分散トランザクションシステム *IXI* の設計と実現

國枝和雄[†,*] 各務達人[†,**]
大久保英嗣[††] 津田孝夫[†]

本論文では，分散トランザクションシステム *IXI* の設計と実現について述べる．*IXI* は分散アプリケーションの構築のためのプラットフォームである．すなわち，*IXI* は，分散アプリケーションの開発コストを軽減することを目的としている．分散トランザクションシステムとしては，Camelot/Avalon, Argus, ISIS などがすでに開発されている．しかし，これらのシステムでは，(1)トランザクション処理に関する一貫性が強い一貫性 (robust consistency) に限られる，(2)障害発生時に，システムが回復するまでサービスが停止する，などの問題点がある．そこで，*IXI* では，(1)弱い一貫性 (weak consistency) に基づく入れ子型トランザクションの実行機能，(2)代行プロセス機能とロギング機能，によってこれらの問題点を解決した．これによって，*IXI* では，処理効率，柔軟性，信頼性のいずれの点においても優れたアプリケーション構築プラットフォームを実現することができた．プログラマは *IXI* の提供するC言語ライブラリを用いてクライアントサーバ型のプログラムを記述することによって，分散アプリケーションを容易に構築することができる．本論文では，*IXI* を構成する各部の処理の概要と特徴について説明した後，既存のトランザクションシステムと機能比較することによってその有効性を示す．

A Design and Implementation of the Distributed Transaction System *IXI*

Kazuo Kunieda,[†,*] Tatsuhito Kagami,[†,**] Eiji Okubo[††] and Takao Tsuda[†]

In this paper, a design and implementation of the distributed transaction system *IXI* is described. *IXI* is a platform to construct distributed applications. Namely *IXI* aims at decreasing the development cost for the distributed applications. Some distributed transaction systems such as Camelot/Avalon, Argus, and ISIS have already been developed. However these systems have following problems: (1) The consistency control of transaction processings is limited to the robust consistency. (2) Once the system fails, the services are stopping until its recovery. To cope with these problems, the following functions are embedded into *IXI*: (1) The nested transaction facility based on the concept of weak consistency. (2) The proxy process and logging mechanism. By incorporating these functions into *IXI*, the platform for constructing distributed applications could be implemented, which is good at the performance, flexibility and reliability. It can be easy for programmers to construct the distributed applications, by writing client/server type programs in C libraries provided by *IXI*. In this paper, the outline and features of components which consists of *IXI* are described. Furthermore its characteristics are also clarified, by making a functional comparison with existing transaction systems.

1. はじめに

分散システムにおいて，処理の複雑さをユーザから隠蔽するトランザクションの概念は重要である．しかし，従来のオペレーティングシステム (OS: Operating System) が提供する機能を用いて，トランザクションを実現し，分散システムの利点である並行性や信頼性の向上を十分に活かしたシステムを構築するのは容易ではない．そこで，われわれは分散アプリケーションの開発コストを軽減することを目的とし，そのた

† 京都大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Kyoto University
†† 立命館大学理工学部情報学科
Department of Computer Science, Faculty of Science and Engineering, Ritsumeikan University
* 現在 NEC
Now with NEC Corporation
** 現在 NEC ソフトウェア関西
Now with Kansai NEC Software Corporation

めのプラットフォームとして分散トランザクションシステム *IXI* を実現した．

分散トランザクションシステムとしては，Mach 上に実現された Camelot/Avalon[1]，分散アプリケーション記述言語として開発された Argus[2]，耐障害性に重点を置いた ISIS[3),4)] などがある．これらのシステムでは，トランザクションの実行モデルとして入れ子型トランザクション（nested transaction）を取り扱うことによって，トランザクションの実行効率を高める手法が用いられている．しかし，これらのシステムに共通する問題点として，取り扱うトランザクションの一貫性が強い一貫性（robust consistency）に限られるために，強い一貫性を必要としないアプリケーションの実行が効率良く行えないことが挙げられる．また，トランザクションの並行処理制御にロッキング方式を用いているため，アクセスの競合が増加するとトランザクションの実行効率の低下やデッドロック処理のコストが問題となる．

われわれは，並行処理制御の問題に対する解として，文献 5）において適応型時刻印方式を提案しその有効性を検証した．適応型時刻印方式では，時刻印に基づく複数の並行処理制御を，トランザクションの特性に合わせて適応的に用いることが可能である．さらに，*IXI* は既存トランザクションシステムと同様に入れ子型トランザクションを対象としており，前述した一貫性制御における問題に関する検討が必要である．これに対する解として，本論文において，従来の強い一貫性を緩和した弱い一貫性（weak consistency）に基づくトランザクションの管理手法を提案する．われわれは，入れ子型トランザクションのコミット処理とトランザクションが破棄された場合の処理の 2 点について，従来の強い一貫性を緩和することを検討した．その検討結果に基づき，*IXI* では入れ子型トランザクションの実行において，弱い一貫性の適用方法の異なる 4 つの実行モードを設け，入れ子型トランザクションの効率良い実行を可能とした．さらに，分散システムのもう一つの利点である障害に対する信頼性を向上させるために，一般に用いられるロギング機能に加えて，サーバの非停止性を向上させる代行プロセス機能を実現した．代行プロセス機能は，障害発生によって動作不可能になったサーバの代行サーバを自動的に起動する機能である．一般に，代行処理の実現では複製を生成するためのコストが問題となるが，*IXI* ではカーネルの提供する分散共有メモリ機能を利用し，さらに，代行の対象をサーバプロセスに限定することによって，低コストでこれを実現した．アプリケーション開発者は，*IXI* の提供するトランザクション記述インタフェースを用いることによって，容易にこれらの特徴を持つアプリケーションシステムを構築することができる．以下本論文では，新たに提案するものとして，2 章で弱い一貫性に基づくトランザクション管理方式，3 章で代行プロセスを用いた耐障害機能について述べる．そして，4 章で *IXI* の構成と各部の処理方式について説明し，5 章では既存の分散トランザクションシステムとの機能比較を行い，*IXI* の有効性について議論する．

2. トランザクション管理方式

本章では，*IXI* における弱い一貫性に基づいたトランザクション管理方式について説明する．

2.1 トランザクションの定義

IXI におけるトランザクションとは，Begin と End で囲まれた一連のデータ操作であり，以下のような特性を備える[6]．

（1）一貫性（consistency）
データを一貫した状態から別の一貫した状態へ遷移させる．

（2）原子性（atomicity）
Begin から End までのデータ操作は，完全に実行されるか，全く実行されないかのいずれかである．

（3）持続性（durability）
いったん完了すると，それを破棄することはできない．

トランザクションを完了することをコミット（commit）といい，途中で中断することをアボート（abort）という．トランザクションをアボートする場合，（2）の原子性により，そのトランザクションが更新したデータをすべてトランザクション実行前の状態に戻さなければならない．これをトランザクションの後退復帰（rollback）という．後退復帰を実現するためには，データの更新についての回復（recovery）機能が必要となる．後退復帰されたトランザクションは通常は再実行（restart）されるが，ライブロック（livelock）を防ぐために破棄（cancel）される場合もある．また，複数のトランザクションが並列に実行される環境において，データの一貫性を保つためには，トランザクションのスケジュールが直列化可能（serializable）でなけ

ればならない．すなわち，トランザクションの並列スケジューリングが，ある直列スケジューリングに等価でなければならない．トランザクションの直列化可能性を保証するために，何らかの規則によってデータへのアクセスを制限することを並行処理制御 (concurrency control) という．

2.2 入れ子型トランザクション

トランザクションシステムでは，1つのトランザクション内でさらにトランザクションを起動する入れ子型トランザクションの機能が提供されることが一般的である．入れ子型トランザクションを用いることによって，明示的にサブトランザクションを並行起動し，トランザクションの処理効率を高めることができる．また，トランザクションの実行が失敗した場合に，親トランザクションを中断することなく，関連したトランザクションだけを部分的に中断再実行させることができる．

入れ子型トランザクションでは，トランザクションは呼び出し関係に基づく親子関係として表現される．親トランザクションにおけるサブトランザクションの起動実行には，

```
BEGIN
  COBEGIN
    SUB_TRANSACTION#1
    SUB_TRANSACTION#2
      .....
    SUB_TRANSACTION#n
  COEND
END
```

形式の COBEGIN-COEND ブロックが広く用いられている[7]．*IXI* においても親トランザクションは Cobegin() 要求によって複数の子供すなわちサブトランザクションを一括起動し，Coend() によってサブトランザクションのコミット要求を待つ．すべてのサブトランザクションがコミット要求を発行すると，親トランザクションはその後の処理を継続することができる．サブトランザクションにおけるコミットは，この段階では保留されており，一族すべてのトランザクションがコミット要求を行った時点で一斉に行われる．

2.3 プログラミングモデル

IXI において対象とする分散アプリケーションは，クライアントサーバモデルに基づいて記述されたプログラムである（**図1**参照）．クライアントはアプリケーション利用者の活動に対応し，サーバはデータの管理者の活動に対応する．一般にトランザクションは，データの参照・更新を要求するクライアントとデータを管理するサーバで構成される．クライアント・サーバ間の通信にはクライアントにおける操作を通常の手続き呼び出しと同様に行えるようにした RPC (remote procedure call) が用いられることが多い．トランザクションの操作対象となるデータは，すべてサーバ内で生成・操作・破棄される．クライアントは共有データを直接処理することはなく，RPC を用いてサーバに操作依頼することによって間接的にアクセスする．現在 *IXI* では，サーバにおいて RPC の受け口となる手続きを登録するインタフェースは用意されていない．しかし，クライアントについては，サーバ識別子とサービス名を引数として CallServ() を呼び出すことによって要求を行うことが可能となっており，RPC を用いた場合とほぼ同様の記述性を備えている．クライアンドおよびサーバの記述には，以下で説明するように**表1**に示すインタフェースを用いる．

（1）クライアントにおける処理

クライアントは，図1で示すように Begin() で始

```
server ()
{
 Atomic_Obj Account;

 StartServ("BANK");
 EnableProxy();
 for( ; ; ) {
   switch ( ReceiveServ(from, SUM) ) {
   case DEPOSIT:
     i = Refer(Account);
     if (i - SUM > 0)
       Modify(Account, i - SUM);
     ReturnServ(from, result);
     break;

   case WITHDRAW:

     break;
   }
 }
}
```

```
client ()
{
  Begin();
    SetUserHandler(retry10);
    CallServ("BANK", SUM, DEPOSIT, ret);

    SetSubMode(sub1, MODE1);
    SetSubMode(sub2, MODE3);
    Cobegin(sub1, sub2);

    if ( Coend() != All_Commited) {
      Abort();
    }
  End();
}


retry10 ()
{
 static n = 0;

   if (n++ < 10)  Restart(); else Cancel();
}
```

図 1 トランザクションの例
Fig. 1 Example of transaction.

まり End() で終了する．その間で，CallServ() を呼び出してデータへアクセスする．CallServ() ルーチンの中では，トランザクション管理部やサーバとの通信が行われる．ただし，この通信はユーザからは隠蔽されている．また，Cobegin() と Coend() を用いることで複数のサブトランザクションを起動することもできる．

（2）サーバにおける処理

サーバは，StartServ() で始まり，ReceiveServ() から ReturnServ() の一連の処理を繰り返すループ構造となる．サーバが StartServ() を発行すると，サーバの識別子とロケーションに関する情報がすべてのサイトのサーバ管理テーブルに登録される．ReceiveServ() はクライアントからの要求を受け付けるため，ReturnServ() はクライアントへ処理結果を返すためのインタフェースである．実際には，これらの手続きの中でクライアントや並行処理制御部との通信が行われる．一連の操作の中でデータすなわちオブジェクトを操作するときには，New()，Refer()，Modify()，Destroy() の各インタフェースを用いる．また，ReceiveServ() の前に EnableProxy() を実行することによって，代行プロセス機能を利用することが可能となる（3.1 節参照）．

表 1 トランザクション記述インタフェース
Table 1 Transaction description interface.

クライアント記述インタフェース	
Begin()	トランザクションの開始
End()	トランザクションの終了
CallServ()	サーバに対するデータアクセス要求
Cobegin()	サブトランザクションの起動
Coend()	サブトランザクションの終了待ち
SetSubMode()	サブトランザクションの実行モード設定
Abort()	トランザクションの強制中断
Cancel()	トランザクションの強制破棄
SetUserHandler()	アボートハンドラの登録
Cancel()	トランザクションの破棄
Restart()	トランザクションの再実行
サーバ記述インタフェース	
StartServ()	サーバの登録
ReceiveServ()	クライアントからの要求待ち
ReturnServ()	クライアントへの結果返信
New()	オブジェクトの生成
Refer()	オブジェクトの参照
Modify()	オブジェクトの更新
Destroy()	オブジェクトの破棄
EnableProxy()	代行プロセス機能の有効化

2.4 弱い一貫性

データベースの分野で求められる強い一貫性（robust consistency）を保証するためには，すべてのサブトランザクションは，親トランザクションのコミット時に一斉にコミットされる．したがって，サブトランザクションの実質的な処理が終了しているにもかかわらず，コミットが保留されて，一族の他のトランザクションの終了を長時間待つような状況が生じる可能性がある．その間，サブトランザクションにおいて操作したデータは，並行処理制御によって他のトランザクションから操作不可能な状態に保たれるため，システム全体で考えた場合，トランザクション実行効率を低下させる結果となる．多重版方式や楽観方式の並行処理制御を用いた場合には，コミット保留中にも，他のトランザクションからの同一データへのアクセスが許される場合があると考えられるが，その場合においても最新データの参照を行ったトランザクションは参照操作時あるいは終了時のいずれかの段階で保留される．

また，強い一貫性のもとでは，柔軟なトランザクション記述を困難とする場合も考えられる．問題となるのは，サブトランザクションの実行において，おのおののサブトランザクションを単に並行実行するのではなく，意味的に関連を持たせて実行するような場合である．例えば，ある目的を達成するための手段が複数存在し，それぞれが別のトランザクションとして用意されている場合を考える．この場合，それらのサブトランザクションを並行実行し，少なくともその中の1つが完了すれば目的は達成される．しかし，強い一貫性のもとでは，サブトランザクションが破棄した場合には，その影響によって親トランザクションが中断されるのが一般的である．すなわち，上記のような場合，完了できないサブトランザクションが1つでも存在すると親トランザクションが中断され，意図した処理を行うことができない．

以上の点を鑑みて，われわれはサブトランザクションのコミットと破棄の2つの処理において従来の強い一貫性を緩和した．この一部の制限が緩和された一貫性のことを**弱い一貫性**（weak consistency）と呼ぶ．

（1）コミット処理における一貫性の緩和

強い一貫性のもとでは，サブトランザクションのコミット要求は親トランザクションの終了時まで保留され，親のコミット処理と同時に処理される．このコミットの保留を行うことなく，親トランザクションに先

行してサブトランザクションがコミットを完了できるように一貫性を緩和する．強い一貫性におけるコミットの保留は，親トランザクションが何らかの要因で中断した場合に，一族すべてのサブトランザクションの実行を白紙に戻すことを保証するためのものである．したがって，これを緩和し先行コミットを行うためには，データベースに矛盾を生じないための条件を検討する必要がある．われわれは，その条件として2.5節で述べる補償可能性に着目した．データが補償可能な性質を持つ場合，そのデータに対する操作は，後から実行される別のトランザクションによって取り消すことが許される．つまり，操作を行ってからそれを取り消すまでの状態を，たとえ他のトランザクションが参照したとしても構わない．強い一貫性の立場にたつと，これはデータに矛盾のある状態と考えられるが，実際のデータを考えた場合，この状態が発生したとしても意味的に問題ない場合があると考えられる．すなわち，ここではデータがそのような性質を持つ場合を補償可能であるという．

以上のように，*IXI* ではサブトランザクションが補償可能な性質を備えるとき，そのサブトランザクションのコミット処理を保留することなく，親トランザクションに先行して行うことができ，これによって，当該データをいち早く解放することが可能となっている．

（2）破棄における一貫性の緩和

強い一貫性のもとでは，サブトランザクションが破棄された場合，その影響によって親トランザクションは中断され，さらにその影響は他のトランザクションにも及ぶ．*IXI* では，サブトランザクションが破棄された場合に，その影響によって親トランザクションが自動的に中断されるのではなく，サブトランザクションの状態をいったん親トランザクションに知らせた上で，親トランザクションがその後の処理を決定できるよう一貫性を緩和する．強い一貫性のもとで，サブトランザクションの破棄によって自動的に親トランザクションが中断されるのは，そのトランザクション一族の実行には，親子を含めてすべてのトランザクションの処理が必要不可欠であるとの前提に基づいている．したがって，これを緩和することは，サブトランザクション間の意味的な関係づけと，それに基づくサブトランザクションの制御を，親トランザクションに委ねることになる．*IXI* では，そのために必要となるインタフェースとして，親トランザクションが Coend() の返り値としてサブトランザクションの終了状態を知る機能と，子孫に当たるトランザクションを強制的に破棄するためのシステム関数 Cancel() を提供している．ここで Cancel() は，例えば，サブトランザクションの実行において，3つを並行実行して2つがコミット待ち状態になったが，その中の1つだけ完了させたいような場合に，不要なトランザクションを強制的に破棄するために用いる．このように，サブトランザクションの管理をトランザクション設計者に委ねることによって，柔軟なトランザクションの設計を可能としている．

ここで，破棄の一貫性を緩和したことによる並行処理制御への影響を考察する必要があるが，*IXI* の並行処理制御においては，2.8 節で述べるように，親も子も含めてすべてのトランザクションが独立した時刻印を持つトランザクションとして処理されるため，たとえ破棄における一貫性を緩和したとしても並行処理制御において矛盾を生じることはない．

2.5 補償可能性

トランザクションおよびデータの補償可能性について説明する．

IXI におけるデータはバージョン化されており，更新を行うトランザクションの完了によって新しいバージョンが生成される．

データ D のバージョン i を D_i とすると，ある時点での D はバージョンの集合 $\{D_1, D_2, \cdots, D_i\}$ として表される．また，D_i を作成した D に対する更新操作を $f_i(D)$ とすると，

$$\{D_1, D_2, \cdots, D_{i-1}\} \xrightarrow{f_i(D)} \{D_1, D_2, \cdots, D_{i-1}, D_i\}$$

である．ここで，$\{D_1, \cdots, D_{i-1}, D_i, D_{i+1}, \cdots, D_{k-1}, D_k, D_{k+1}\}$ において，操作 $f_i'(D)$ を挿入することによって，$f_i(D)$ が打ち消されるとき，すなわち $\{D_1, \cdots, D_{i-1}, D_i, D_{i+1}, \cdots, D_{k-1}, D_k, D_{k+1}\}$ と $\{D_1, \cdots, D_{i-1}, D_{i+1}, \cdots, D_{k-1}, D_{k+1}\}$ が意味的に等価であるといえるとき，$f_i(D)$ を**補償可能操作**と呼ぶ．また，$f_i(D)$ を打ち消す作用を持つ $f_k(D)$ を $f_i(D)$ に対する**補償操作** $(=f_i'(D))$ と呼ぶ．さらに，補償可能操作によってのみ操作されるデータ D を補償可能データ，D を格納しているオブジェクトを**補償可能オブジェクト**と呼ぶ．

次にトランザクションの補償可能性について考える．更新操作を f，任意の参照操作の集合を R とするとトランザクション T は，f と R の系列，

$$T=\langle R_0, f_1, R_1, f_2, R_2, \cdots, R_{n-1}, f_n, R_n\rangle$$

と表される．ここで，すべての f_i $(1 \leq i \leq n)$ が補償可能であれば，T に対する補償トランザクション T' を定義することができ，

$$T' = \langle f'_n, f'_{n-1}, \cdots, f'_2, f'_1 \rangle$$

となる．このような T を**補償可能トランザクション**と呼ぶ．

データが補償可能となる典型的なものとして，データに対する操作が，現在値からの相対変化を意味するものに限られている場合が挙げられる．カウンタとそれに対するインクリメントおよびデクリメント操作はその一例である．ある状況では，データ管理者にとってカウンタに対するこれらの操作の実行順序は意味を持たず，操作が実行された回数のみが意味を持つ場合がある．例えばチケット予約システムを考えると，チケット販売者にとっては，誰がどういう順番でチケットを購入したかではなく，何枚売れたかということのみが重要である．その場合，インクリメントおよびデクリメント操作は，互いに補償可能操作であり（チケット予約でいえば，いつ解約しても構わない），そのような操作のみを行うトランザクションは補償可能トランザクションであると言える．

2.6 トランザクションの実行モード

IXI ではサブトランザクションの実行において，前述したコミットおよび破棄における制限の緩和を適用することによって以下に示す4種類の実行モードを設けている．

（1）通常モード

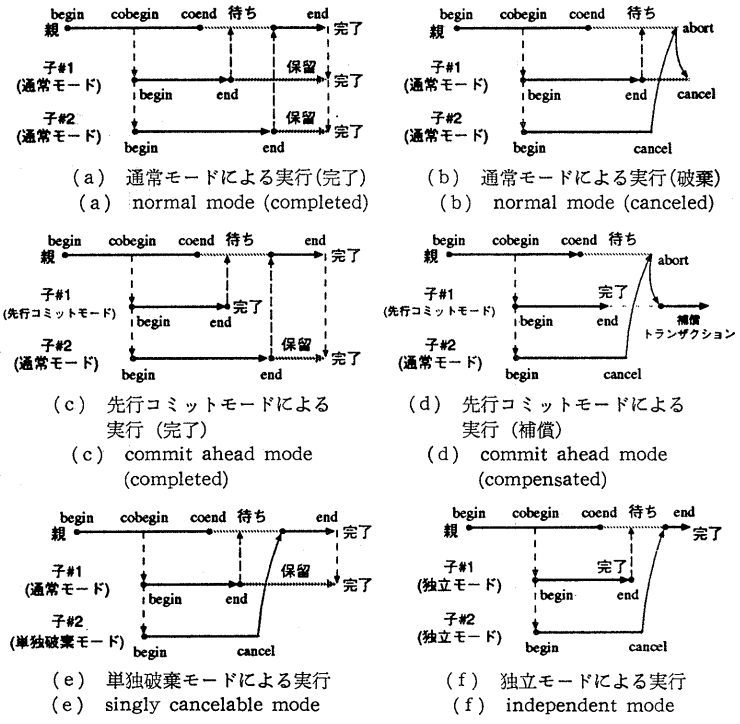
強い一貫性を必要とするトランザクションの実行に用いるモードである．したがって，このモードのトランザクションは前述した入れ子型トランザクションの一般的な手順でコミット処理が行われる．サブトランザクションが破棄（cancel）された場合，同時に起動されたサブトランザクションも自動的に破棄され，親トランザクションのアボートハンドラが起動される．

（2）先行コミットモード

サブトランザクションにおけるコミット処理における制限を緩和したモードであり，サブトランザクションは，親トランザクションの終了を待つことなく先行して完了することができる．このモードで実行するためには，当該サブトランザクションの実行を打ち消す作用を持つ補償トランザクションが定義可能であることが条件となる．先行コミットモードのサブトランザクションが完了した後，何らかの原因で親がアボートした場合には，システムが自動的に補償トランザクションを実行

表 2 各実行モードの相違点
Table 2 Differences in each mode of execution.

	通 常	先行コミット	単独破棄	独 立
自分が破棄された場合	親が中断	親が中断	影響なし	影響なし
コミット終了後に親が中断した場合	破 棄	補 償	破 棄	補 償
コミットのタイミング	親の完了待ち	独立先行	親の完了待ち	独立先行



（a） 通常モードによる実行（完了）
（a） normal mode (completed)

（b） 通常モードによる実行（破棄）
（b） normal mode (canceled)

（c） 先行コミットモードによる実行（完了）
（c） commit ahead mode (completed)

（d） 先行コミットモードによる実行（補償）
（d） commit ahead mode (compensated)

（e） 単独破棄モードによる実行
（e） singly cancelable mode

（f） 独立モードによる実行
（f） independent mode

図 2 各実行モードにおける処理
Fig. 2 Processings in each mode of execution.

することによってその実行を打ち消す.

(3) 単独破棄モード

サブトランザクションが破棄されても親トランザクションがアボートされないモードである. すなわち, 前述の破棄における制限を緩和したモードである. このモードでは, 親トランザクションは Coend() の返り値としてサブトランザクションの終了状態を調べ, 任意の処理を行うことが可能であり, 終了状態に応じた柔軟な記述を行うことができる.

(4) 独立モード

コミットおよび破棄における制限を両者とも緩和したモードである. したがって, このモードで実行されるトランザクションは, コミットおよび破棄のいずれの状態で終了しても他のトランザクションに影響を与えることはない. 利用可能条件および利点ともに, 先行コミットおよび単独破棄の両モードの内容を持つ.

各モードにおける相違点を**表 2** に, 処理の概略を**図 2** に示す.

2.7 サブトランザクション呼出し

すでに述べたように *IXI* におけるサブトランザクション呼出しは, Cobegin() と Coend() によって実現される. ここで, サブトランザクションの終了状態として Coend() 時に親に通知されるものは, 以下の4状態である.

(1) コミット待ち状態

通常モードのトランザクションおよび破棄可能モードのトランザクションが正常に終了し, 親トランザクションがコミットされるのを待っている状態である.

(2) 先行コミット状態

先行コミットおよび独立モードのトランザクションが正常に終了し, 親に先行してコミットした状態.

(3) 破棄状態

通常モードおよび先行コミットモードのトランザクションが破棄された状態.

(4) 破棄可能状態

破棄可能モードおよび独立モードのトランザクションが破棄された状態.

トランザクション設計者は, これらの返り値を利用して適切な処理を記述することができる. 例えば, **図 3** は, 同じ目的を別の手段で果たすためのサブトランザクションが複数存在するときに, それらを並行実行し1つでも正常に終了すればよい場合のトランザクション記述例である. ここでは, 親トランザクションはサブトランザクションをすべて破棄可能モードで実行し, すべての終了状態が破棄可能状態になった場合には Cobegin() によってサブトランザクションを再度実行し, また, 2つ以上がコミット待ち状態となった場合には不要なサブトランザクションを Cancel() によって強制的に破棄することによって, 最終的に1つのサブトランザクションのみが完了するように記述している.

2.8 入れ子型トランザクションの並行処理制御

入れ子型トランザクションにおけるサブトランザクション間の並行処理制御については2相ロックに基づく方式が広く用いられている[7]. この場合, トランザクションはロックを hold および retain する. サブトランザクションがコミットされると, そのロックは親によって継承され, 親はロックを retain する (ロックの上方継承). データにアクセスするトランザクションは, ロックを hold していなければならない. retain しているだけでは, アクセスすることはできない. すなわち, ロックの retain は, 「retainer の子孫はロックできるが一族以外のトランザクションはロックできない」ことを示す目印にすぎない. このようにして, トッ

```
client ()
{
  Begin();
    ・・・・・・・・・・・・
  SetSubMode(sub1, 破棄可能モード);
  SetSubMode(sub2, 破棄可能モード);
  SetSubMode(sub3, 破棄可能モード);

  for( ; ; ) {
      Cobegin(sub1, sub2, sub3);
        ・・・・・・・・・・・
      ret = Coend();
      if (ret == sub1,sub2,sub3のすべてが破棄可能状態) {
            continute;
      } else {
            if (ret == 2つ以上がコミット待ち状態) {
                  Cancel(不要なトランザクション);
            }
            break;
      }
  }
  End();
}
```

図 3 弱い一貫性を用いたサブトランザクションの実行例
Fig. 3 Example of sub-transaction execution with weak consistency.

プレベルのトランザクションがトランザクション内のすべてのロックを retain したときトランザクションはコミットされる．この技法に柔軟性を持たせ，下方継承を可能としたアルゴリズムも提案されている[8),9)]．

一方，*IXI* では時刻印方式に基づくアプローチを採用している[5)]．時刻印方式ではトランザクションの発行時刻を基準にトランザクションの論理的な実行順序を決定することによって，トランザクション間の一貫性を保証する．したがって，サブトランザクションは自分自身の時刻印に加えて親トランザクションの時刻印を継承しなければならない．

トランザクション T が入れ子レベル k で実行されているとき，T を含むレベル i のトランザクションの開始時刻印を $TS_i(T)$ とする．すなわち，T を含むトップレベルトランザクションの開始時刻印は $TS_1(T)$，T 自身の時刻印は $TS_k(T)$ である．ここで，二つのトランザクション，T_1（入れ子レベル m）と T_2（入れ子レベル n）の時刻印を比較する場合，i が1から $\min(m, n)$ までについて，$TS_i(T_1)$ と $TS_i(T_2)$ を順次比較し，最初に $TS_i(T_1) \neq TS_i(T_2)$ となった時点の大小をもって，T_1 と T_2 の時刻印の大小関係とする．*IXI* では，これに基づいて入れ子型トランザクションに関する時刻印処理を行っている．**図4**に示す例において，サブトランザクションCおよびDが競合している場合には，時刻印1と2を比較し，サブトランザクションDとEが競合している場合には，時刻印4と5を比較する．

3. 耐障害機能

障害に対して頑強なシステムを構築するには，障害が発生しても可能な限り処理を継続する機能と，障害の発生したサブシステムを矛盾のない状態に復旧する回復機能を備える必要がある．*IXI* では，システムの完全復旧までの間に，動作可能なサブシステムで代行処理を行う代行プロセス（proxy process）機能と，障害発生サブシステムにおける回復処理を可能とするロギング機能によって高い耐障害性を達成している．

3.1 代行プロセス機能

IXI における代行プロセス機能とは，あるプロセスの代行プロセス（真のプロセスとデータを共有しているプロセス）を他のサイト上で動作させ，障害発生時に直ちに代行処理をさせる機能である．一般に，このようなプロセスの多重化では真プロセスと代行プロセス間でのデータの共有，入出力の共有，処理コンテキストの共有などの機能が必要となり，あらゆるプロセスを対象として代行プロセス機能を実現するためには実行時のコストが問題となる．

そこで，*IXI* では代行処理の対象を2.3節で述べたサーバプロセスに限定し，対象となるプロセスにおける処理に制限を加えることによって低コストで代行プロセス機能を実現した．制限とは，

（1）メッセージ通信以外の入出力は行わない．データへのアクセスも，オブジェクト管理部へのメッセージ通信で行われる．これによって，入出力の共有機能は不要となる．

（2）プロセスが**図5**で示される制御構造を持ち，ノードAからノードXに相当する部分で，他のプロセスとの処理のコンテキストに依存する状態を持たない．ただし，図5において，ノードAは各種初期化ルーチン，ノードXは要求待ち，ノード a, b…m, n は各要求に応じたサービス処理に対応するものと考える．

である．以下，代行機能実現に必要となるコンテキストとデータの共有と機能について検討する．

（1）コンテキストの共有

一般にコンテキストの共有を実現するためには，プロセスの状態を示す各種情報や，場合によってはカーネル内部の状態情報を含めて情報を共有する必要があ

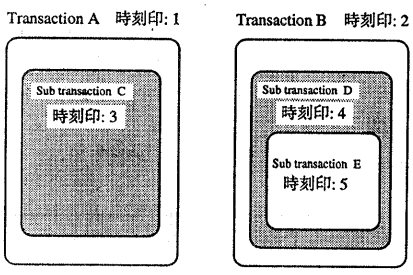


図4 時刻印の継承
Fig. 4 Time-stamp inheritance.

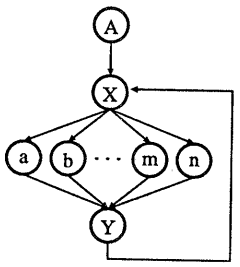


図5 サーバプロセスの制御構造
Fig. 5 Control structure of a server process.

る．どの程度の状態情報を必要とするかは，代行処理を行う粒度に依存する．より細かい粒度での代行処理を実現するためには，より多くの情報を共有しなければならず，粒度と処理量とのトレードオフとなる．*IXI* はトランザクションシステムであり，トランザクションやトランザクションからの要求といった粒度での代行処理が適当である．そこで，われわれは後者，すなわち2.3節で述べたクライアントにおける処理のNew(), Refer(), Modify(), Destroy() を単位とした代行処理を行うこととした．図5において説明すると，サービスaを処理中に障害が発生しサーバプロセスの実行が不可能となった場合には，当該サービスaは処理不能として破棄されるが，それ以後は代行サーバが処理を代行することによって，aからnのすべてのサービスを処理することができる．次に，この代行機能を実現するための共有情報について考える．われわれは前に図5のノードXの実行時点では状態を持たないことを制限事項(2)として挙げた．したがって，ノードXまでは真プロセスと代行プロセスを別々に実行しても等しい状態にあり，この段階では特に状態情報の共有は必要ない．一方，*IXI* の代行処理はサービスの要求を単位としており，代行処理は必ずノードXから開始される．したがって，*IXI* では真プロセスと代行プロセスとの間での状態情報の共有は必要ない．

(2) データの共有

IXI における真プロセスと代行プロセス間でのデータ共有は，$R^2/V3$ カーネルの提供する分散共有メモリ機能[10]を利用することによって実現される．分散共有メモリでは，複数のサイトに存在する分散共有メモリの複製間での内容の一致を保証する整合性 (coherence) 制御が必要となる．その方式としては，共有メモリへの書き込みが発生する度に内容を反映する write-through 方式と，書き込み時には他の複製の無効化のみを行い，無効化された複製に対する参照が行われたときに始めて内容を反映する write-invalidate 方式が一般的である．write-through 方式は，常に共有メモリ間の内容が一致することが保証されるので高い耐障害性が得られるが，共有メモリへの書き込みが頻繁に行われる場合には，通信コストが大きくなるため実用的ではない．汎用に設計した $R^2/V3$ では，この通信コストの問題を無視できないため，整合性制御として write-invalidate 方式を用いている．そこで，これを基本としてサービス要求単位での確実な代行処理を可能とするために，明示的に共有メモリ間での内容を一致させる sync 操作を $R^2/V3$ カーネルの機能として新たに実現した．*IXI* において代行プロセス機能を利用するサーバは，1つのサービスを完了する度に sync 操作を行う．これによって，サービス単位で共有メモリ間の整合性が保証される．

(3) 代行処理の概要

障害発生時の代行処理の概要について述べる．サイト A でサーバ S，サイト B でその代行サーバ S' が実行中であり，その他に S のデータを利用するクライアント C があるとする．S でアクセスするオブジェクトおよび S で用いるデータ領域は，S が EnableProxy() を発行することによって，自動的に分散共有メモリ上に割り付けられ，S' と共有状態になる．この状態でサイト A に障害が発生し S が実行不能になった場合を考える．まず，C もサイト A 上で実行されている場合には，C も同時に実行不可能になるので，サイト A の回復時に回復処理を行う (**図6**(a))．C が A 以外のサイトで実行されている場合には，以下で述べる代行プロセス機能によって可能な限り継続して処理が行われる．C の要求が S に行われていない，すなわち S が要求待ち状態であったならば，sync 操作によって S と S' のデータの整合性が保証されているので，トランザクション管理部がそれ以後の S に対する要求を S' に対して行うように変更すればよい (図6(b))．C の要求を処理中であった場合には，まず，処理中の要求を失敗したとみなして C をアボートする (通常は後退復帰後，再実行)．このとき，S' のデータは C の要求前の状態となっているので，後は同様に，S に対する要求を S' に対して行うように変更することによって代行処理が行われる (図6(c))．

以上で述べたように *IXI* では，対象をサーバプロセスに限定することによって，小さい実行時コストで代行プロセス機能を実現している．また，もう1つの特徴として，サーバ設計者にこれを利用するための負担をほとんど与えないことにある．サーバ設計者が代行プロセス機能を利用するために必要となるのは，真プロセスと代行プロセスの間で共有しなければならないデータを分散共有メモリ上に割り当てるために EnableProxy() を発行すること，サービス終了時に sync 操作を行うこと，および実行時にサーバを複数サイトで起動することの3点である．

3.2 オブジェクト管理

IXI におけるオブジェクトは，データを格納するために用いられる受動的な存在である．オブジェクトは

バージョン化されており，内部には一定期間に生成された複数バージョンのデータが蓄えられる．オブジェクトへのアクセスは，全オブジェクトに共通に用意されているメソッドを用いて行われる．メソッドとしては，オブジェクトの生成，新バージョンの作成，指定バージョンの参照，バージョンの確定，バージョンの破棄，オブジェクトの削減の各処理を行う手続きが用意されている．

オブジェクトは，二次記憶と主記憶によって構成されるオブジェクト空間上で管理されており，要求に応じて主記憶と二次記憶間での転送を行う．ただし，ここで述べる主記憶とは OS の仮想記憶機能によって提供される記憶空間である．トランザクションがデータを参照するとき，オブジェクトに対してはバージョン参照要求が行われ，必要なバージョンのみが二次記憶から主記憶上へ読み込まれる．また，トランザクションがデータを更新する場合には，新バージョンの作成要求が行われ，新たな領域を主記憶上に確保する．その後，トランザクションの2相コミット処理の第1相において，まずバージョン未確定のまま二次記憶に書き込まれる．そして，第2相でコミットが成立するとバージョンが確定し，新たに作成されたバージョンが最新バージョンとなる．主記憶上でもはや利用されなくなったオブジェクトは削除される．

サーバ設計者が代行プロセス機能を利用する場合，オブジェクトは通常の仮想記憶空間ではなく分散共有記憶上に読み込まれる．

3.3 ロギング機能

回復機能を実現するための一般的な手法として書き込み先行ログ方式 (write ahead log)[11] がある．この方式は，データの更新操作に先立って，その更新を取り消す undo ログと再度更新を行うための redo ログを記録しておくものである．*IXI* では，書き込み先行ログとオブジェクトのバージョン管理を組み合わせることによって回復機能を実現している．ログは，トランザクションの存在するサイトの安定記憶装置 (stable storage) に格納される．

IXI では以下の情報をログとして記録することでトランザクションの回復処理を行っている．

（1）トランザクション識別子
（2）トランザクションの開始時刻印
（3）トランザクションの実体が格納されたファイル名
（4）トランザクションの状態フラグ

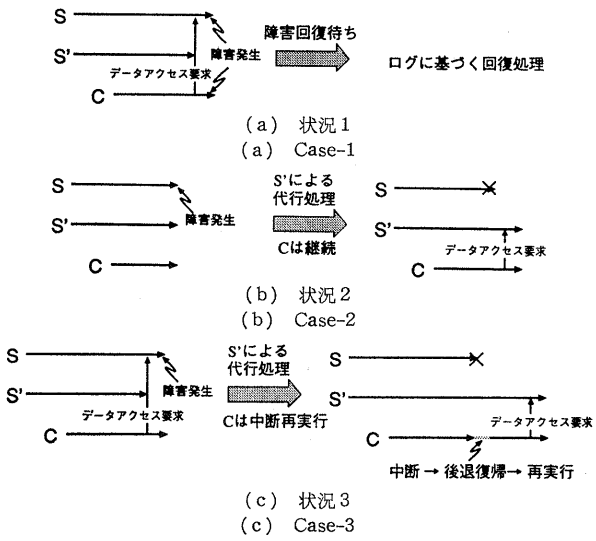


図 6 代行サーバの処理概要
Fig. 6 Outline of proxy server processing.

ここで，状態フラグにはトランザクションのコミット処理に関連して begun, prepared, comitted, aborted, completed の各フラグが設定される．*IXI* では2相コミット方式でコミット処理を行っており，以下の手順でログが記録される．

（1）トランザクションが Begin() を発行する [トランザクション識別子，時刻印，ファイル名，begun フラグを記録する]．
（2）トランザクションの処理を行う．
（3）トランザクションが End() によってコミット要求を発行する [prepared フラグを記録する]．
（4）トランザクションのサイトのコミット調停プロセス M は，他のサイトの調停プロセス S_i にコミットの準備を促す prepare メッセージを送る．
（5）問い合わせを受けたサイトでは，当該トランザクションが作成したオブジェクトを未確定バージョンとして二次記憶に反映させておく．その後，各 S_i は応答を返す．
（6）応答がすべて肯定であれば M はコミット指示を S_i に送る [commited フラグを記録する]．そうでなければアボート指示を S_i に送るとともにトランザクションをアボートする[aborted

フラグを記録する].

(7) 各 S_i は，受けた指示に従って処理を行い，正しく終了すれば完了通知を M に返す．コミット指示を受けた場合には，未確定バージョンのオブジェクトを確定する．

(8) M は完了通知を受けとる [completed フラグを記録する].

3.4 障害発生時の処理

IXI の回復制御部にはシステム監視機能があり，それによって他サイトの動作状態を監視している．障害が発生した場合，システムは複数のサブシステムに分断され，各サブシステムはその状況に応じて以下で示す4状態の中のいずれかになる．

(1) 停止状態

1つ以上のサイトによるサブシステムが，サイト内部の障害によって，動作不可能になった状態である．

(2) 孤立状態

システムは動作しているが，ネットワークの障害で自分以外のサイトと切り離された状態である．この場合，当該サブシステムは1つのサイトで構成される．

(3) 分断状態

ネットワーク障害で自分以外のサイトの中の2つ以上と切り離された状態である．この場合，分断された相手側サブシステムも同じ状態になる．

(4) 残存状態

システム内の自分以外の1つのサイトが切り離された状態である．切り離されたサイトは，停止状態あるいは孤立状態にある．

孤立，分断，残存の各状態となったサブシステムは，障害発生後もサブシステム内での処理は行うことができる．また，接続不可能になったサーバの代行サーバが登録されていれば，障害発生と同時に処理を開始してサブシステム内での要求に応じる．この状態のとき，サブシステム外のオブジェクトへのアクセス要求を行ったトランザクションはアボートされる．そして，アボートハンドラによって再実行が指定された場合には，回復後に再実行するために回復待ち状態となる．それ以外のトランザクションは処理を行った後，コミット要求を行う．障害発生前にサブシステム外のデータにアクセスしたトランザクションと，発生後に代行サーバを用いて処理を行ったトランザクションは，そのままコミットすることはできない．したがって，そのようなトランザクションは，prepared フラグを記録した後，回復待ち状態になる．残りのトランザクションは，通常のコミット処理を行う．

3.5 回復処理

停止状態以外のサイトではシステム監視機能によってシステムの復旧を検知すると，回復可能であることを他サイトに通知する．

停止状態のサイトではシステム管理者からの指示によって回復処理を開始する．そして，次の手順で処理を行う．

(1) サイト内のログを調べ実行中であったトランザクションの回復処理を行う．

(a) begun 状態のトランザクションはアボートし，再実行のために回復待ち状態にする．

(b) prepared 状態のトランザクションについては，prepare メッセージを送る時点からコミット処理を再実行する．

(c) commited 状態のトランザクションがあれば，コミット調停者は再度コミット指示を行い，コミット処理を継続する．

(2) 自サイトのサーバのための代行サーバが有効になっているか調べる．有効になっていた場合にはそれを無効にした後，真のサーバを起動する．サーバ起動時に再度 EnableProxy() を発行すると，分散共有メモリ領域がサーバの記憶空間にマッピングされ，自動的に代行サーバのデータが反映される．

(3) 回復可能であることを他サイトに通知する．

こうして，すべてのサイトが回復可能になると，有効な代行サーバが残っているか否か調べる．これは，複数のサブシステムが分断状態になっていたり，分断状態と孤立状態になっていると，真のサーバが動作しているにもかかわらず代行サーバが動作していることが考えられるからである．その場合には，代行サーバを無効化し，代行サーバにデータ更新要求を行ったトランザクション（回復待ち状態になっている）をアボートする．最後に，すべてのサイトの回復待ち状態にあるトランザクションを実行可能状態にして通常の処理に戻る．

4. *IXI* の概要

4.1 システム構成

IXI は，以下で説明するモジュール群によって構成

される（**図7**参照）．

（1）OS カーネル

IXI の機能を完全に利用可能とするためには，OS カーネルの最低限の機能として，メモリ管理，タスク管理，ネットワーク通信，分散共有メモリの機能が必要となる．この条件を満たすものとして，Mach カーネルとわれわれが独自に開発した $R^2/V3$ カーネル[10]がある．Mach カーネルについては，ユーザが利用可能な外部ページャ機能を用いて分散共有メモリを実現することによって，*IXI* のカーネルとして利用することができる．$R^2/V3$ カーネルでは，タスクをグループ化し，グループ内のタスク間でメモリ領域を共有可能とする分散共有メモリ機能を実現している．*IXI* では，この $R^2/V3$ の分散共有メモリ機能を用いている．

（2）トランザクション管理部

各トランザクションに関する情報を保持し，並行処理制御部との協調作業によって，入れ子型トランザクションの起動，中断，後退復帰，完了の各処理を制御する．

（3）並行処理制御部

トランザクションの時刻印とデータのバージョンを管理し，適応型時刻印方式によって並行処理制御を行う．サーバにおいて行われる回復可能オブジェクトへのアクセスは，まず並行処理制御部においてその正当性が判定された後，回復制御部を通して処理される．アクセスの結果，矛盾を生じた場合には，要求元トランザクションの中断要求をトランザクション管理部に要求する．*IXI* では，文献 5）において提案した適応型時刻印方式を用いている．適応型時刻印方式は，多重版時刻印方式[12]に基づく複数の方式の集まりから構成されている．トランザクションの構成時に，それらの方式の中からトランザクションの特性に応じた方式を選択することによって，時刻印方式の問題点であるトランザクションの後退復帰の発生を低減させることができる．また，トランザクションの実行時において，システムの負荷の状況やそれまでの実行状況（後退復帰の有無など）に応じて動的に方式を選択することも可能となっている．この動的な選択を用いることで，トランザクションが無用に再実行を繰り返すライブロックを防ぐことができる．すなわち，データ更新の失敗によって後退復帰を繰り返しているトランザクションについては，更新の成功率の高い方式へ変更することによって，その終了確率を高めることができる．

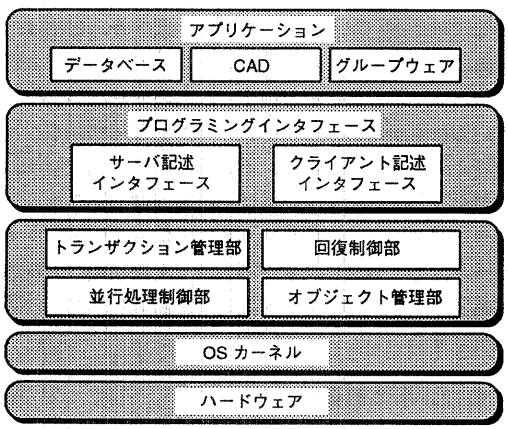


図 7 *IXI* のシステム構成
Fig. 7 System configuration of *IXI*.

（4）オブジェクト管理部

IXI におけるデータは，オブジェクト管理部が管理するオブジェクトに格納される．オブジェクト管理部は，主記憶と二次記憶によって構成されるオブジェクト空間の管理を行う．また，並行処理制御と回復処理に必要となるオブジェクトの版管理も行う．

（5）回復制御部

回復制御部は，書き込み先行ログによってシステムのログを記録し，オブジェクトを障害から回復可能とする．また，システムの動作状態を監視し，障害が発生するとその情報をシステムの各部に知らせるシステム監視機能も持つ．

（6）トランザクション記述インタフェース

IXI におけるトランザクション記述インタフェースは，C 言語のライブラリとしてユーザに提供される（表1参照）．本インタフェースは，クライアント記述インタフェースと，サーバ記述インタフェースに大別される．クライアント記述インタフェースは，さらに，トランザクションの実行に関するもの，サブトランザクションの実行に関するもの，サーバに対して要求を行うためのものに分けられる．サーバ記述インタフェースには，トランザクション管理部に対してサーバを登録するためのもの，クライアントとの通信を行うためのもの，およびオブジェクトアクセスのためのインタフェースがある．

4.2 処理の概要

本節ではシステムの処理概要について述べる（**図8**参照）．

（1）トランザクションの開始処理

クライアントが Begin() を発行すると，トランザ

クション開始メッセージがトランザクション管理部(TM)に送られる．TMはトランザクション識別子，開始時刻印を割り当て，内部テーブルにトランザクションを登録する．

（2）サーバへの操作要求

トランザクションが行ったサーバへの操作要求は以下の手順で処理される．

（a）クライアントがCallServ()によって操作要求を行うと，サービス名を引数としてサーバ情報に関する問い合わせがトランザクション管理部に対して行われる．

（b）TMはサーバ管理テーブルを参照して情報を返す．

（c）その情報を基にサーバへリクエストメッセージを送信する．

（d）サーバは受けとった要求にしたがって処理を行う．そこで，データオブジェクトへのアクセスが必要となれば，オブジェクトアクセスのためのルーチンを呼び出す．すると，アクセス要求メッセージが並行処理制御部に送られる．

（e）並行処理制御部では，まず，時刻印に基づいてアクセスが正当か否か判定する．不当と判断した場合には，サーバへエラー値を返すと同時に，アボート要求を要求元サイトの TM に送る．正当と判断した場合には，要求をオブジェクト管理部に送る．

（f）オブジェクト管理部は，トランザクションの時刻印を基に適切なバージョンのデータを選択し，主記憶上へ割り当てる．そして，主記憶上の先頭アドレスを並行処理制御部経由でサーバへ返す．

（g）サーバは残りの処理を行い，ReturnServ() によって処理結果をクライアントへ返す．

（3）トランザクションの終了処理

クライアントが End() を発行すると，トランザクション終了メッセージがTMに対して送られる．TMは，トランザクションが入れ子のトップレベルで実行されているか調べ，トップレベルで実行されていれば，並行処理制御部にコミット処理を依頼する．トップレベルでなければ，トランザクションの実行モードに基づいた処理を行う（2.4節および2.6節参照）．

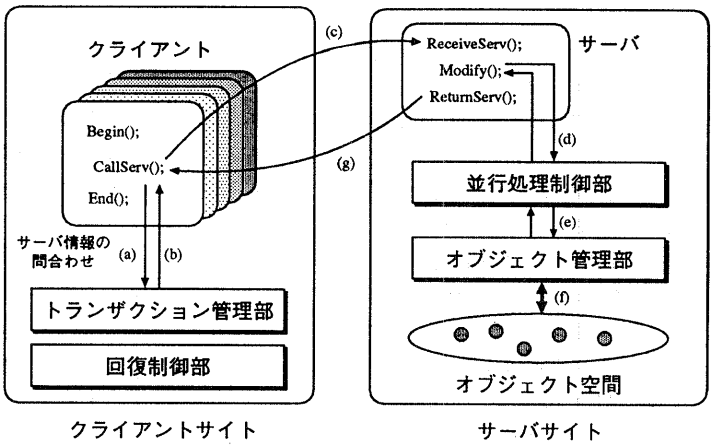


図 8 トランザクション処理の概要
Fig. 8 Outline of transaction processing.

（4）トランザクションのアボート処理

IXI においてトランザクションがアボートするのは次の場合である．

（a）データのアクセス要求時に並行処理制御部によって不当と判断された場合．

（b）コミット処理に失敗した場合．

（c）通常モード，先行コミットモード（2.6節参照）で実行中のサブトランザクションが破棄された場合．

（d）システム障害などの外的要因でシステムによって強制的にアボートされた場合．

これらの要因によってトランザクションがアボートすると，トランザクションは後退復帰された後，自動的にアボートハンドラが起動される．アボートハンドラは，後退復帰後にトランザクションの再実行と破棄のいずれの処理を選択するかを決定するための手続きであり，トランザクションごとに登録される．通常は，システムが標準で提供する標準ハンドラが用いられるが，ユーザが任意に定義することも可能である．アボートハンドラによってトランザクションの破棄が選択されると，その情報はトランザクション管理部を通じて当該トランザクションの親トランザクションに通知される．

5. 既存システムとの機能比較

本章では，既存の分散トランザクションシステムである Camelot/Avalon, Argus, ISIS と *IXI* との機

能比較を行う．

Camelot は Mach 上に実現された分散トランザクション処理システムであり，Avalon は Camelot を利用するための専用言語である．Camelot はフォールトトレラントなクライアントサーバ型のアプリケーション構築を支援する．データは回復可能オブジェクトとして実現される．クライアントトランザクションは，RPC を用いてサーバに要求を行い，サーバを介してデータにアクセスする．Camelot はC言語のライブラリとして実現されており，Mach 上であればハードウェアアーキテクチャに依存なく動作可能である．

Argus (MIT) は CLU 言語と類似した分散アプリケーション記述言語である．Argus の特徴は，言語内でアトミック配列やアトミックレコードなどのアトミックな性質を持つデータ型とトランザクションを扱える点である．これらは，ガーディアンおよびアクションと呼ばれる機構で実現される．Argus ではプログラマはアクションの並行性を意識しなければならず，相互排除を行うためには，Mutex と呼ばれる特殊なオブジェクトを使用しなければならない．

ISIS は，プログラマに分散を意識させることなくフォールトトレラントなシステムを構築する枠組を提供する．その基本となるのは，resilient オブジェクトである．resilient オブジェクトはコンポーネントで構成される．トランザクションは，RPC を用いたコンポーネント呼び出しとして実行される．データは，resilient オブジェクト内の resilient な抽象データ型として表現される．resilient オブジェクトは実行環境に合わせて複製され，障害に対する高い可用性を達成している．複製間でのデータの更新やロックの制御には，特殊なブロードキャストを用いている．ISIS の処理は，メッセージのブロードキャストに基づいているため，ブロードキャストのコストの高い環境での性能に問題がある．

IXI は代行プロセス機能を実現しており，複製機能のない Camelot や Argus と比較して高い可用性を達成しているといえる．また，上記のシステムではロックを基本とする並行処理制御を行っており，デッドロック発生時の処理が問題となるが，*IXI* では，適応型時刻印方式を用いることによってデッドロックフリーな並行処理制御を実現している．さらに，*IXI* では，上記のシステムにはない弱い一貫性を扱うことができる．これによって，入れ子型トランザクションにおいて柔軟なサブトランザクションの実行が可能となっている．

6. おわりに

本論文では，分散トランザクションシステム *IXI* の特徴となる機能について述べた．*IXI* では一貫性の制限を緩和することによって，広範な分野のアプリケーションを効率良く実行することを可能とした．また，カーネルの提供する分散共有メモリ機能を利用することによって，サーバの代行処理を実現する手法について説明した．さらに，本論文では2相コミット制御，データのバージョン管理，および従来の先行書き込みログを用いることによって，回復可能オブジェクトを実現する手法についても述べた．現在までに各部の実現および評価は終了しており，それぞれ有効な結果が得られている．異種計算機による分散システムへの対応が今後の課題である．

今後，システム全体での評価を行い，*IXI* を利用した種々のアプリケーションシステムの実現およびテストを行う予定である．

参考文献

1) Eppinger, J. L., Mummert, L. B. and Spector, A. Z.: *Camelot and Avalon—A Distributed Transaction Facility*, Morgan Kaufmann Publishers (1991).
2) Liskov, B., Curtis, D., Johnson, P. and Scheifler, R.: Implementation of Argus, *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pp. 111-122 (1987).
3) Birman, K. P.: Replication and Fault-tolerance in the ISIS System, *Proceedings of 9th ACM Symposium on Operating Systems Principles*, pp. 79-86 (1985).
4) Birman, K. P. and Joseph, T. A.: Exploiting Virtual Synchrony in Distributed Systems, *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pp. 123-138 (1987).
5) 國枝和雄，畑田孝幸，大久保英嗣，津田孝夫：適応型時刻印方式に基づく同時実行制御方式，情報処理学会論文誌，Vol. 33, No. 6, pp. 802-811 (1992).
6) Gray, J. N.: Transaction Concept: Virtues and Limitations, *Proceedings of 7th International Conference on Very Large Data Bases*, pp. 145-154 (1981).
7) Moss, J. E. B.: Nested Transactions: An Approach to Reliable Distributed Computing, MIT/LCS/TR-260 (1981).
8) Badrinath, B. R. and Ramamritham, K.: Se-

mantics-Based Concurrency Control: Beyond Commutativity, *Proceedings of 3rd IEEE International Conference on Data Engineering*, pp. 304-311 (1987).
9) Rothermel, K. and Mohan, C.: ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions, IBM Research Report RJ 6650 (1989).
10) 國枝和雄, 長野 晋, 大久保英嗣, 津田孝夫: 分散オペレーティングシステム $R^2/V3$ における分散共有メモリの実現, 情報処理学会オペレーティングシステム研究会資料, OS-51-8 (1991).
11) Schwarz, P. and Spector, A.: Synchronizing Shared Abstract Types, *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 223-250 (1984).
12) Bernstein, P. A. and Goodman, N.: Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems, *Proceedings of 6th International Conference on Very Large Databases*, pp. 285-300 (1980).

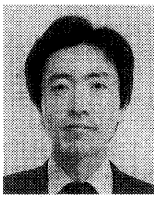
(平成4年9月28日受付)
(平成6年2月17日採録)

國枝 和雄（正会員）

昭和39年生. 昭和62年京都大学工学部情報工学科卒業. 平成元年同大学院修士課程修了. 平成4年同大学院博士後期課程単位修得認定退学. 同年日本電気(株)に入社, 現在に至る. オペレーティングシステム, データベースシステム, 分散処理等に興味を持つ. 日本ソフトウェア科学会会員.

各務 達人

昭和 42 年生. 平成4年京都大学工学部情報工学科卒業. 同年関西日本電気ソフトウェア(株)入社, 現在に至る. 在学中, 分散トランザクションシステムの研究に従事.



大久保英嗣（正会員）

1951 年生. 1974 年北海道大学理学部数学科卒業. 1977年同大学工学部情報工学科大学院修士課程修了. 同年(株)日立製作所ソフトウェア工場に入所. 主として FORTRAN コンパイラの開発に従事. 1979年より京都大学工学部情報工学科助手. 1985年同講師, 1987年同助教授, 現在立命館大学理工学部情報学科教授. 工学博士. オペレーティングシステム, データベースシステム等の研究に従事. 日本ソフトウェア科学会, システム制御情報学会各会員.

津田 孝夫（正会員）

1957年3月, 京都大学工学部電気工学科卒業. 現在京都大学工学部情報工学科教授. 計算機ソフトウェア講座担当. 工学博士. 自動ベクトル化／並列化コンパイラ, スーパコンピューティング, オペレーティングシステムの研究に従事.「モンテカルロ法とシミュレーション」(培風館),「現代オペレーティングシステムの基礎」(オーム社, 共著),「数値処理プログラミング」(岩波書店)などの著書がある. 昭和 63 年度および平成3年度情報処理学会論文賞受賞. 元本学会関西支部長. ACM, SIAM 各会員. IFIP/WC 2.5 (Numerical Software) 委員.