階層トランザクション機構によるワークステーション上の高信頼分散処理環境†

金井達徳†† 白木原敏雄††

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．本研究では，リモート・プロシージャ・コール（RPC）によって通信するクライアント・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 UNIX 上に開発した．プログラムの実行環境は，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，システム管理を行う専用化した複数のサーバ構成をとる．各サーバにはマルチ・スレッドで RPC 要求を処理できる機構をユーザ・レベルで実装した．トランザクションの分散を管理するのに必要な処理はすべて 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．実行性能を測定した結果，低負荷のアプリケーションであればワークステーション上でも十分な応答時間でトランザクション処理が可能であり，トランザクションに階層を持たせるためのオーバヘッドも少ないことが明らかとなった．

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処理に代って，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．このような分散型のシステムでは，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トランザクションの概念が広く用いられる．トランザクションはデータベース[1]や OLTP (On-Line Transaction Processing) に用いられることが多いが，これらの分野に限らず一般的に適用可能なメカニズムであり，Camelot[2]や QuickSilver[3]のようなトランザクション概念に基づく汎用の高信頼分散処理環境が開発されている．

現在分散処理システムの構築に広く用いられる UNIX オペレーティング・システム[4]は，ファイル管理やプロセス管理がデータベースやトランザクション処理には必ずしも適していないことが指摘されている[5]．そのため高信頼分散処理環境は UNIX の欠点を改善した新しいオペレーティング・システム上に構築されてきた．すなわち Camelot は Mach オペレーティング・システム[6]上のサーバとして階層トランザクション機構[7]を実装しており，また QuickSilver はオペレーティング・システムのレベルでトランザクション機構を組み込んでいる．一方 UNIX 自身も従来不足しているとされていたプロセス間通信やマルチ・スレッド，リアルタイム，バッファ管理などの機能を補う形で進化している．本研究はそのような新しい UNIX の機能[8]を利用することによって，UNIX 上で汎用の高信頼分散処理環境の実現性と性能を明らかにすることを目的としている．

UNIX 上にトランザクション処理を実装するためには，どのような機能をどのような形で UNIX に組み込むか，さらにそれらの機能をどのようなインタフェースでアプリケーション・プログラムに提供するかを明らかにする必要がある．これらの課題を本論文で述べるシステムでは以下のように解決した．

1. トランザクション処理に必要となる機能を整理して4つに分類し，管理サーバとして提供した．
2. 管理サーバやユーザのアプリケーション・プログラムが RPC として呼び出される処理要求をマルチ・スレッドで実行することを可能にする機構を新たに開発した．この機構は UNIX のカーネルを変更することなくユーザ・レベルで実現している．
3. 管理サーバの提供する機能の呼び出しを RPC のスタブ・コードに隠蔽したトランザクション処理指向の RPC を実現し，アプリケーション・プ

† Nested Transaction Based Reliable Distributed Computing Environment for a Network of Workstations by TATSUNORI KANAI and TOSHIO SHIRAKIHARA (Research Lab. II, Communication and Information Systems Research Laboratories, Research and Development Center, Toshiba Corporation).

†† (株)東芝研究開発センター情報・通信システム研究所第二研究所

ログラムの作成を容易にした.

本論文ではまず2章でトランザクション処理に一般的に用いられる手法を概観し，本システムの位置付けを明らかにする．3章ではユーザ・レベルでマルチ・スレッド RPC サーバを実現するための手法と，本システムで採用したサーバ構成を述べる．4章でそれらのサーバがどのように連係することでトランザクション処理が実現されるかを説明する．5章ではトランザクション処理指向の RPC とそれを用いたアプリケーションのプログラミング法を述べる．6章で本システムの性能を示す.

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．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[9]として実行するのがトランザクション処理である.

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）かのいずれかであるという**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では，プログラム中で指定し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版になりやすく，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)[7]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，並列に動作する複数のサブ・トランザクションが同一のデータを更新する場合にはサブ・トランザクション間で並行制御が行われ，矛盾したデータの更新を防止して意味的に安全な実行が可能になる．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，その障害箇所を実行するサブ・トランザクションのみをアボートすることができ，その部分を他の手段で実行する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，柔軟な耐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うためには，一般に2相コミット[10]と呼ばれる手法を用いて，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恒久記憶と揮発性の主記憶の間でデータの配置を制御しながらプログラムの実行を行う．ここではクライアント・アプリケーション・プログラムが2つのサーバ・アプリケーション・プログラムに対して 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磁気デ

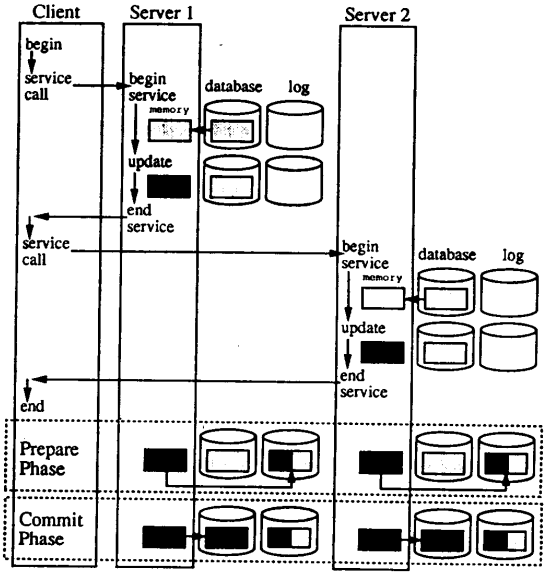


図 1 分散トランザクションの実現法
Fig. 1 An implementation method of distributed transaction.

表 1 トランザクション処理に必要な機能
Table 1 Needed functions for distributed transaction processing.

	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記録する.
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.

ィスク上のデータベース中のデータを主記憶上のバッファにコピーし，主記憶上で更新する．主記憶上の更新結果は，全体の処理が終了した時点で磁気ディスク上のデータベースにコピーして反映させる．処理が正常に終了できずアボートする場合には，主記憶上の更新結果のみ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う．

1. **準備相(prepare phase)**：主記憶上のバッファにある更新内容を磁気ディスク上のログに記録する．
2. **決定相(commit phase)**：主記憶上の更新結果を磁気ディスク上のデータベースに反映させる．

このような2相コミット処理を行うことにより，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主記憶上の更新結果を磁気ディスク上のデータベースへコピーして反映させたが，コミット処理時以前でも更新内容を磁気ディスク上のログに記録した後であれば，磁気ディスク上のデータベースへ反映させても構わない．この場合，アボート時にはログを基にデータベースの更新を取り消す．

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**に示すように，

1. トランザクションがどのノード計算機のどのアプリケーション・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**.

の2つの機能を各アプリケーション・プログラム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．本システムでは 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提供した．

3. 実行環境

本システムでは，分散・階層トランザクション処理に必要な機能を，トランザクション管理 (TM)，サービス管理 (SM)，ログ管理 (LM)，ノード管理 (NM) の4つの機能に分類した．**図2**に示すように，これらの機能を管理サーバ・プロセスとして実装し，実行環境として提供している．アプリケーション・プログラムは必要に応じてこれらの管理サーバの提供するサービスを組み合わせて利用することができる．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 マルチ・スレッド RPC サーバ

実行環境を構成する4つのサーバの提供するサービス機能の呼び出しや，分散アプリケーション間の通信は，リモート・プロシージャ・コール (RPC) によって行う．そのため，管理サーバやアプリケーション・プログラムは同時に複数の RPC サービス要求の処理を進めることのできるマルチ・スレッド RPC サーバとして実装した．

本システムの RPC 機構は，データ表現には Sun-OS の提供している XDR (eXternal Data Representation)[8] を用いた．一方，RPC のプロトコルは本システム独自のものを開発した．RPC はソケット機構[8] をプロセス間通信機構として用いたコネクション型の通信上に実装した．同一ノード内の通信は unix ドメイン・ソケットを，異なるノード間の通信は inet ドメイン・ソケットを用いる．管理サーバ間の通信，ユーザのアプリケーション・プログラム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うアプリケーション・プログラム間の

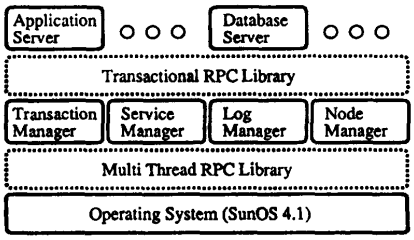


図 2 システム構成
Fig. 2 System configuration.

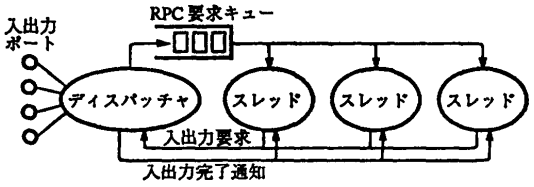


図 3 マルチ・スレッド RPC サーバの構成
Fig. 3 Structure of a multi thread RPC server.

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 RPC を用いる．

複数の RPC サービス要求を同時に処理するために，図3に示すようなマルチ・スレッド機構を開発した[11]．SunOS はユーザ・レベルのスレッドである LWP (Light Weight Process) ライブラリ[8]を提供しているが，本システムではこのライブラリ上に独自のディスパッチャ・スレッドを設けた．ディスパッチャは RPC 要求を受けるポートを含むすべての入出力ポートを管理する．ポートに RPC のサービス要求が到着すると，それを RPC 要求キューに入れる．他のスレッドはこのキューから RPC 要求を順次取り出して処理を行う．処理の途中で入出力を行う場合はディスパッチャに入出力を依頼する．ディスパッチャは入出力が完了するまでは他のスレッドに制御を渡し，入出力が完了すればそれを依頼したスレッドの処理を再開する．このように本システムでは，SunOS の提供するソケット，XDR，LWP といった機能を組み合わせることにより，ユーザ・レベルでマルチ・スレッド RPC サーバを実現している．

3.2 トランザクション・マネージャ (TM)

TM は分散処理を行う各ノード上に存在する．アプリケーション・プログラムは TM の begin サービスを RPC によって呼び出すことで，トランザクションの開始を宣言する．トランザクションが他のノードのアプリケーション・プログラムに分散する時は TM の leave サービスを呼び出し，どのノードに分散するかを伝える．分散したトランザクションが到着したアプリケーション・プログラムは，TM の join サービスを呼び出してどのトランザクションの処理に参加するかを伝える．このようにして TM はそのノードの各サーバが処理を行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，アプリケーション・プログラムは TM の commit サービスを呼び出す．これを受けて TM は，そのトランザクションに関与しているすべてのアプリケーション・プログラムを対象に2相コミットの指示を行う（リソース管理機能）．何らかの障害で処理を完了できない場合，あるいはアプリケーション・プログラムが陽に TM の abort サービスを呼び出した場合は，そのトランザクションに関与しているすべてのアプリケーション・プログラムに対してアボートの指示を行う．2相コミットの各相の処理やアボート処理の指示は，TM がアプリケーション・プログラムに対して RPC 要求を出すことで実現している．

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う[7]．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 lock_transfer サービスを提供する．

3.3 ログ・マネージャ (LM)

LM は磁気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，そのログの記録に LM を用いる．ユーザのアプリケーション・プログラムも LM のログ機能を利用できるが，独自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，スプールしたログを磁気ディスク上に記録したい時は LM に RPC で依頼する．ログ・ファイルは UNIX のファイル・システム上で同期書き込み[8]を用いて実現している．

3.4 サービス・マネージャ (SM)

SM は各ノード上の各アプリケーション・サーバが提供しているサービスのデータベースを管理し，トランザクション処理で必要とされる位置透過なトランザクションのルーティングを実現する．サービスを受けたいクライアント・アプリケーション・プログラムがサービス名を SM に伝えると，そのサービス名をキーとしてデータベースを検索し，それを提供しているサーバ・アプリケーション・プログラム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．この点

は従来から分散処理環境で提供されているネーム・サーバの機能と同等である．それに加えて本システムの SM は，RPC のサービス名だけでなくその引数まで含めたものをキーとしてデータベースを検索することが可能である．この機能を実現するために各アプリケーション・プログラムは，サービス名と引数が与えられた時にそれを自プログラムで実行可能か否かを判定する関数をプログラムとして定義し，SM に伝える．SM は伝えられたプログラムを SunOS のダイナミック・リンク機構[8] を用いてリンクし，データベース検索時の判定に用いる．この機能により，同じサービス要求に対してもその引数によって複数のアプリケーション・プログラムに処理を振り分けることが可能になる．処理の分散や障害対策としてアプリケーション・プログラムの複製 (replication) を作る場合にも，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)

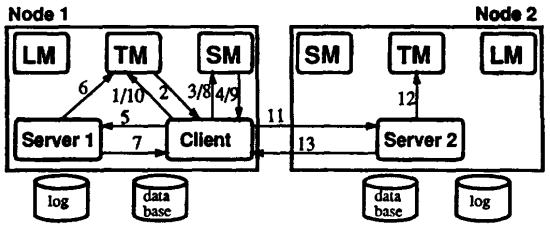
NM はそのノード計算機でどのような管理サーバやサーバ・アプリケーション・プログラムが動いているかを管理する．登録されたサーバの障害を検出すると，NM はそのサーバの再起動を行う．システムの起動時に各ノード上で動作させるサーバを記述した構成ファイルを読み込んで必要なサーバを起動し，終了時に関係するサーバを停止させる機能も持つ．

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うかを述べる．

4.1 分散管理方式

図 4 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す．Client はまずトランザクションの開始を begin サービス要求によって 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 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・アプリケーション・プログラム（以下サーバと呼ぶ）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く到着したトランザクションの処理に参加することを join サービス要求によって 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 leave サービス要求によって伝えておく．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 は各トランザクションを実行しているサーバとそれが存在するノード計算機を把握することができる．



	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 (begin)	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 (lookup)	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 (join)	Server Stub→TM
7	処理結果を返す	Server Stub→Client Stub
8	Server 2 の位置を探す (lookup)	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 (leave)	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 (join)	Server Stub→TM
13	処理結果を返す	Server Stub→Client Stub

図 4 分散トランザクション処理の流れ
Fig. 4 Message flow for distributed transaction processing.

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 2 相コミット処理を行う．**図 5** にコミット処理の流れを示

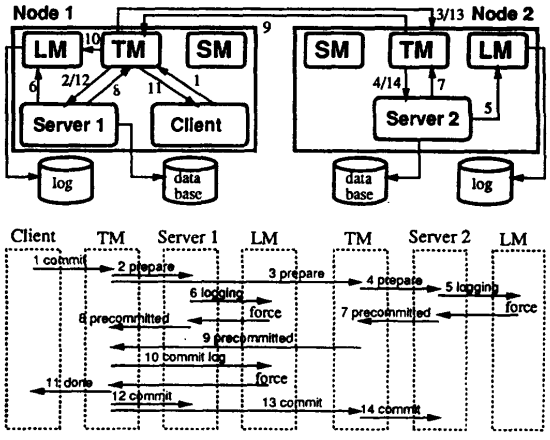


図 5 コミット処理の流れ
Fig. 5 Message flow for commit processing.

す．Client が処理の終了を commit サービス要求によって TM に伝えると TM はコミット処理を開始する．まずデータの更新を行っている Server 1 と Server 2 に対して prepare サービス要求を RPC で送り，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 を返す．すべてのサーバから 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 サービス要求を送り，決定相の処理を行ってデータの更新を有効にするよう指示する．

4.3 リカバリ方式

トランザクション処理におけるリカバリは，トランザクション・リカバリとシステム・リカバリの2つに分けられる．トランザクション・リカバリは，アプリケーション・プログラムが TM に特定のトランザクションのアボートを陽に要求した時，あるいは TM が障害によって処理を継続できないトランザクションを検出した時に起動される．TM はその時点までのトランザクションの処理結果を取り消すため，そのトランザクションに関与しているすべてのアプリケーション・プログラムに対して，そのトランザクション識別子を引数にした abort サービス要求を RPC で送る．abort サービス要求を受けた各アプリケーション・プログラムが，そのプログラム内の該当するトランザクションによる処理結果を取り消すようにプログラミングするのはプログラマの責任である．

システム・リカバリは障害によりダウンしたノード計算機の再立ち上げ時に起動される．このとき NM はログを検索し，障害発生時に実行中であったアプリケーション・プログラムを再起動する．次に TM は各アプリケーション・プログラムに対して，障害発生時に prepare 状態であったトランザクションの識別子を問い合わせる．これらのトランザクションに対して TM は，自ノードのログを見るかあるいはそのトランザクションを開始したノードに問い合わせることによってそのトランザクションをコミットすべきかアボートすべきかを判定する．その結果に基づいて関係するアプリケーション・サーバに commit あるいは abort サービス要求を送り，障害で中断していたコミット処理を完了する．

5. プログラミング環境

本システムでは分散・階層トランザクション処理を行う分散アプリケーション・プログラムを C++ 言語[12]で記述する．さらにトランザクション処理特有のプログラミングを支援するために，RPC のスタブ・ジェネレータとプログラミング言語上の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アプリケーション・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容易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 RPC 用スタブとトランザクション・モード RPC 用スタブの両方を生成可能である．

本システムで開発したスタブ・ジェネレータの入力となる仕様記述の例を**図 6** に示す．仕様記述は，RPC で用いるデータ型の記述と，RPC として呼び出す関数を宣言するインタフェース記述からなる．RPC のデータ表現は SunOS の RPC と同じ XDR を使用しているので，データ型の記述は SunOS の RPC 仕様記述言語に準じる構文を用いている．

インタフェース記述の構文は本システム独自のもので，キーワード interface の後に，

1. インタフェースを識別する名前（図 6 の例では tp）
2. 通信に用いるソケットのドメイン名（同 inet）
3. ソケットのポート番号あるいはソケット名（同 2499）
4. このインタフェースの提供する手続き名とその

```
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 6 RPC の仕様記述例
Fig. 6 Example of an RPC specification.

引数および結果のデータ型の宣言（同 deposit）を記述する．

本システムで開発したスタブ・ジェネレータは，RPC の仕様記述を入力として，C++ のクラス定義の形で各種スタブ・コードを生成する．通常モードの RPC では，スタブ・ジェネレータはサーバ・スタブとクライアント・スタブを生成する．どちらのスタブも仕様記述のインタフェース記述部に宣言された関数をメンバ関数として持つクラスとなる．クライアント・スタブのメンバ関数を呼び出すとそれがソケットを通してサーバ・スタブに伝わり，同名のメンバ関数が実行されることで RPC が実現される．この観点からサーバは分散オブジェクトと見なすこともできる．

トランザクション・モードの RPC に対してスタブ・ジェネレータは3種類のスタブを生成する．サーバ・スタブとクライアント・スタブの役割は通常モードと同じであるが，トランザクションの分散を TM に伝えるために次のような機能をスタブ内に隠蔽している．

サーバ・スタブ：サービス要求を受けると，TM の join サービスを呼び出して新しいトランザクションの到着を伝えた後，そのサービスを実行する．

クライアント・スタブ：サーバにサービスを要求する前に，異なるノード上のサーバへの要求であれば，TM の leave サービスを呼び出してトランザクションが他のノードに拡がることを伝える．

トランザクション・モード特有の第3のスタブはサービス・スタブと呼ぶ．サービス・スタブは，クライアント・スタブの機能とサービス・マネージャ (SM) の提供するネーム・サービス機能を組み合わせたものである．クライアント・スタブはその初期化の時点で通信の相手となるサーバを陽に指定する必要がある．それに対してサービス・スタブは，そのメンバ関数が呼び出されるとその関数名と引数を SM に伝え，そのサービスが受けられるサーバの位置を知る．そのサーバと結合したクライアント・スタブを介してサーバに RPC 要求を送る．サービス・スタブを使うことによって，プログラマは分散したサーバ・アプリケーション・プログラムの位置を意識せずにプログラムを開発することができる．

トランザクション処理に必要な機能を隠蔽した RPC のスタブは Camelot でも用いられている．しかしサービス・スタブのように SM と組み合わせた柔軟で強力なルーティング機能を隠蔽したスタブは本システム特有のものである．

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，以下の言語要素を提供する．

TxBegin/TxEnd：トランザクションの開始と終了を宣言する．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．TxBegin/TxEnd の組を入れ子にすると階層トランザクションになる．

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可能か否かを TM に問い合わせる．

上記の言語要素とスタブ・ジェネレータの生成したトランザクション・モードのスタブを用いると，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7**は図6に示したインタフェースで通信するクライアントとサーバのアプリケーション・プログラムの例である．プログラムはC++で記述するが，この例では説明に必要な部分以外はコメント文中の疑似コードとして示している．クライアント側のプログラムは 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．処理が正常終了すれば 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 で初期化した後，ディスパッチャがサービス要

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

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

（1） クライアント側のプログラム
(1) Client's program.

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

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

（2） サーバ側のプログラム
(2) Server's program.

図7 アプリケーション・プログラムの例
Fig. 7 Example of an application program.

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．

トランザクションとしての一貫性が必要なデータを自分で管理せず，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サーバは，このように容易にプログラミングできる．一方，データを自分で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，アボート時の TM からの要求に答えて，対象となるトランザクション識別子を引数として呼び出される．これらの関数の中でどのような処理を行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．

6. 性　　能

トランザクション処理を行うと，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必要となる個々の機能に要するコストを SPARC station 1+(CPU : 25 MHz, Memory 64 MB)/SunOS 4.1[8] 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

通常モードおよびトランザクション・モード双方の RPC に要する時間を表2に示す．この値は何も処理を行わ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を，手続きの引数および結果の大きさを変化させて計測した．トランザクション・モードの RPC は通常モードの RPC のコストに加えて，サーバに新しいトランザクションが到達したことを TM に登録するためのローカル RPC のコストが必要となる．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．

表 2 リモート・プロシージャ・コールの性能
Table 2 Performance of remote procedure calls.

データ・サイズ：引数／結果（バイト）	0/0	32/32	32/1 k
通常モード（ローカル）	3.5	3.7	4.3
通常モード（リモート）	3.8	4.1	5.7
トランザクション・モード（ローカル）	10.0	11.4	12.1
トランザクション・モード（リモート）	14.8	17.3	19.2

単位：ミリ秒

6.2 コミット処理

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8に示すような動作を行うプログラムを用いて行った．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 5.2 msec，サブ・トランザクションの開始処理に要する時間 Tbs は 4.9 msec 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結果を図9に示す．サーバがすべてローカル

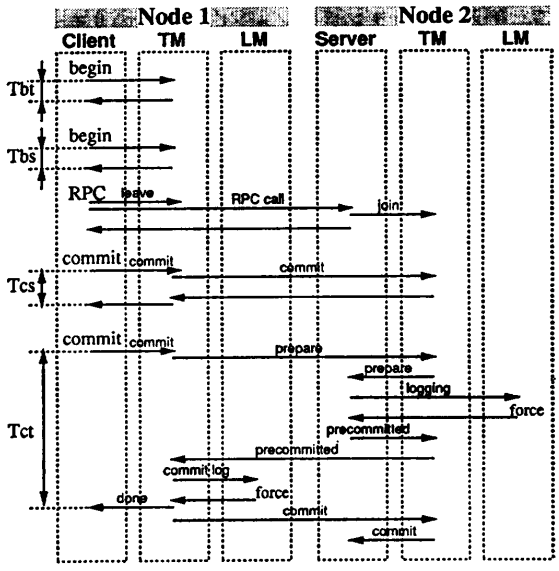


図 8 測定に用いたプログラムの動作
Fig. 8 Behavior of the program to be measured.

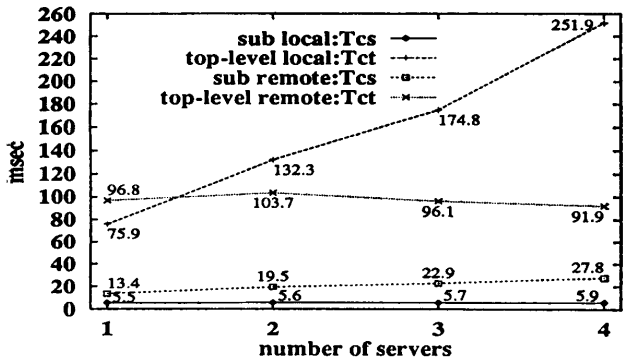


図 9 コミット処理の性能
Fig. 9 The time required for commit processing.

の場合はサーバ数に比例し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．ログは UNIX のファイル・システム上に記録しているが，より高速の記録方式を工夫すればトップ・レベルのコミット処理に要する時間をより小さくすることができる．

6.3 トランザクション処理の性能

トランザクション処理のオーバヘッドは，アプリケーション・プログラムの応答時間の増加として現れる．例えば1トランザクション中で2回のリモート・ノードへの RPC をそれぞれサブ・トランザクションとして行った場合，この遅延時間は 6.1 および 6.2 節で示した測定結果より，

（1） トップ・レベルの開始処理 5.2 msec
（2） サブ・レベルの開始処理 4.9 msec×2
（3） RPC のトラザクション化 13.2 msec×2
（4） サブ・レベルのコミット処理 13.4 msec×2
（5） トップ・レベルのコミット処理 103.7 msec

の合計 171.9 msec となる．

通常，会話型のアプリケーションにおいて許容される応答時間は3秒程度と言われている．例えばトランザクション処理の性能評価に広く用いられる TPC-A ベンチマーク[13]では，測定対象のトランザクションの 90% が2秒以下の応答時間であることが要求されている．この基準と照らし合わせれば，通常の会話型アプリケーションにとって 171.9 msec の遅延時間は十分許容範囲内であるといえる．ただしここでの測定値は他のプログラムが動作していない状態で測定したものであるため，低負荷のトランザクション処理時にはここでの議論があてはまる．しかし RPC の処理は CPU 資源を消費するため，負荷が高くなるとここで示した応答時間が満たされなくなってくると思われる．

サブ・トランザクションを1つ生成することによるオーバヘッドは，その開始処理とコミット処理に必要な時間の和である．図9を参照すれば，ローカルなサブ・トランザクションの場合は約 10 msec，2ノードに分散したトランザクションでは約 18 msec である．この値はトップ・レベルのコミット処理に必要な時間に比べて小さく，サブ・トランザクションを生成することで並列処理を行えば，応答時間を短縮する効果は大きい．

7. おわりに

本システムでは，ユーザ・レベルのマルチ・スレッド RPC 機構をベースに，サーバ構成の高信頼分散処理環境を構築した．実装に用いた SunOS の機能は現在多くの UNIX で提供されており，その意味で移植性の高いシステムとなっている．本システムで採用したサーバ構成は，トランザクション処理に必要となる基本的な機能をサーバとして提供している．Camelot ではリカバリ・マネージャのような高レベルの機能をサーバとして提供している．しかしこのような機能は特定のリカバリ方式をプログラマに強要するという欠点もある．本システムでは，低レベルでもより柔軟性のある機能を提供しておけば，リカバリのような高レベルの機能はそれらを組み合わせて実現できると考えている．

本システムを用いたアプリケーションとして，マルチ・ユーザのスケジュール管理システム[14]を開発している．このシステムでは，異なるノードに存在する複数のユーザのスケジュール・データベースを検索・更新する処理を，それぞれサブ・トランザクションとして並列処理を行っている．また，部分的な処理の取り消しを含むような複雑な条件でのスケジュールの検索・更新処理を，階層トランザクションの持つ記述能力の高さを生かして，サブ・トランザクションを組み合わせて実現している．このような従来繁雑になりがちであった分散トランザクション処理のアプリケーションを，本システムで提供したトランザクション処理

指向の RPC とプログラミング言語サポートによって容易に開発できることが明らかになっている．今後はより多くのアプリケーション・プログラムの開発を通じて，階層トランザクション処理の有効性を実証していく必要がある．

SPARC station/SunOS 上で開発したシステムで計測した性能から，低負荷のトランザクション処理には十分対応できることが明らかとなった．高負荷のトランザクション処理にも対応できるようにするのは今後の課題である．

一方，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．Camelot では Avalon[2] と呼ぶプログラミング言語サポートによってプログラミングを容易にしている．本システムにおいてもこのようなプログラミング言語サポートは必須と考えている．

参考文献

1) Özsu, M. T. and Valduriez, P.: *Principles of Distributed Database Systems*, p. 562, Prentice-Hall (1991).
2) Eppinger, J. L., Mummert, L. B. and Spector, A. Z.: *CAMELOT AND AVALON—A Distributed Transaction Facility*, p. 505, Morgan Kaufmann (1991).
3) Schmuck, F. and Wyllie, J.: Experience with Transactions in QuickSilver, *Proc. 13 th ACM SOSP*, pp. 239-253 (1991).
4) Bach, M. J.: *The Design of the UNIX Operating System*, p. 471, Prentice-Hall (1986).
5) Stonebraker, M.: Operating System Support for Database Management, *Comm. ACM*, Vol. 24, No. 7, pp. 412-418 (1981).
6) 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 (1986).
7) Moss, J. E.: *Nested Transactions—An Approach to Reliable Distributed Computing*, p. 160, The MIT Press (1985).
8) SunOS Reference Manual, Sun Microsystems, Inc. (1990).
9) Haerder, T. and Reuter, A.: Principles of Transaction-Oriented Database Recovery, *Comput. Serv.*, Vol. 15, No. 4, pp. 287-317 (1983).
10) Gray, J.: Notes on Data Base Operating Systems, Bayer, R., Graham, R. M. and Seegmuller, G. (eds.), *Operating Systems—An Advanced Course*, Vol. 60, pp. 393-481, Lecture Notes in Computer Science, Springer-Verlag (1978).
11) 白木原，金井：UNIX 上のトランザクション処理のためのマルチスレッド環境，第 44 回情報処理学会全国大会論文集，1 H-1 (1992).
12) Ellis, M. A. and Straustrup, B.: *The Annotated C++ Reference Manual*, p. 447, Addison-Wesley (1990).
13) Gray, J.: *The Benchmark Handbook for Database and Transaction Processing Systems*, p. 334, Morgan Kaufmann (1991).
14) 関口，関川，真鍋：階層トランザクションを利用したアプリケーションの開発，第 44 回情報処理学会全国大会論文集，1 H-2 (1992).

(平成 4 年 3 月 9 日受付)
(平成 4 年 9 月 10 日採録)

金井 達徳（正会員）

1961 年生．1984 年京都大学工学部情報工学科卒業．1989 年同大学院博士課程修了．1989 年(株)東芝入社．現在，研究開発センター情報・通信システム研究所第二研究所に所属．分散処理のためのオペレーティング・システム，データベースシステム，プログラミング言語などの研究に従事．電子情報通信学会会員．

白木原敏雄（正会員）

1964 年生．1987 年九州大学工学部情報工学科卒業．1989 年同大学大学院修士課程修了．1989 年(株)東芝入社．現在，研究開発センター情報・通信システム研究所第二研究所に所属．分散処理のためのオペレーティング・システムなどの研究に従事．