計算機アーキテクチャ 92－8
（1992. 1. 24）

階層トランザクション機構によるUNIX上の高信頼分散処理環境

金井達徳　　白木原敏雄
(株)東芝 総合研究所

分散処理の普及にともなって，その高信頼化が求められている．我々はUNIXワークステーション上で，リモート・プロシージャ・コール(RPC)を通信と同期の機構とするクライアント・サーバ型の分散処理アプリケーション・プログラムを，階層トランザクション機構によって高信頼に実行するシステムを開発した．プログラムの実行環境として，階層トランザクションの管理，ログの管理，分散透過なトランザクション・ルーティング，システム管理を行なう専用化した複数のサーバ構成をとる．分散トランザクションの管理に関する煩雑な処理はすべてRPCのスタブ中に隠蔽することにより，容易なプログラミングを可能にした．

NESTED TRANSACTION BASED RELIABLE DISTRIBUTED COMPUTING ENVIRONMENT FOR UNIX WORKSTATIONS

Tatsunori Kanai　　Toshio Shirakihara
Research and Development Center, Toshiba Corporation.
1 Komukai Toshiba-cho, Saiwai-ku, Kawasaki 210, Japan

Today, distributed computing becomes common and many people wish to solve their problems with easy-to-use and cost-effective workstations connected with high-speed networks. However, it is difficult to manipulate distributed data among several computers consistently and provide services without meeting failures. Newly developed nested transaction based computing environment enables programmers of any distributed application programs to use reliable execution mechanism with minimal effort. Our application execution environment comprises four server processes for management of nested transactions, logging, location transparent routing of transactions and management of system configuration. Automatically generated RPC stub encapsulates management of distribution and realizes easy transactional programming.

1 はじめに

マイクロ・プロセッサの高速化は，デスク・トップのワークステーションやその上位に位置するサーバ計算機を高性能でしかも低価格なものにしてきた．その結果，旧来の大型計算機による集中型システムからのダウンサイジングがさけばれ，ワークステーションやサーバ計算機による分散処理が一般的な情報システムの構成方法として定着してきている．しかし分散処理型のシステムでは、障害の発生に対して分散した処理やデータの一貫性を保証し、高い信頼性を実現することが重要な課題となっている。

分散処理環境で信頼性を高めるために，データベース [1] や OLTP(On-Line Transaction Processing) の分野ではトランザクションの概念が用いられる．トランザクションはこれらの分野に限らず一般的に広く適用可能なメカニズムであり，オペレーティング・システムのレベルでトランザクション機構を組み込んだ QuickSilver[2] や，Mach オペレーティング・システム [3] 上に階層トランザクション機構を実装した Camelot[4] のような汎用の高信頼分散処理システムが開発されている．

我々は標準的な UNIX ワークステーション上で，リモート・プロシージャ・コール (RPC) をベースとするクライアント・サーバ型の分散処理プログラムに対して，階層トランザクション機構を用いた高信頼な処理を可能にする環境を研究している．この環境では，RPC ベースの分散処理プログラムが，分散・階層トランザクションによる高信頼化機構を容易に利用できることを目指している．今回，広く使われているワークステーション/UNIX のひとつである Sun Microsystems 社の SPARCStation 1+/SunOS 4.1.1 上に，階層トランザクションによる高信頼なプログラムを開発・実行する環境を構築した．本稿ではその実現法と性能について述べる．

2 トランザクションによる高信頼化

2.1 トランザクション

一般に計算機の実行する処理は，入力されたデータを基に主記憶や 2 次記憶装置上のデータを更新し，何らかの結果を出力する一連の流れを持つ．トランザクション処理はこの一連の流れを，次の 4 つの性質を持つトランザクション [1] として実行する．

1. 処理の単位としてのトランザクションは，完全に実行される (コミット) か全く実行されない (アボート) かのどちらかであるという原子性.
2. トランザクションの実行によって，データは矛盾のない状態から矛盾のない状態に変化するという一貫性.
3. トランザクションがコミットするまで，その実行途中の結果は他のトランザクションに見えないという分離性.
4. コミットしたトランザクションの更新したデータは永久的なものであって，その後の障害等によって失われることはないという永続性.

このようなトランザクション処理を，分散処理環境で実行するのが分散トランザクション処理である．

2.2 階層トランザクションと分散処理

トランザクション処理は，プログラムの決まった区間の処理をトランザクションとして実行する．そのため，プログラムの制御構造が平板になりやすく，汎用的なプログラミング機能として用いるには制限が大きかった．そこで，トランザクションに階層構造を持ち込むことによって，柔軟な制御を実現可能にする階層トランザクション (Nested Transaction)[5] の概念が提案されている．階層トランザクションでは，1 つのトップ・レベル・トランザクションは複数のサブ・トランザクションから構成されると考え，サブ・トランザクション間の分離性や原子性，一貫性を保証する．その結果，サブ・トランザクション間の並列処理を行なった場合に，サブ・トランザクション間の干渉を防止して意味的に安全な並列処理が可能になる．また，トランザクションの一部でなんらかの障害が発生した場合にトランザクション全体をアボートするのではなく，その部分を実行するサブ・トランザクションのみをアボートすることができるので，その部分を他の手段で実行し直すなどの柔軟な制御が可能になる．階層トランザクションの持つこのような性質は，分散処理環境で分散・並列処理の効果を上げると共に，柔軟な対障害機能を実現することを可能にする．

2.3 トランザクションの実装法

分散環境でこのようなトランザクションとしての性質を満たした実行を行なうためには，例えば図 1に示すように磁気ディスクのような不揮発性記憶と揮発性の主記憶の間でデータの配置を制御

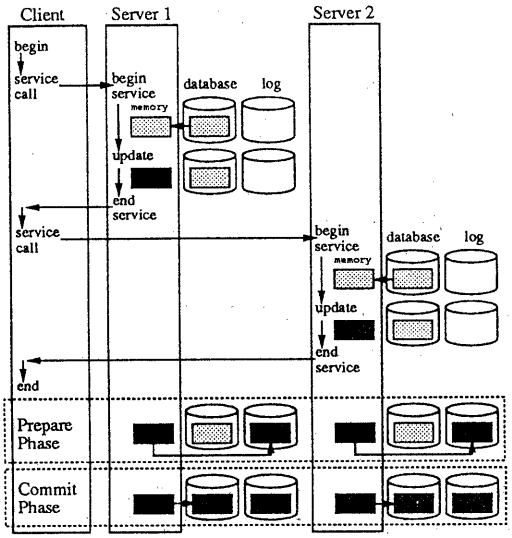


図 1: 分散トランザクションの実現例

しながらプログラムの実行を行なう．ここではクライアント・プロセスが2つのサーバ・プロセスに対してRPCによってサービス処理を依頼する．クライアントとサーバは異なる計算機上に存在しても構わない．各サーバはそれぞれのデータベース中のデータをメモリ上のバッファに持ってきて更新する．その更新結果を全体の処理が終了するまでデータベースには書き戻さないことにより，処理が正常に終了できなくてアボートする場合にメモリ上のデータを無効化して処理の効果を取り消すことができる．全体の処理が終了すれば，次の2相に渡るコミット処理を行なう．

1. **準備相 (prepare phase)**: メモリ上の更新結果をログに記録する．
2. **決定相 (commit phase)**: メモリ上の更新結果をデータベースに書き戻す．

このような2相コミット処理を行なうことにより，分散したデータの同時更新を実現する．ここではコミット処理時にデータベースへの書き戻しを行なったが，コミット処理時以前でも更新内容をログに書き込んだ後であればデータベースへの書き戻しをして構わない．アボート時にはログを基に更新の取消ができる．

分散処理環境においてこのようなトランザクション機構を実装するためには表1のように，

1. トランザクションがどのノード (計算機) のどのサーバ (アプリケーション・プログラム) に拡がって実行されているかという**分散の管理**．
2. トランザクションのコミット/アボートによってどのデータを有効にしてどのデータを取り消すかというリソースの管理．

表 1: トランザクション処理に必要な機能

	アプリケーション・サーバ	トランザクション管理部
分散管理	トランザクションの他サーバ/他ノードへの分散をトランザクション管理部に伝える．	トランザクションがどのサーバ/ノードで実行されているかを管理する．
リソース管理	トランザクション管理部の指示に従って，コミットしたデータを恒久記憶に書き込み，アボートしたデータは取り消す．	トランザクションに関与しているサーバ/ノードにコミット/アボートの指示を出す．

の2つの機能を各アプリケーション・サーバとトランザクション管理部の連係によって実現しなければならない．我々はUNIX上でこれらの機能を実現するために，トランザクション管理部をサーバ・プロセスとして実装し，その機能をアプリケーション・プログラムから呼び出すための言語サポートを開発した．

3 実行環境

今回開発した高信頼分散処理環境は，分散・階層トランザクションの管理を行なう実行環境と，その機能を利用した分散処理プログラムを開発するためのプログラミング環境からなる．実行環境はトランザクション・マネージャ(TM)，ログ・マネージャ(LM)，サービス・マネージャ(SM)，ノード・マネージャ(NM) の4つの管理サーバ・プロセスから構成される．TMとNMは分散環境の各ノードに必ず存在する必要があるが，LMとSMは他ノードのものを利用することも可能である．

3.1 実装環境

実行環境を構成する管理サーバおよびその上でアプリケーション・プログラムを実行するサーバは，いずれもリモート・プロシージャ・コール(RPC)で通信するマルチ・スレッド・サーバとして実現している．RPCはソケット機構を用いたコネクション型の通信上に実装している．同一ノード内の通信はunixドメイン・ソケットを，異なるノード間に跨る通信はinetドメイン・ソケットを用いてい

る．ここで，データグラム型の通信を用いずにコネクション型の通信を用いたのは，通信路の切断などの障害検出が容易に実現できるためである．通信に用いるデータ表現にはSunOSの提供しているXDRを用い，1回はサービス要求を送るが失敗しても何もしないat-most-once方式のRPCを実現している．管理サーバ間の通信，ユーザのアプリケーション・サーバと管理サーバの間の通信はこのRPC機構を用いる．トランザクション処理を行なうアプリケーション・サーバ間の通信は，このRPCを拡張した後述のトランザクション・モードのRPCを用いる．

各サーバ・プロセスにはディスパッチャが存在し，RPCのサービス要求を受けるポートを管理している．サービス要求が来るとディスパッチャはフリー・スレッド・キューからスレッドを取り出してサービス処理を実行させる．このスレッド機構は，SunOSの提供するLWP(Light Weight Process)ライブラリを用いて実現している．

3.2 トランザクション・マネージャ(TM)

TMは分散処理を行なう各ノード上に存在し，そのノードの各サーバが処理を行なっているトランザクションのリストと、そのノードへ他のノードから拡がってきたトランザクションのリスト，および他のノードに拡がったトランザクションのリストを管理している(分散管理機能)．トランザクションの処理が完了すればコミット処理を行ない，何らかの障害で処理を完了できないことがわかればアボート処理を行なう(リソース管理機能)．

階層トランザクションでは，コミットしたサブ・トランザクションの持つロックは親のトランザクションに譲り渡し，また，親の持つロックをその子にあたるサブ・トランザクションが譲り受けることを可能にすることによって，サブ・トランザクション間の並行制御を行なう．このロック所有権の譲渡の可否は，トランザクションの階層構造を完全に知っているTMにしか判断できない．そのため，TMはロックの所有権の譲渡が可能か否かを判断するサービス機能も提供する．

TMの提供するサービスをまとめると，以下のようになる．

server_join/server_leave サーバがトランザクション処理のサービスの開始/終了を伝える．

begin 新しいトランザクションの開始を宣言する．

leave トランザクションが他のノードのサーバに拡がることを伝える．

join トランザクションが他のサーバ/ノードから拡がってきたことを伝える．

commit/abort トランザクションのコミット/アボート処理を依頼する．

lock_transfer サブ・トランザクション間でロック所有権を譲り渡せるか否かを問い合わせる．

3.3 ログ・マネージャ(LM)

LMはディスク装置上のログ・ファイルへのログの書き込み/読み出しを一括して管理する．実行環境を構成する他の管理サーバは，そのログの書き出しにLMを使う．ユーザのアプリケーション・サーバもLMのログ機能を利用できるが，自前のログを用いても構わない．LMを利用するサーバとLMとの間の通信は，ログ・データの受け渡しには共有メモリをスプールとして利用し，ログの強制書き込み等のサービスはRPCで依頼する．

3.4 サービス・マネージャ(SM)

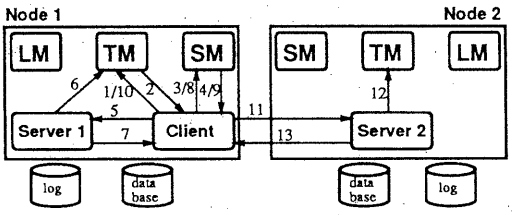
SMは各ノード上の各サーバが提供しているサービスの名前を管理する．サービスを受けたいクライアントがサービス名をSMに伝えると，そのサービスを提供しているサーバの存在するノードのアドレスと通信用ポート番号を返す．SMはトランザクションのルーティングを位置透過にするために用いる．処理の分散や障害対策としてサーバの複製(replication)を作る場合には，SMで処理を振り分けることができる．

3.5 ノード・マネージャ(NM)

NMはそのノードでどのような管理サーバやアプリケーション・サーバが動いているかを管理し，サーバの障害を検出した場合には再起動を行なう．システムの起動時には，各ノード上で動作させるサーバを記述した構成ファイルを読み込んで必要なサーバを起動したり，あるいは終了時に関係するサーバを停止させる機能も持つ．

4 処理方式

ここでは，前章で述べた管理サーバとアプリケーション・プログラムが，どのように協調しながら分散・階層トランザクション処理を行なうかを説明する．



メッセージの意味	プログラミング・サポート
1. トランザクションの開始を宣言	TxBegin()→TM
2. トランザクション識別子を発行	TM→TxBegin()
3. Server 1 の位置を探す	Service Stub→SM
4. Server 1 の位置を教える	SM→Service Stub
5. Server 1 の RPC 呼び出し	Client Stub→Server Stub
6. トランザクションの到着を通知	Server Stub
7. 処理結果を返す	Server Stub→Client Stub
8. Server 2 の位置を探す	Service Stub→SM
9. Server 2 の位置を教える	SM→Service Stub
10. 他ノードへの分散を通知	Client Stub→TM
11. Server 2 の RPC 呼び出し	Client Stub→Server Stub
12. トランザクションの到着を通知	Server Stub
13. 処理結果を返す	Server Stub→Client Stub

図 2: 分散トランザクション処理の流れ

4.1 分散管理方式

図2は，図1の Client と Server 1 がノード 1 に存在し，Server 2 が別のノード 2 に存在する場合の処理の流れを示したものである．Client はまずトランザクションの開始を TM に伝え，そのトランザクションを分散環境内で一意に識別可能なトランザクション識別子を受けとる．その後 Client が他サーバにサービスの実行を依頼する場合には，まず SM に依頼したいサービス内容を伝え，そのサービスを提供するサーバの通信用ポート番号とそのサーバの存在するノードのアドレスを知る。こうして知ったサーバにサービス要求を RPC として伝えると，要求先のサーバは新しいトランザクションが到着したことを TM に伝える．ここで Server 2 へのサービス依頼のように異なるノードにトランザクションが拡がる場合には，RPC を実行する前にどのノードにトランザクションが拡がるかを TM に伝えておく．このように，トランザクションが異なるサーバ/ノードに拡がる時点で TM に知らせることによって，TM はどのトランザクションがどこで実行されているかを把握することができる．

4.2 リソース管理方式

トランザクションの処理が完了すると TM はコミット処理を行なう．図3はコミット処理の流れを示している．Client が処理の終了を TM に伝える

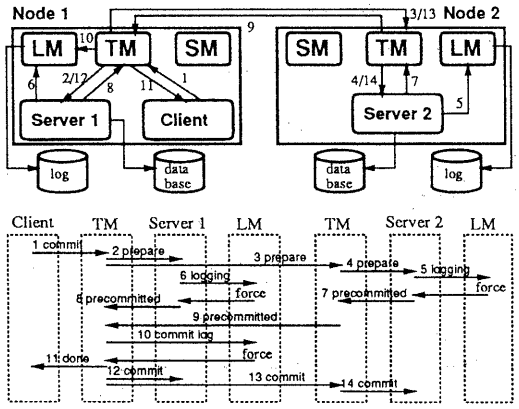


図 3: コミット処理の流れ

と，TM はコミット処理を開始する．まずデータの更新を行なっている Server 1 と Server 2 に対して prepare メッセージを送り，準備相の処理を指示する．各サーバは正常に終了できるならば更新内容をログに書き込んだ後，TM に precommitted メッセージを返す．各サーバから precommitted が返ってくると TM はコミットすることをログに書き込む．書き込みが終了すれば，Client に対してコミット終了を伝えると共に各サーバに commit メッセージを送り，決定相の処理を行なってデータの更新を有効にするよう指示する．

5 プログラミング環境

分散・階層トランザクションによる高信頼処理を行なう分散アプリケーション・プログラムの開発支援環境として，RPC のスタブ・ジェネレータと，プログラム中でトランザクション処理の機能を呼び出すための言語サポートを提供する．これらの支援により，RPC で通信するクライアント・サーバ型のトランザクション処理プログラムを容易に記述することができる．

5.1 スタブ・ジェネレータ

RPC を用いたプログラムの開発を用意にするために，スタブ・ジェネレータを開発した．このスタブ・ジェネレータは，先に述べた通常モードの RPC 用スタブと，トランザクション・モードの RPC 用スタブの両方を生成可能である．図4に，今回開発したスタブ・ジェネレータの入力となる仕様記述の例を示す．`interface` 部に RPC として提供

```
struct request  {
        int     account;
        int     teller;
        int     branch;
        int     delta;
    };
struct reply {
        int     account;
        int     teller;
        int     branch;
        int     delta;
        int     balance;
    };
interface tp inetdomain 2499 {
        reply   deposit(request);
    };
```

図 4: RPC のインタフェース記述例

する関数名とその引数および結果の型を記述する．この例で tp はこのインタフェースの名前であり，inetdomain というキーワードで，inet ドメインのソケットを使った通信であることを示している．各スタブは C++のクラス定義の形で生成される．その結果，他のサーバへのサービス依頼はオブジェクトのメンバ関数の実行と同じ形式で記述できる．

トランザクション・モードのスタブ・ジェネレータは，図4のような仕様記述を基に，3種類のスタブを生成する．図 2に示したような，トランザクションの分散管理に必要となる TM や SM との通信はスタブの中に埋め込まれ，RPC とトランザクションを組み合わせて exactly-once RPC を実現する．各スタブは以下のような機能を持つ．

サーバ・スタブ: サービス要求を受けた後，そのサービスを実行する前に TM に対して新しいトランザクションが到着したことを join で伝える．サービスの実行途中でアボートする場合は，アボート・コードを送り返す機能も持つ．

クライアント・スタブ: サーバにサービスを要求する前に，それが異なるノード上のサーバであれば，TM にトランザクションが他のノードに拡がることを leave で伝える．クライアント・スタブは，通信相手のサーバと 1 対 1 に結合する．

サービス・スタブ: SM に対してサービスを提供するサーバを問い合わせ，そのサーバと結合したクライアント・スタブを呼び出す．クライアント・スタブを直接使う場合には，通信相手のサーバの存在するノードを陽に指定する必要があるが，サービス・スタブを用いると，要求したいサービス名を指定するだけで，その存在するノードを知らなくても通信できる．

5.2 プログラミング言語サポート

プログラム中で分散・階層トランザクション機構を利用するために，以下のような言語プリミティブを提供している．

TxBegin/TxEnd トランザクションの開始と終りを宣言する．TxEnd まで実行されてきたトランザクションは自動的にコミットされる．

TxCommit/TxAbort トランザクションのコミット/アボートを陽に要求する．

TxLockTransfer サブ・トランザクション間でロックの所有権を委譲できるかどうかを TM に問い合わせる．

上記のプリミティブとスタブ・ジェネレータの生成したスタブを用いると，分散・階層トランザクション機構を利用したアプリケーション・プログラムは容易に記述できる．図5と図6は，図4に示したインタフェースを実現するクライアントとサーバのプログラムの例である．クライアント側のプログラムでは，TxBegin でトランザクションを開始した後，tp_service という型のサービス・スタブ tv を用いて deposit サービスを依頼する．その結果が正常であれば TxEnd で自動的にコミットする．

サーバ側の main プログラムは，TxInitialize で初期化すれば，あとはディスパッチャがサービス要求を受けて tp_server 型のサーバ・スタブ ts のメンバ関数として記述した deposit 手続きを呼び出す．

```
tp_service tv;
reply *rpl;
request req;
int s;

TxBegin();
  // prepare argument into req
  rpl = tv.deposit(&req);
  // *rep is a result
TxEnd(s);
```

図 5: クライアント側のプログラム

```
tp_server ts;
main() { TxInitialize(); }

void
tp_server::deposit(request *req) {
  reply rp;
  // process *req & put result into rp
  RETURN(&rp);
}
```

図 6: サーバ側のプログラム

データを自身では管理しないサーバは，このように容易にプログラミングできる．データを管理するサーバのプログラムを記述する場合は，TM がコミットの準備相の処理や決定相の処理を指示した場合に実行するデータ管理用の手続きをプログラマが与えなければならない．そのために，SvrPrepare, SvrCommit, SvrAbort という 3 つの関数名が予約されている．これらの関数の本体をプログラマが与えると，それぞれコミットの準備相，決定相，アボート時に呼び出される．これらの関数の中でどのような処理を行なうかはプログラマの責任であり，トランザクションとしての一貫性や永続性，分離性を保証するようなプログラムを記述しなければならない．トランザクションとして管理しなければならないデータの管理を TM の管理する既存のデータベース管理システムに任せる場合は，このような手続きを与える必要はない．

6 性能

トランザクション処理を行なうと，分散管理やリソース管理のオーバヘッドのために実行に要する時間が増加する．トランザクション処理に要するコストはアプリケーション・プログラムに依存するため，トランザクション処理に必要となる個々の機能に要するコストを基に全体のコストを見積もる必要がある．そこでその基礎となるデータを，SPARCstation1+ (CPU:25MHz, Memory 64MB) を用いて測定した．ログ・マネージャのログ・データは内蔵のディスク装置に記録している．

6.1 リモート・プロシージャ・コール

我々のシステムは RPC を分散処理の基本的な通信機構とし，それをトランザクションの概念を用いて高信頼化するというアプローチを採っている．

表 2: リモート・プロシージャ・コールの性能

mode	0/0	32/32	32/1k
normal mode (local)	3.5	3.7	4.3
normal mode (remote)	3.8	4.1	5.7
transaction mode (local)	10.0	11.4	12.1
transaction mode (remote)	14.8	17.3	19.2

data size: *argument*/*result* (byte)
unit: millisecond

表 2 に通常のモードおよびトランザクション・モード双方の場合の RPC に要する時間を示す．この値は何も処理をしない空の手続きを提供するサーバを，クライアント・スタブを直接使って呼び出した場合に，手続きの引数および値の大きさを変化させて計測している．トランザクション・モードの RPC は通常モードの RPC のコストに対して，サーバに新しいトランザクションが到着したことを TM に登録するためのローカル RPC のコストが加わる．リモートのトランザクション・モード RPC の場合は，さらにトランザクションが他ノードに分散することを TM に登録するための RPC がクライアント側で必要になる．

6.2 トランザクション処理のオーバヘッド

begin を呼び出して TM にトランザクションの開始を宣言する TxBegin と，commit を呼び出して TM にコミット処理を依頼する TxEnd の実行に要する時間を，トップ・レベル・トランザクションとサブ・トランザクションの場合について測定した．測定は図 7 に示すような動作を行なうプログラムを用いて行なった．Client プログラムはトップ・レベルのトランザクションを開始した後，サブ・トランザクションを開始する．このサブ・トランザクションの中で複数の Server にトランザクションを分散させる．その後，サブ・トランザクションをコミットし，さらにトップ・レベル・トランザクションをコミットする．各 Server が準備相でログに書き出すデータは 64 バイトに固定している．

トップ・レベル・トランザクションの開始処理に要する時間 Tbt は約 5.2msec，サブ・トランザクションの開始処理に要する時間 Tbs は約 4.9msec であった．次に，Server と Client がすべて同じノードにある場合と，すべて異なるノードにある場合に，Server 数を 1 から 4 に変化させてコミット処理に要する時間を測定した．その結果を図 8 に示す．

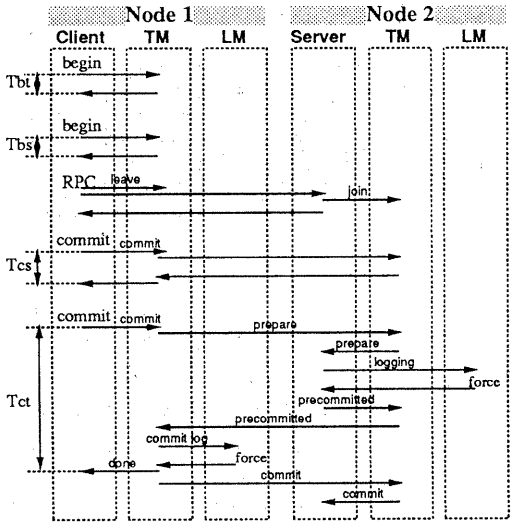


図 7: 測定に用いたプログラムの動作

サーバがすべてローカルの場合は，サーバ数が増えるに従って準備相におけるログの書き込み回数が増加するため，トップ・レベルのコミット処理に時間がかかる．それに対して，リモートの場合はログをそれぞれのノードで書き込むため，サーバ数が増えてもコミット処理に要する時間はほとんど変化しない．同一ノード内のログを一括して書き込むような機構を導入すれば，ローカルの場合でもコミット処理に要する時間の増加を緩やかにすることは可能である．この最適化を施せば、トランザクションを分散させることに必要なコストは、リモート RPC に必要な約 5 ～ 7 msec と、コミット処理時の約 20msec の遅延時間分である。

サブ・トランザクションのコミット処理時には，関係するノードにどのサブ・トランザクションがコミットするかを伝えるため，リモートの場合はローカルの場合より RPC の時間が余計にかかる．この通信は省略する実装法も可能である．図 8からわかるように，サブ・トランザクションに必要なコストはトップ・レベルのトランザクションに比べてはるかに小さい．

7 おわりに

本稿では，我々が UNIX 上に開発した分散・階層トランザクション処理を実現する機構の構成法について述べた．トランザクションを管理するための機構をサーバ・プロセスとして実現すると共に，トランザクション処理に関する機能の多くを RPC のスタブ・ジェネレータに隠蔽することで，容易なプログラミングを可能にした．しかし，リソース管理に関してアプリケーション・プログラム側で行なう準備相や決定相の手続きは，システムの動作に関する知識がないとプログラミングするのは困難である．我々は現在、このプログラミングを容易にするためのプログラミング言語サポートを検討している。

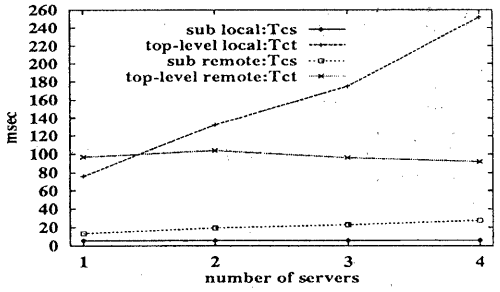


図 8: コミット処理の性能

参考文献

[1] Özsu, M. T. and Valduriez, P.: “Principles of Distributed Database Systems”, Prentice-Hall, Inc. (1991).

[2] Schmuck, F. and Wyllie, J.: “Experience with Transactions in QuickSilver”, Proc. 13th ACM SOSP, pp. 239–253 (Oct. 1991).

[3] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: “Mach: A New Kernel Foundation For UNIX Development”, Proc. USENIX Summer ’86, pp.93–112 (Jun. 1986).

[4] Eppinger, J. L., Mummert, L. B. and Spector, A. Z.:, “CAMELOT AND AVALON — A Distributed Transaction Facility”, Morgan Kaufman Publishers, Inc. (1991).

[5] Moss, J. E.: “Nested Transactions — An Approach to Reliable Distributed Computing”, The MIT Press, (1985).