コミットを行なうよう要求する。

方式案 2 CMP とは別に、代行コミットの制御プロセスを導入する。制御プロセスがプリペア要求を受信し、CMP へ、当該 TR の代行コミットを行なうよう要求する。

方式案 3 CMP が、プリペア要求を、直接、受信し、当該 TR の代行コミットを行なう。

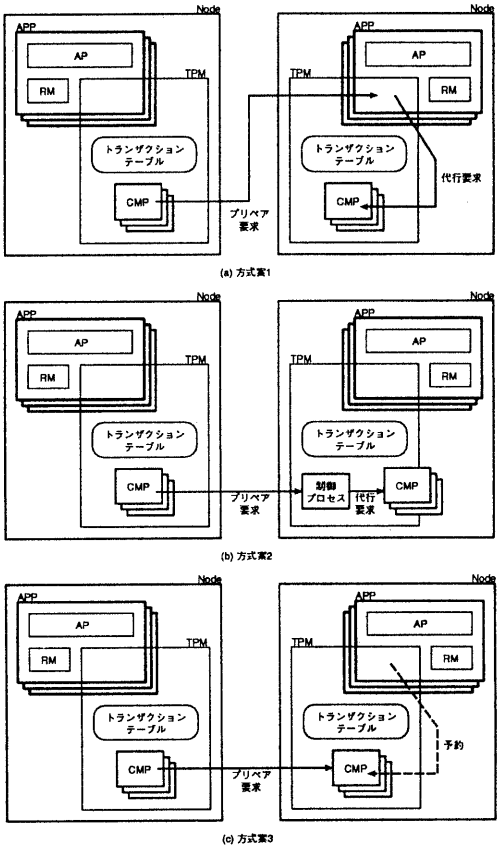

図 6　CMP への割当て方式

先ず、実現性について論じる。上位の TR が、下位の TR へのプリペア要求を送信するとき、その送信先のプロセスが判明していなければならない。これは、案 1 と 2 では、満たされている。案 1 では、上位 TR のノードにおいて下位 TR が結合している APP がわかっているので、当該 APP に送信すればよい。案 2 では、下位 TR のノードの制御プロセスに、常に送信すればよい。しかし、案 3 では、複数の CMP が存在するので、上位ノードでは、いずれの CMP に送信すべきかがわからない。CMP が一つしかなければ問題は解決するが、CMP のスレッド数が不足したとき、コミット処理を中断することになってしまい好ましくない。

一方、性能面を見た時には、案3が優れている。案1と2では、余分なプロセス間通信(コミット処理を行なうべきTR数分)が発生してしまう。対して、案3は、コミット処理を行なうCMPに、直接、プリペア要求を送信できるので、性能面で勝っている。

案3の実現性での問題点は、CMPの"予約"を行なうことにより解決することができる。これは、TRを開始した時に、あらかじめ、当該TRのコミット処理を行なうCMPを決めるものである。決定は、CMPに、スレッド数の余裕があるかによって行ない、スレッド数の不足が発生しないように制御する。

ここで、並行して処理を行なうCMPの数は、できるだけ少ない方が良い。コンテクストスイッチの発生回数を抑制できるからである。これを考慮すれば、CMPの予約は、余裕がある限り、単一のCMPに集中するように行なう方が良いことになる。しかし、これにはペナルティがある。つまり、従来の方式ならば、複数のAPPにより、並行して受信されていたメッセージが、単一のCMPにより受信されることになるため、処理の逐次化が発生してしまうのである。これらのトレードオフについては、現在、評価中である。

2.3 APPの解放

APPが、結合しているTRから解放され、新たなTRを受け入れることができるのは、以下のいずれかの条件が満たされたときである。

(1) 結合しているTRについての、新たなRPCを待たなくても良いこと。

(2) 2.1項でも述べたように、結合しているTRのコミット処理を開始しても良いこと。

(1)は、TPMにおけるRPCの処理方式に依存した条件である。一つのTRにおいては、一般に、上位のTRから、複数回のRPCが要求される。この複数回のRPCの処理方式には、次の二通りが考えられる[5]。一つ目は、個々のRPCが、任意のAPPにより処理されうるとする方式である。そこでは、初回のRPCと次回のRPCが、異なるAPPにより処理されても構わない。これをマイグレート方式と呼ぶ。今一つは、複数回のRPCが、全て、同一のAPPにより処理される方式である。

マイグレート方式では、APPは、それが結合していたTRに属する次のRPCを、自発的に待つことはない。たまたま、前回のRPCと同一TRのRPCを受け入れることになるかもしれないが、単なる偶然にすぎない。いいかえれば、この方式では、APは、複数回のRPCに跨ってコンテクストを保持できない。対して、チェインド方式では、APは、複数回のRPCに跨ってコンテクストを保持することができる。一つのAPPが、一つのTRに結合されたままになるからである。しかし、このために、APPは、それが結合していたTRに属する次のRPCを、自発的に待たなければならない。

以下に、上記の二つの方式それぞれがとられている場合の、代行コミットにおけるAPPの解放について考察する(表1参照)。

表1 APP解放処理の必要性

	マイグレート方式	チェインド方式
同期RPC	不要	要
非同期RPC	要	要

(1) マイグレート方式の場合

APPが受け入れたRPCが同期RPCの場合には、APPの解放のための特別な処理は不要である。APPは、次のRPCを待つためにTRと結合し続けないし、当該TRへのプリペア要求が送信されてきた時、先のRPCの処理は終了しているからである。

しかし、APPが受け入れたRPCが非同期RPCの場合には、APPの解放のための処理が必要になる。これは、プリペア要求が送信されてきた時、当該APPが、RPCの処理中であるかもしれないからである。

(2) チェインド方式の場合

APPが最後に受け入れたRPCが同期RPCでも非同期RPCでも、APPの解放のための処理が必要である。APPは、結合しているTRにおいて、次のRPCを待ち続けるからである。

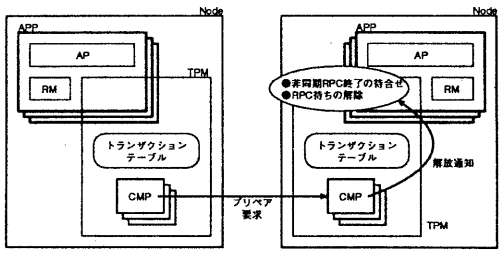

図7 APPの解放

APPの解放のための処理は、CMPが、あるTRへのプリペア要求を受信すると、当該TRと結合しているAPPに解放を通知することにより行なえる(図7参照)。非同期RPCの場合には、この通知処理において、CMPが、APPにおけるRPCの処理の終了を待ち合わせる。ここで、解放処理により、新たにプロセス間のメッセージ通信が発生するのを避けるため、共有メモリを介したポーリングを適用することができる。また、チェインド

方式においては、呼出し元のAPが、呼出し先のTRにおける最後のRPCを、明示的に指定することにより、APPを、"自発的に"解放させて、CMPによるAPPの解放処理を省くことができる。

2.4 ルートTRの扱い

ルートTRと結合するAPPにおいて、APは、TPMに対して、分散トランザクションの開始と終了(コミットやロールバック)を要求する。つまり、当該APは、コミットを要求すると、その応答を受けとらねばならない。従って、ルートTRには、他のTRと同様の代行コミットを適用することはできない。

X/OpenのDTPでも、上記のようなAPが分散トランザクションの開始や終了を要求するための標準インタフェースを規定している[7]。そこでは、APが発行したコミット要求に対して、APに応答が戻る時期についても定められている。それによると、APは、応答時期として、コミット処理の第1フェーズ完了時点か、第2フェーズ完了時点かを指定することができる。

よって、上記の標準インタフェースに準拠したTPMにおいては、ルートTRの代行コミットを、以下のように行なわざるを得ない。

(1) コミット応答時期が第1フェーズ完了時点の場合には、APへ応答後、第2フェーズのみをCMPにより代行する。

(2) コミット応答時期が第2フェーズ完了時点の場合には、このルートTRの代行コミットは行なわない。

3 一括処理の導入

本節では、CMPによる複数TRの一括処理の方式について述べる。一括処理は、通信ネットワークやディスクなどへの出力処理を、複数のTRについて、個々に行なうのではなく、一度の処理により行なってしまうものである。その目的は、スループットの向上にある。一括処理は、個々のTRについて見れば、それぞれのコミット処理の完了時点を遅延させるように働くが、ある一定期間内に処理できるTR数を増加させる効果を持つ。

しかし、その効果が発揮されるのは、個々のノードに、多数のTRが並行して投入されている状況下に限られる。代行コミットは、従来のコミット処理に、CMPという新たなプロセスを介在させることになるため、プロセス間のコンテクストスイッチの発生回数を増加させる。つまり、一括処理により、このデメリットを上回る効果を得れるだけのTR数が存在しないと、かえってスループットの劣化を招く恐れも持っている。

3.1 一括処理

コミット処理の内容は、APのセマンティクスに依存しない。例えば、あるTRに対するプリペア要求を受けとった時にTPMが行なうべき処理は、一般に、下位TRに対するプリペア要求メッセージの送信、RMに対するプリペアの要求、及び、トランザクションログの出力である。TPMは、APの内容が変わっても、それらをTRとして認識することにより、同一のロジックを適用する。もちろん、これらに現れるメッセージや、要求のパラメータ、ログには、それぞれの種類を表すデータや個々のTRを識別するための情報(TR識別子)が付加される。しかし、それぞれのサイズやフォーマットは固定とするのが一般的であろう。つまり、これらの処理は、一括して扱うのに適しているわけである。

冒頭では、2PCによるメッセージ通信回数の増大を、一括処理導入の根拠として取り上げたが、本稿の代行コミットでは、メッセージの一括送信だけでなく、RMに対する要求(関数コール)の一括化や複数のトランザクションログの一括出力も導入を図っている。

(1) メッセージの一括送信

同一ノードへ送信すべき複数のメッセージを、一つのメッセージにパッキングして送信する(図8参照)。複数のTRについてのメッセージの通信回数を、総量として削減できる。

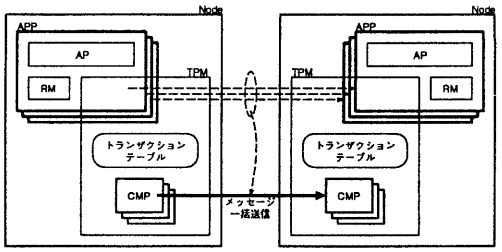

図 8 メッセージの一括送信

(2) トランザクションログの一括出力

複数のトランザクションログのディスク出力を、一回の出力処理によって行なう。複数のTRについてのトランザクションログのディスク出力回数を、総量として削減できる。

(3) RMに対する要求の一括化

コミット処理において、TPMは、RMに対して、ある一つのTRのプリペアや、コミットなどを要求する。X/OpenのDTPでは、これらの要求は、TPMとRMの間のC言語関数インタフェースとして規定されている[5]。つまり、代行コミットにおいても、CMPは、RMに対して、プリペ

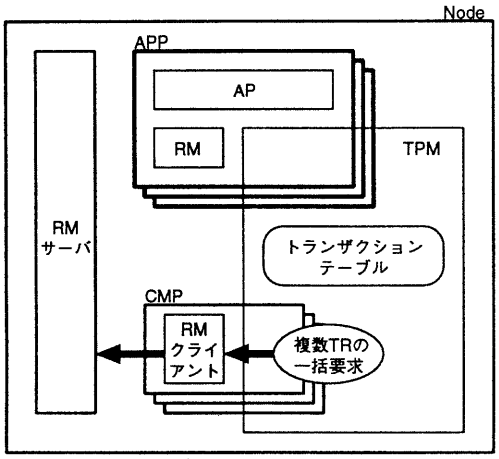

図 9　RM への要求の一括化

アやコミットを要求できるインタフェースを持たねばならない。

代行コミットにおいては、TPM と RM の間に、一回の関数コールにより複数の TR についてのプリペアやコミットを要求するインタフェースを設けることにより、CMP が、複数の TR についての RM への要求を一括化できる。

これは、TPM と RM の間の関数呼び出し回数が削減されただけで、TPM と RM を合わせた処理の総量でいえば、変化がないようにも思える。しかし、RM が、クライアント / サーバ型のシステム構成をとっている場合に効果を発揮する (図 9参照)。つまり、RM クライアントが、複数 TR について一括化された要求を、分割することなく RM サーバに送信することにより、CMP と RM サーバの間のメッセージ通信回数やコンテクストスイッチの回数を削減することができる。

3.2　処理方式

一括処理には、次の二つの方式がありうる。

(1) 分散トランザクションに依存して行なう方式

同一の分散トランザクションに属する複数の TR を一括して処理する。同一の分散トランザクションに属する TR 群のコミット処理は、ルート TR のコミット要求に応じて、並行して同時期に行なわれる。よって、(2) に比べて、個々の TR のコミット処理完了の遅延が小さい。

本方式は、同一ノードに、同一分散トランザクションに属する複数の TR が投入されるような場合でないと、効果を発揮できない。つまり、本方式は、例えば、バッチ的に大量データの処理を行なう分散トランザクションで、さらに、個々のノードにおいて、データに依存して分割された複数 TR が、並行して動作するような場合に適切である。

(2) 分散トランザクションに依存しないで行なう方式

個々の TR が属する分散トランザクションに関わりなく、複数の TR を一括して処理する。オンライン処理において典型的な、大量の TR が、短時間の間に投入されるような場合に適切である。

ただし、この場合には、別々の分散トランザクションのコミット処理は、必ずしも並行して行なわれるわけではないので、(1) に比べて、個々の TR のコミット処理完了の遅延が大きくなる危険性を持っている。しかし、前述したように、それを補うだけの効果が上がるに十分な負荷 (TR 数) がある場合には、有効な方式である。

ここで、一括する TR の集合を、TR グループと呼ぶことにする。以下に、個々の実現方式について考察する。

3.2.1　分散トランザクション依存方式

本方式では、CMP は、当該ノードに存在し、かつ、同一の分散トランザクションに属する TR 群をリストアップし、それらを一括して処理する。2.1項で述べたように、TR グループに属する TR 群のうちで、ある一つの TR へのプリペア要求を受信したからといって、即座に、当該 TR グループの一括コミット処理を開始して良いというわけではない。プリペア要求が送信されてこないと、コミット処理に移れない TR がありうるからである。つまり、このままでは、常に、TR グループの全ての TR にプリペア要求が送信されてくるまで、当該 TR グループの一括コミット処理を開始できない。

この制約は、TR グループに属する個々の TR について、それぞれの祖先の TR に非同期 RPC により生まれたものがあるかを調べることにより緩和することができる。全ての TR の祖先に、非同期 RPC により生まれたものが無い場合には、いずれかの TR へのプリペア要求を受信した時点で、当該グループのコミット処理を開始できる。

一方、TR グループに、非同期 RPC により生まれた祖先を持つ TR がある場合には、その TR のプリペア要求の受信を待たないと、当該グループのコミット処理を開始することができない。この時、特殊な状況下では、コミット処理の途中停止 (デッドロック) を招いてしまうことがある。

図 10は、プリペア要求待ちによるデッドロックの様子を示している。図 10では、TR-B は、TR-A の子 TR、TR-C は、TR-B の子 TR となっている。TR-A は、祖先に非同期 RPC により生まれた TR を持つとする。このとき、Node-A では、TR-A と TR-C のプリペア要求の受信を待ち、それらが揃ってから、TR-A と TR-

Cのコミット処理を開始する。しかし、TR-Cへのプリペア要求は、TR-Bがプリペア要求を受信しない限り送信されてくることはない。つまり、TR-AとTR-Cのコミット処理は、永遠に開始されないことになり、デッドロックとなってしまう。このようなデッドロックは、TRグループに、他ノードを経由した子孫のTRが含まれている場合に発生する。

上記のデッドロックは、分散トランザクションの部分木の形状を表す情報により解決することができる。つまり、TRグループを構成するTRの祖先TRについて、それぞれが存在するノードを示す情報を用いる。例えば、図10であれば、この情報により、TR-Cについて、祖先のTRであるTR-Aが同一ノードのNode-Aに存在することがわかる。これにより、TR-Cへのプリペア要求の受信待ちを、TR-Aとともに行なわないようにすれば、デッドロックを解消できる。

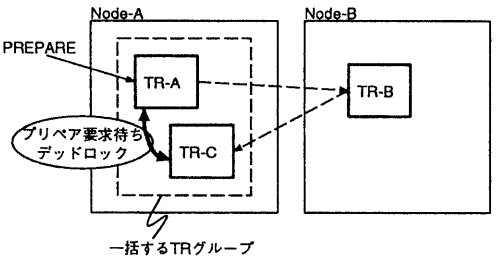

図 10　一括処理におけるデッドロック

3.2.2　分散トランザクション非依存方式

本方式では、CMPは、当該ノードに存在する複数のTRを一括して処理する。一括処理されるTRグループは、異なる分散トランザクションに属するTRから構成されてもよい。図11は、本方式の概要を示す。

本方式では、それぞれのCMPが、コミット処理すべきTRの識別子を、ひとまずためこんでおくバッファ(一括処理バッファ)を備える。CMPは、受信した一括メッセージに含まれるTR群を、このバッファに登録しておく。CMPは、ある一定時間ごとに定期的に、あるいは、一括処理バッファに一定数以上のTRが滞留したら、バッファに登録されているTR群をTRグループとして、一括コミット処理を行なう。ここで、個々のTRについて考えると、ある時点で、それぞれに適用すべき、2PCにおける次の手順(トランザクションログ出力、RMコール、メッセージ送信)が、一般に、異なる。そこで、CMPは、行なうべき手順別にTRグループを構成し、一括処理を行なう。

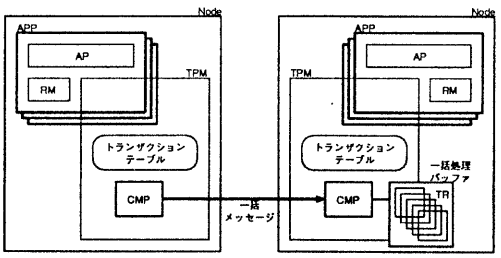

図 11　分散トランザクション非依存方式

4　おわりに

代行コミットを実現する上での、問題点と解決策、及び、方式の定性的評価について述べた。代行コミットは、その方式自体のオーバヘッドを抑制しないと、かえって、分散OLTPシステムのスループット劣化を招きかねないものである。従って、各ノードに投入されているTR数によっては、従来のAPPによるコミット処理方式を選択するなどの、動的な制御を必要とする。今後は、実測による性能評価を行なって、そのような動的選択の基準を明らかにしていく方針である。

参考文献

[1] Mohan,et al.,"Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," ACM Proceedings of the 2nd SIGACT/SIGOPS Symposium on Principles of Distributed Computing, pp.76-88, 1983.

[2] "Distributed Transaction Processing: Reference Model Version 2," X/Open Company Ltd., 1993.

[3] Gray,et al,"Transaction Processing: concepts and techniques," Morgan Kaufmann, 1993.

[4] 渡辺他,"高信頼UNIXシステム," マグロウヒル, 1994.

[5] "Distributed Transaction Processing: The XA Specification," X/Open Company Ltd., 1991.

[6] "Distributed Transaction Processing: The XATMI Specification," X/Open Company Ltd., 1993.

[7] "Distributed Transaction Processing: The TX (Transaction Demarcation) Specification," X/Open Company Ltd., 1992.