マルチメディア通信と分散処理 37－11
（１９８８．５．２０）

オブジェクトモデルのためのアトミックトランザクション

中島達夫　所真理雄
慶応義塾大学 理工学部

アブストラクト

アトミックトランザクションは分散システムでの信頼性を向上させるため必要不可欠なものである。本論文では、オブジェクトのセマンティックを利用して並列度を向上させるアトミックトランザクションメカニズムを提案する。

The Atomic Transaction Mechanism for the Object Model

Tatsuo Nakajima and Mario Tokoro

Department of Electrical Engineering
Keio University
3-14-1 Hiyoshi, yokohama 223

Abstruct

The atomic transaction is a very important technique for increasing the reliability of distributed system. In this paper, we propose a mechanism of atomic transaction which increases concurrency by using the semantics of an object.

1 はじめに

パーソナルワークステーションとローカルエリアネットワークの発達によりコンピュータの使用形態が変わろうとしている。ネットワークでつながれた強力なワークステーションを各自が持つことにより、幅広い情報処理やシミュレーションが実現可能となり、それによりコンピュータは単なる計算機ではなく思考の道具、コミュニケーションの道具などのハイパーメディアとしての第一歩を踏み出したと言える。

ここで重要な問題は、強力なワークステーションの能力を最大限にまで発揮させる柔軟な計算モデルの構築である。このモデルではパーソナルな環境を支援するだけでなく、ワークステーションが多数接続されたインターパーソナルな環境をも支援しなければいけない。

強力なインターパーソナルな環境を構築するために、優れた並行記述、モジュラリティ、ネットワーク透過性を持った計算モデルが必要である。このような条件を満足するモデルとして並行オブジェクトモデルが提案されている。しかし、現在提案されているモデルでは分散環境で必要な高度の並列性、柔軟な同期メカニズム、障害が起きてもシステムの一貫性が失われないようなメカニズムのいずれをも備えていない。そこで本論文では、並行オブジェクトモデルと信頼性を向上させるメカニズムとして多く用いられているアトミックトランザクションと融合することについて考察する。並行オブジェクトモデルにアトミックトランザクションを導入することにより、1つの計算をアトミックトランザクションとしてモジュール化し、データの一貫性および、プログラムのモジュール性を向上させる。

本論文では、インターパーソナルな環境を構築するためのプリミティブな道具としてSynchronizing Concurrent Object Modelを提案する。

このモデルに基づき言語を設計することにより、高度な分散アプリケーションを容易に構築することが可能となる。

2 分散システム

この節では、現在稼働中のいくつかのオブジェクトモデルに基づいた分散システムについて解説する。

2.1 V-kernel[Cheriton88]

V-Kernelはスタンフォード大学で作られた分散OSである。このシステムではカーネルはプロセスとプロセス間通信、及びメモリ管理のみからなる。そして、それ以外の機能は全てユーザレベルのサーバとして実現されている。また、ネーミングや入出力を統一的なプロトコルにより行うことによりインタオペラビリティを向上している。V-Kernelでは、各プロセスはあるプロトコルを持ったオブジェクトとして考える。このプロトコルは十分効率がよく、かつ汎用的でなければ行けない。どのため、このシステムではIPCのパフォーマンスに非常に注意を払っている。

2.2 Mach[Acceta86]

MachはCMUで作られた分散OSである。このシステムは基本的にはV-System同様プロセス、プロセス間通信、メモリ管理の基本機能のみサポートしている。しかし、それに加えて大きなアドレス空間のサポート、マルチプロセッサへの対応などの新たな機能を持つ。このシステムの特徴は、ページングをコントロールする機能、および、ネットワークプロトコルのユーザレベルでのサーバとしての実現、仮想記憶のマシンディペンデントとマシンインディペンデント部分への分離による移植性の向上、基本オブジェクトへの処理の要求をポートへのメッセージ送信とするオブジェクト指向概念の導入などである。

2.3 Sprite[Nelson88]

SpriteはUCBで作られた分散OSである。このシステムの特徴は高速な分散ファイルシステムにある。リモートサイトのファイルをアクセスすると、そのファイルのブロックはキャッシュをとられ、そのあとのアクセスからはそのキャッシュを利用し処理を高速化する。

2.4 Argus[Liskov88]

ArgusはMITで開発された分散アプリケーション開発用言語である。この言語では、各リソースはガーディアンとして抽象化され、アプリケーションからはRPCを用いてアクセスされる。また、分散処理の際のガーディアンの一貫性を保つためアトミックアクションを導入している。

2.5 Emerald[Black87]

EmaraldはWashington大学で開発された分散言語である。この言語ではプログラムに現れる全てをオブジェクトとして考える。これにより、非常に容易にアプリケーションが開発できるようになる。また、オブジェクトはどのサイトへでもマイグレーションできる。

リモートのサイトのオブジェクトにメッセージを送る時などはアーギュメントをマイグレーションすることができる。

3 アトミックトランザクション

アトミックトランザクションはデータベースの分野から生まれた概念で、共有データへの変更や障害に対して一貫性を保つための強力な手段である。アトミックトランザクションでは、実行を異常なく終了するか(これをコミットするという)、異常により実行前の状態に戻すか(これをアボートするという)のどちらかの状態しか存在しない。

アトミックトランザクションは次の3つの性質により定義される。

(1) Atomicity
(2) Serializability
(3) Permanence

Atomicityとは、すべてのアトミックトランザクションは他のアトミックトランザクションの実行前の状態と後の状態のみ見ることができることを意味する。Serializabilityとは、すべてのアトミックトランザクションのリソースへのアクセスはそれと同様な効果を生む逐次実行と同じ様に振る舞うことを意味する。Permanenceは、一度アトミックトランザクションが完了すると、そのデータは決して失われないことを意味する。

アトミックトランザクションの導入により信頼性の高い計算を行うことが可能となる。しかし、これをオブジェクトモデルに導入する際次の2つの問題がある。

(1) トランザクション内でアボートが起きると常にそのアトミックトランザクション全体がアボートする。
(2) Read-Write同期によりコンカレンシコントロールを行うとシステム全体のパフォーマンスが悪くなる。

一番目の問題を解決するため、Nested Transaction[Moss81]が提案された。これでは、トランザクションは入れ子状に呼び出され、そのルートにあたるトランザクションをトップトランザクション、その他をサブトランザクションと呼ぶ。ここで、あるサブトランザクションのアボートはそのトランザクションより下位のトランザクションのアボートのみを管理し、その上位、および同レベルのトランザクションとは無関係である。これにより、トランザクションがアボートする可能性を低くすることができ、システム全体のパフォーマンスを向上することが可能になる。

二番目の問題は、変更されるデータを含むデータ集合(例えばファイル)の全体にWriteロックをかけなければいけないために生じる。これによりこのデータはそれをアクセスするトランザクションがコミットかアボートするまで他のトランザクションからはアクセスすることが不可能となる。それを解決するため、そのデータのセマンティックを利用したコンカレンシコントロールを行う必要がある。これに関しては、今までいくつかの研究が行われているが[Schwartz84], [Weihl85]、きわめてad-hocなもので分散システム一般に用いるツールとしては問題がある。次節では、並行性が高いアトミックオブジェクトを定義するためのモデルについて述べる。

4 並行オブジェクトモデルから分散オブジェクトモデルへ

オブジェクトモデルはプログラムを開発する際の有効な手段として広く使われてきている。そして、そのモデルに並列性を導入することによりより記述性を増した並行オブジェクトモデルは今後広く使われるようになると思われる。このモデルに基づいた言語を用いることによりプログラムの開発時間を格段に減らすことができる様になる。

このモデルの多くでは、各オブジェクトはアトミックなものとして一時に一つの処理を実行するプロセッサとして考えている。これにより、空間的なモジュラリティは向上するが、時間的なモジュラリティは今までより減少することになる。つまり、全てのオブジェクトは独立であると考えられるため、メソッドコールにより生じる因果関係のシーケンスを1つのまとまりとしてとらえることが非常に困難になる。ここでは、従来のプログラムのような概念はなく、全ての計算はオブジェクト間の協調により進められる。そのためプログラムとしてのまとまりがなくなりプログ

ラムの可読性、デバッギングが非常に困難になる。我々が提案するモデルではメソッドコールの因果関係を一つのまとまりとするためアトミックトランザクションを導入する。

しかし、そのためには今までのようにオブジェクトをモニタであると考えるのでは問題がある。つまり、モニタのsignal/waitによる条件同期はアトミックアクションの一貫性を失わせる。また、一時期に一つのメソッドを実行する排他同期は並列性を格段に低くする。

そのため、このモデルではオブジェクトをマルチプルスレッドであると考える。つまり、オブジェクトに送られたメッセージはすぐに実行を開始される。しかし、それではオブジェクトの一貫性が失われる可能性があるためオブジェクト内の実行エンティティを制御するための同期をexplicitに指定しなければいけない。

5 Synchronizing Concurrent Object(SCO) Model

5.1 計算モデル

SCO Modelでは、1つのメソッドコールを1つのトランザクションと考える。また、トップトランザクションを実行するための特別なメソッドコールを別に用意する。そして、そのトランザクションから呼ばれる全てのメソッドコールはサブトランザクションであると考える。

SCOモデルでの計算は、次の4つのエンティティより構成される。

(1) オブジェクト
(2) メッセージ
(3) アクティビティ
(4) トランザクション

オブジェクトは状態とそれをアクセスするためのメソッドより構成される。メッセージはオブジェクトの特定のメソッドを呼び出すために用いられる。アクティビティはオブジェクトにメッセージが送られるごとに作られ、実際にメソッドの実行を行うエンティティである。これは、SCOモデルではサブトランザクションとしての役割も持つ。つまり、トランザクションはアクティビティの集合として実現される。この実行はいつもアトミックに行われる。

SCOモデルでは、この4つのエンティティの協調により計算が行われる。

5.2 同期メカニズム

Synchronizing Concurrent Object Model ではマルチプルスレッドを採用しているため、クリティカルリージョンを作るための同期メカニズムが必要である。また、一貫性を保つため、signal/waitに基づかない条件同期のメカニズムも必要である。この節ではそれらについて解説する。

5.2.1 Method Relation

SCOモデルでは、排他同期のメカニズムとしてMethod Relationを持っている。これはメソッドの間に排他関係を付けることにより同期を行うものである。SCOモデルではこの関係として以下の3つを持っている。

(1) No Relation
(2) Exclusive Relation
(3) Serial Relation

これらの関係は、すべての2つのメソッドの間に付けられる。この関係は、今実行中のメソッドとこれから実行しようとするメソッドの間の関係を表わす。No Relationは2つのメソッドは同時に実行できることを意味する。Exclusive Relationは今実行中のメソッドが実行を終了するまでサスペンドされることを意味する。Serial Relationでは今実行中のメソッドを呼び出したアトミックトランザクションの実行が終わるまでサスペンドされることを意味する。いま、メソッド1とメソッド2がSerial Relationにあるとするとメソッド1を呼び出したアトミックトランザクションが終了するまでメソッド2の実行はサスペンドされる。

これを実現するため、タイプスペシフィックロック[Schwartz84]を用いる。このロックはこれを実行したアトミックトランザクションがコミット/アボートするまで保存される。

5.2.2 Guarded Method

signal/waitにより条件同期を実現することは、Method Relationの並列性を極めて低くする。つまり、現在実行中のメソッドが条件同期によりまたされると、Exclusive Relation, Serial Relationによりサスペンドさ

れているメソッドはそれが終了するまで実行できないため、更にウエイトされることになる。

そのため、SCOモデルでは各メソッドにガードを持たせるようにする。そして、Method Relationをチェックする前にこのガードをチェックする。SCOではこれをGuarded Methodと呼ぶ。

例えば、キューオブジェクトを考えと、キューがいっぱいのときenqueueを実行するとそのメソッド内で条件同期サスペンドされることになる。もし、enqueueとdequeueがExclusive RelationかSerial Relationならdequeueは実行することができないためデッドロックが生じる。そのため、条件同期によりまたされる時は一時的にMethod Relationを解除しなければならない。しかし、これをアトミックトランザクションとして用いると一貫性を失う可能性が生じる。そのため、ガードによる条件同期の導入は非常に有用である。

これにより、Method Relationの並列度を最大に発揮することが可能となる。

5.3 リカバリメカニズム

この節では、SCOモデルのリカバリについて述べる。SCOではリカバリメカニズムとしてロギングを用いる。そのため、リカバリを実現するためのステーブルストレージとWrite-Ahead-Logginが必要である。SCOでは、Method Relationを効果的にするため、リカバリメソッドとして、Value Loggingだけではなく、Operation Loggingも併用する。

5.3.1 Value Logging

これは現在データベースなどで広く使われている方法である。つまり、オブジェクトの内容を変更する前に、undoとredoのログをとっておいてそれが完了した後にオブジェクトの実体を変更する。しかし、この方法は書き込んだ内容はそれを行ったアトミックトランザクションが終了するまで確定しないので、他のアトミックトランザクションはこれを読みだしたり、更に変更することができない。これはシステムのパフォーマンスを非常に悪くする。

もし、この方式だけサポートすると、前節で述べたMethod RelationはSerial Relationのみ用いることができる。(実際は、Read-Write同期を考えるとRead-ReadはNo Relation、他の関係はSerial Relationとなる。) これについては5.3.2で述べる。

5.3.2 Operation Logging

Method Relationを有効に利用するためにはこのロギングは必要不可欠である。この方法では、redo,undoログをオペレーションの形で取って置く。そして、アボートがおきるとundoログに書かれたオペレーションを実行する。そのオペレーションは実行した内容を打ち消すことができる。これは、いつでも使うことができる訳ではないが、システムの並列性を向上させるために必要不可欠である。

また、一度実行してしまうと、そのundoは値のundoとしては実行できないような問題は多く存在する。例えば、ミサイルの発射は値のundoとして実行できない。そのため、undoが必要な時は自爆するとか、そのミサイルを打ち落とすためのミサイルを発射するとかなどの実行したメソッドを補償するためのメソッド(Compensate Method)が必要になる。つまり、undoオペレーションはそのような問題に対して自然な記述を可能にする。

5.3.3 Operation LogginとMethod Relation

ここでは、カウンタオブジェクトを例に取り、Method Relationの並列性を生かすためにはOperation Loggingが必要不可欠なことを示す。カウンタオブジェクトはincrementとdecrementの2つのメソッドを持つ。オブジェクトのセマンティックを考えるとこの2つのオブジェクトはExclusive Relationとして実現できるが、リカバリメカニズムとしてValue Loggingを選ぶとそのメソッドを実行したトランザクションが終了するまで他のトランザクションはそのカウンタオブジェクトにアクセスすることができなくなる。そのため、Method RelationとしてSerial Relationを用いなければならなくなり並列性が減少することになる。そのため、Method RelationとしてNo Relation, Exclusive Relationを用いる時はリカバリメソッドとしてOperation Loggingを用いなければいけない。

5.4 コミットメカニズム

SCOではコミットメカニズムとして、ブロッキングコミット、ノンブロッキングコミットのそれぞれを用意している[Spector87]。ここでは、それ以外のコミット処理の特徴について述べる。

5.4.1 Commit Operation

これはアトミックトランザクションがコミットした時に実行するべきオペレーションである。これは、各オブジェクトの各メソッドごとに定義できる用になっていてコミット時にはそれが実行される。これは、例えばキューオブジェクトを考える時コミット時にenqueueメソッドを実行できるようにする。こうすると、コミットした内容のオブジェクトのみキューイングされるのでいつでもdequeueを実行することができる。

これは、undoが不可能で、補償する方法も存在しない場合にも適用できる。つまり、このようなオペレーションはコミット時まで実行せず、完全にそのトランザクションがアボートすることがないことがわかった時(トップトランザクションがコミットした時)に実際に実行する。

5.4.2 Early Commit

あるメソッドが次の用な関係にある時、SCOではこれをEarly Commit Methodと呼ぶ。

method1 ¯ method-i (¯ はNo Relation, Exclusive Relationのどれか)

ここで、method-iはこのオブジェクトの持つ全てのメソッドである。

このメソッドは実行終了後すぐロックを解除しコミットする。例えば、トラベラーズエージェントシステムでキャンセルメソッドを実行する時このメソッドはEarly Commit Methodとなる。そして、トランザクションをアボートするときはundo methodをトップトランザクションとして実行する。このようにすると、このオブジェクトに関してコミット時の処理が不必要になるのでコミット処理を高速化することができるようになる。

6 Synchronizing Concrrent Object Modelでの並列性

この節では、Queue, Directory, Bufferの3つのオブジェクトについてSCOモデルでの実現について考察する。

6.1 Queue

Queueオブジェクトは、dequeue, enqueueの2つのメソッドをもつ。このオブジェクトでは、ガードとコミットオペレーションを用いることにより並列性を増すことが可能となる。以下に、Queueオブジェクトの定義を示す。

Method Relation

No Relation
enqueue(a) ¯ dequeue(b)
dequeue(a) ¯ enqueue(b)

Exclusive Relation
enqueue(a) ¯ enqueue(b)

Serial Relation
dequeue(a) ¯ dequeue(b)

Can't execute
enqueue(a) ¯ dequeue(a)

Gaurded Method

[not self empty] dequeue: a
[not self full] enqueue: a

Commit Operation

enqueue(a) -> realEnqueue(a)

ここでは、並列性を増すため2つのことを行っている。1つめは、ガードの利用である。これにより、オブジェクトの状態により実行が不可能なメソッドは実行可能となるまでサスペンドされる。そのため、後から来た実行可能なメッセージを先に処理することができる。2番目は、コミットオペレーションの利用である。つまり、キューへの実際の処理はメソッドが実行された時行われるのではなく、それを実行したトランザクションがコミットする時に実行される。そのため、キューイングされたエントリは全てコミットされており、いつでもdequeueすることが可能である。ここでは、enqueueはコミットオペレーションを用いているためEarly Commit Methodにはならない。また、アボート時のリカバリは、enqueueの場合は何もせず、dequeueの場合はキューの先頭に戻す。

6.2 Directory

Directoryオブジェクトは、lookup, modifyの2つのメソッドを持つ。このオブジェクトはMethod Relationのアーギュメントを利用

して並列度を向上させている。
以下にこのオブジェクトの定義を示す。

Method Relation

Serial Relation
lookup(a) ¯ modify(a)
modify(a) ¯ lookup(a)
modify(a) ¯ modify(a)

No Relation
lookup(a) ¯ lookup(a)
lookup(a) ¯ lookup(b)
lookup(a) ¯ modify(b)
modify(a) ¯ lookup(b)
modify(a) ¯ modify(b)

一般に1つのトランザクションから同じキーのエントリに対してアクセスされることはほとんどないと仮定すると、Method Relationを次のように変更することによりより並列度を向上することができる。

Excluisve Relation
lookup(a) ¯ modify(a)
modify(a) ¯ modify(a)

また、このオブジェクトでは、リカバリ法としてvalue loggingを用いる。

6.3 Buffer

Bufferオブジェクトは、Queueオブジェクト同様dequeue, enqueueの2つのメソッドを持つ。このオブジェクトではenqueueした順番とdequeueした順番は関係ない。つまり、2つのdequeue、2つのenqueueはお互いにその順序に無関係なため次のように定義することにより並列性を向上できる。

Exclusive Relation
enqueue(a) ¯ dequeue(b)
dequeue(a) ¯ enqueue(b)
dequeue(a) ¯ dequeue(b)
enqueue(a) ¯ enqueue(b)

Serial Relation
enqueue(a) ¯ dequeue(a)

Compensate Method
enqueue(a) -> dequeue(a)
dequeue(a) -> enqueue(a)

ここでは、dequeueは他のメソッドとSerial Relationの関係にあるので、Early Commit Methodとすることができる。

6 DistributedConcurrentSmalltalk

DistributedConcurrentSmalltalkは現在我々が開発中の分散並行言語である。この言語は次の3つの特徴を持つ。

(1) 単一レベルの並行オブジェクト
(2) Transparent Remote Object Access
(3) Interpersonal Environment Support

(1)は、DCSTでのすべてのオブジェクトは並行に実行されうることを意味する。そのため、プロセスとオブジェクトという2つの抽象化が存在しなくなるためプログラミングを容易にする。

(2)は、ローカルにあるオブジェクトもリモートにあるオブジェクトも全く同じ形態でアクセスできることを意味する。これにより、分散化を意識せずプログラムを各ことが可能となる。

(3)は、各ユーザは各自のネームスペースを持つことができることを意味する。これにより、ユーザは各自の環境を構築することが可能となる。

このDCSTに前節までで述べた、SCOモデルを組み込むことを現在検討中である。

しかし、SCOモデルをそのまま実現するのは非現実的である。第一に、各メソッドコールをサブトランザクションにするのはコストが大き過ぎる。そのため、サブアクションのためにもう少し大きな単位を選べるよう検討しなければならない。第二に、オブジェクトの単位が小さ過ぎるためリカバリのコストが大きくなる。これは、オブジェクトをグループ化して物理的に大きな単位で扱えるようにするか、アーキテクチャ的なサポートを考えなければ行けない。第三に、Method Relationは、インヘリタンスの時に有用であると述べたが、No Relation, Exclusive Relationと定義されたメソッド内で他のアトミックオブジェクトにアクセスしていると問題が起きることもある。今後、これらの問題について検討しなければいけない。

7 結論

本論文では分散システムに適したSynchronizing Concurrent Object Modelについて述べた。このモデルは空間的かつ時間的モジュラリティを実現するものである。また、各オブジェクトをマルチプルスレッドにして、柔軟な同期メカニズム、リカバリメソッドを導入することにより、アトミックトランザクションを導入してもシステム全体のパフォーマンスを極端に低くすることはなくなる。

SCOモデルにより、容易にモジュラリティの高い分散アプリケーションを構築することができる。

[Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", USENIX86

[Cheriton88] D. R. Cheriton, "The V Distributed System", CACM Mar. 1988

[Liskov88] B. Liskov, "Distributed Programming in Argus", CACM Mar. 1988

[Moss85] J.E.B. Moss "Nested Tranzaction: An approarch to reliable distributed computing", MIT Press 1985

[Nelson88] M. W. Nelson, B. B. Welch and J. K. Ousterhout, "Cashing in the Sprite network file system", ACM TOCS Feb. 1988

[Schwartz85] P.Z. Schwartz, "Transaction on Typed Object", CMU-CS-84-166

[Spector83] A.Z. Spector and P.Z. Schwartz, "Transactions: A Construct for Reliable Distrubuted Computing", CMU-CS-82-143 1983

[Spector87] A. Z. Spector et al. "Camelot: A distributed tranzaction facility for Mach and Internet - An interim report", CMU-CS-87-129 1987

[Weihl85] W. Weihl and B. Liskov, "Implementing of resilient, atomic data types", ACM TOPLAS Apr. 1985

[Yonezawa87] A. Yonezawa and M. Tokoro, "Object-Oriented Concurrent Programming" MIT Press 1987

[永松87] 永松竜夫、 中島達夫、 所真理雄 "分散OS: TonTosのシステムインタフェース言語DCSTの設計と実装" 4th Proc. of JSSST 1987

[落合87] 落合真一、中島達夫、所真理雄 "分散OS: TonTosのシステムインタフェース言語DCSTの分散化" 4th Proc. of JSSST 1987