

06-開-15

データベース構築促進及び技術開発に関する報告書

フォールトトレランスな多データベース
サーバシステムに関する調査研究

平成7年3月

財団法人 データベース振興センター

委託先 株式会社シネジャーナルプロダクション



この事業は、競輪の補助金を受けて実施したものです。

序

データベースは、わが国の情報化の進展化、重要な役割を果たすものと期待されている。今後、データベースの普及により、わが国において健全な高度情報化社会の形成が期待される。さらに海外に対して提供可能なデータベースの整備は、国際的な情報化への貢献および自由な情報流通の確保の観点からも重要である。しかしながら、現在わが国で流通しているデータベースの中でわが国独自のものは1/3にすぎないのが現状であり、わが国データベースサービスひいてはバランスある情報産業の健全な発展を図るためには、わが国独自のデータベースの構築およびデータベース関連技術の研究開発を強力に促進し、データベースの拡充を図る必要がある。

このような要請に応えるため、財データベース振興センターでは日本自転車振興会から機械工業振興資金の交付を受けて、データベースの構築および技術開発について民間企業、団体等に対して委託事業を実施している。委託事業の内容は、社会的、経済的、国際的に重要で、また地域および産業の発展の促進に寄与すると考えられているデータベースの構築とデータベース作成の効率化、流通の促進、利用の円滑化・容易化などに関係したソフトウェア技術・ハードウェア技術である。

本事業の推進に当って、当財団に学識経験者の方々と構成されるデータベース構築・技術開発促進委員会（委員長 前山梨学院大学教授 蓼沼良一氏）を設置している。

この「フォールトトレランスな多データベースサーバシステムに関する調査研究」は平成6年度のデータベースの構築促進および技術開発促進事業として、当財団が(株)シネジャーナルプロダクションに対して委託実施した課題の一つである。この成果が、データベースに興味をお持ちの方々や諸分野の皆様方のお役に立てば幸いである。

なお、平成6年度データベースの構築促進および技術開発促進事業で実施した課題は次表のとおりである。

平成7年3月

財団法人 データベース振興センター

平成6年度 データベース構築・技術開発促進委託課題一覧

分野	課題名	委託先
社会	1 報道ニュースのマルチメディアデータベース構築	日本電子計算(株)名古屋支店
	2 携帯型電子新聞プロトタイプの開発	(株)日本経済新聞社/(株)日経データ
	3 SPORTS POWER INDEXのパイロット版データベースの構築	(株)ビデオ・リサーチ
	4 画像付き地図資料データベースのプロトタイプ の作成	(財)地図情報センター
	5 パソコンを利用した副作用症例データベースの 構築調査	協立医師協同組合
中小企業振興 地域活性化	6 関西イベント&プロジェクトデータベース構築	(株)京都新聞社
	7 ハイビジョン大型映像ソフト流通促進用データ ベース構築	(財)大阪科学技術センター
	8 分散協調型データベースのための高度運用シス テムの開発と学習支援環境への応用	(株)エマーズ
海外	9 電子デバイス情報の電子化ドキュメントのプロ トタイプ作成	電子デバイス情報サービス(株)
	10 海外ユーザ向けの国産データベース検索支援用 端末ソフトウェアの開発	カテナ(株)
技術	11 連想検索写真データベースのプロトタイプ作成	シャープ(株)
	12 レーザー研究データベースの構築	(財)応用光学研究所
	13 砂漠化防止技術データベース	(株)新産業創造センター
	14 メディア変換型感性データベースの構築方法の 研究	(財)イメージ情報科学研究所
	15 フォールトトレランスな多データベースサーバ システムに関する調査研究	(株)シネジャーナルプロダクション
	16 創出キーワードの自動付与に関する調査研究	(株)エレクトロニック・ライブラリー

目 次

I. 課題性の研究

II. フォールトトレランスな多データベースサーバシステム

1 章 序	3
2 章 データベースシステム	7
2.1 統合化から分散へ	7
2.2 リレーショナルモデル	9
3 章 分散データベースシステムにおける計算モデル	16
3.1 クライアントサーバモデル	16
3.2 データの分散方法	18
4 章 障害	24
5 章 トランザクション	27
5.1 トランザクションの定義	27
5.2 実行履歴	30
5.3 正しくない履歴	34
6 章 同時実行制御	37
6.1 直列可能性	37
6.2 二相ロック	40
6.3 デッドロック	47
7 章 コミットメント制御	56
7.1 障害	56
7.2 二相コミットメント	58
7.3 終結プロトコル	60
7.4 障害復旧	62
7.5 評価	63

7.6 三相コミットメント	64
8章 分散質問処理.....	66
9章 多重化.....	70
9.1 システム構成	70
9.2 障害	71
9.3 多重化方式	72
10章 複合多重化.....	76
10.1 レプリカグループ.....	76
10.2 指揮レプリカグループ.....	77
10.3 候補レプリカグループ.....	79
10.4 関連レプリカグループ.....	79
10.5 複合的多重化のモデル.....	79
10.6 各モデルの考察.....	86
10.7 モデル3、モデル4と能動的多重化の比較.....	87
11章 今後の課題.....	88

I. 課題性の研究

1. 目的

ワークステーションおよび通信ネットワークの技術革新により、データベースを複数のワークステーションから構成させ、これによってシステムのコストパフォーマンスの向上が図られつつある。こうしたデータベースシステムの多くは、クライアントとサーバを基本要素として、異なったワークステーションに機能分散する形態をとっており、このため各ユーザはクライアントを通じて複数のデータベースサーバに対する操作を行うことができる。

一方、近年、光ファイバをベースとした極めて高速の通信ネットワークが急速に普及しつつあり、これによってネットワークの信頼性は飛躍的に向上している。しかし、一方ではワークステーションに対する低信頼性が指摘されており、こうした中でシステム全体の信頼性をいかに高めていくかが課題となっている。

こうした視点から、高速ネットワークを介した分散データベースの信頼性を確保する方策について研究を行い、もってわが国のデータベースの振興に資することを目的とする。

2. 実施内容

本認査研究では、以下の項目を主たる内容としている。

1) システムフォールトの種類の検討

ワークステーションおよびネットワーク上の障害の種類についての検討。

2) 多重化方式

データベースサーバの障害に対する、信頼性と可用性の向上方策の研究。

3) 同期方式

障害が発生した時点での、データベースサーバ間、クライアントとサーバ間、クライアント間における同期方式についての研究。

4) グループ通信

複数のサーバとクライアントをグループとした信頼性のある通信方式の検討。

5) フォールトトレランスな多データベースサーバシステムのアーキテクチャ

サーバおよびクライアントの障害に対してデータベースのアクセスを提供するシステムアーキテクチャについての調査・研究。

6) フォールトトレランスな多データベースサーバシステムの今後の課題

データベースサーバ、クライアント等の多用化、高機能化等にもなう今後の課題についての検討。

3. 実施体制

株式会社シネ・ジャーナルプロダクション内に当該事業遂行のための事務局を設置し、このもとで「調査・研究会」を編成して、調査研究を実施した。

調査研究会メンバー

滝沢 誠（東京電気大学理工学部・経営工学科教授）

山鳥雄嗣（日本情報処理開発協会・調査部長）

市川 隆（A I・ファジー振興センター所長）

道田国雄（新世代システムセンター企画部長）

近藤伸一（イフ・アドバタイジング調査部長）

Ⅱ．フォールトトレランスな多データベース サーバシステムの研究

1 章 序

近年のワークステーションと通信ネットワークの発展により、データベースシステムを複数のワークステーションから構成することが、コストパフォーマンスの点から可能となってきた。また、通信ネットワークは、これまでのEthernetを中心としたローカルエリアネットワーク(LAN)と公衆通信ネットワークを中心とした構成から、光ファイバとATM交換機を用いたGbps程度の通信速度の高速ネットワークへと変化してきている。

現在の情報システムは、こうした高速ネットワークを用いて、複数のワークステーションが相互接続された形態となってきた。こうした情報システムでは、データベースシステムが重要なコンピュータ資源である。データベースシステムも、一つの大型汎用コンピュータ内ではなく、高速ネットワークにより相互接続された複数のワークステーションから構築されてきている。いわゆるクライアントサーバモデルである。データベースシステムは、クライアントとサーバを基本要素として、これらが異なったワークステーションに機能分散された形態をとっている[ORAC92,SYBA90]。クライアントは、データベースシステムのユーザのインタフェース機能を持ち、サーバは、データベースに対する基本操作機能を持つ。一台のコンピュータ内に実現されていたデータベースシステムをクライアントとサーバの機能に分割し、おののおを異なったコンピュータ内に実装することにより、負荷の分散と、ユーザに容易なデータベース操作を提供しようとするものである。サーバは、複数のクライアントからのデータベース操作要求を処理する。さらに各ユーザは、クライアントを通じて、複数のデータベースサーバに対する操作を行える。

光ファイバ等を用いた通信ネットワークは、Gbps程度の伝送速度をもち高速である。さらに、通信ノイズに対して頑強であり(10^{-9} 程度)、通信誤りはほとんどないと考えられるほど高信頼となってきた。これに対して、現在のワークステーションは、従来の大型汎用コンピュータと比較しても、その信頼性と可用性は高くない。高速通信ネットワークにより相互接続された複数のワークステーションから校正されるシステムでの一つ

の重要な課題は、各ワークステーションが低信頼であるもとの、どのようにシステム全体の信頼性を高めていくかである。特に、データベースサーバの信頼性が高くないとしたときに、どのようにして、データベースのサービスを提供しつづけるか、さらに、応答時間等のサービスの品質を維持するかが問題となる。航空管理システムや銀行の現金自動支払システムといった情報システムでは、安全の確保や金銭の厳密な計算を行なう必要がある。このため、種々の障害に対して、仕様どおり計算を継続できかつ信頼できる(*dependable*)システムであることが望まれている。システムの構成要素の誤動作にもかかわらず、システムとしては仕様どおりに正しく動作し続ける能力をフォールトトレランス(*faulttolerance*)という。本報告書では、タンデム社のノンストップシステム[GRAY86]のような特別なハードウェアを用いずに、ワークステーション等の商用に広く普及したコンピュータとローカルエリアネットワーク(LAN)等の既存のネットワークを用いて実現する方法を考える。

ここでは、データベースサーバを複数の異なったコンピュータに配置し多重化することを試みる。データベースサーバの複数のコンピュータに冗長に配置するとき、配置された各コピーをレプリカ(*replica*)とする。あるコンピュータの障害に対しても、残りの動作中のコンピュータによりデータベースのサービスを提供することができる。これにより、フォールトトレラントなシステムを実現することを試みる。多重化の方法として能動的、受動的、準能動的の三種類の方法が、[POWE94]で示されている。能動的多重化では、全レプリカが同じ動作を行なうことからメッセージ数と計算時間が大きくなる。準能動的と受動的では、出力を行うレプリカが1つしかないために、ビザンチン障害を起こしたときに頑強性がない。本報告書では、メッセージ数と計算時間を小さくし、かつビザンチン障害[LAMP82]に頑強な多重化方式を考える。コンピュータの障害に対して、データベースシステムのサービスを中断することなく、かつサービスの品質を低下させることなく提供させるための複合方式を示す。これにより、データベースシステムが、高速通信ネットワークにより結合された複数のデータベースサーバから構成されるシステム形態のもとで、システム全体をフォールトトレランスとする方式について考察することとし、以下の研究を行う。

1. システムのフォールトの種類の検討：分散システムがこうむる障害として、どのようなものがあるかを検討する。ワークステーションの障害として、従来から検討さ

れている停止障害に加えて、各サーバとクライアントが誤動作するビザンティン障害も検討する。さらに、通信ネットワークでの障害として、ネットワークの分割について考察する。

2. 多重化方式：複数のサーバから構成されるデータベースシステムで、信頼性と可用性を向上するためには、データベースシステム内のデータを複数のサーバに冗長に配置することである。データベースサーバの種々の障害に対して、信頼性と可用性を向上するためのデータの冗長配置方法について検討する。ここでは、従来の能動的、受動的、準能動的多重化[GRAY86]に対して、ビザンティン障害に対する頑強性を提供するとともに通信量と処理量を低減するために、新たに複合多重化方式を示す。
3. 同期方法：分散したデータベースシステムでは、障害が発生するもとで、データベースサーバ間、クライアントとサーバ間、クライアント間で種々の同期をとる必要がある。
 - (a) 各サーバ間でのデータの一致性を保つための同時実行方式。各データオブジェクトが、冗長に配置されている場合の同時実行方式を検討する。各トランザクションが、従来のように一時に一つのオブジェクトを要求する場合だけでなく、 n 個を同時に要求し、これの全てを待つ場合および、この中の一つの獲得を待つ場合等でのデッドロックの解決方式を研究する。
 - (b) 各サーバとクライアントが障害を起こすもとで、トランザクションが複数のサーバ内のデータを更新するときの原子性、即ち、全てのサーバを更新したかまたは全く更新しなかったかを保障するためのコミットメント制御方式を検討する。これまで、サーバまたはクライアントの停止障害のみが想定されていたが、ここではビザンティン障害についても検討する。
 - (c) 複数のサーバ間で任意にチェックポイントを取ると、あるサーバが故障した後、復旧した場合に、サーバ間でのデータが矛盾したものとなる場合がある。このために、サーバ間で無矛盾なチェックポイントをとる方式と、サーバの復旧方式について検討を行う。
 - (d) 分散システムでは、各ワークステーション間でのローカル時間の同期をとる必要があるため、ローカル時間の同期方式について検討する。また、サーバとクライアント間での種々の合意方式について検討する。

4. グループ通信：分散システムでは、これまでの二つのプロセス（サーバまたはクライアント）間の通信に加えて、複数のサーバとクライアントを一つのグループとした通信が必要となる。本研究では、サーバとクライアントが障害を起こすもとで、高速ネットワークを用いて信頼性のあるグループ通信を提供する方式を研究する。
5. フォールトトレランスな多データベースサーバシステムのアーキテクチャ：サーバとクライアントの種々の障害に対して、データベースへのアクセスを提供するシステムのアーキテクチャを検討する。

第2章では、フォールトトレラント分散データベースシステムを考えるときの基礎となるデータベースシステムの基本概念について述べる。第3章では、分散データベースシステムの計算モデルを示す。第4章では、分散データベースシステムがこうむる障害について論じる。第5章では、トランザクションの概念を示す。第6章では、同時実行制御について述べる。冗長なデータを更新するための同期方式を示す。第7章では、コミットメント制御方式を述べる。第8章では、SQL質問を複数のデータベースサーバにより処理する方式を示す。第9章では、従来の多重化方式を述べる。第10章では、複合多重化方式を論じる。

2章 データベースシステム

まず、フォールトトレラントな分散データベースシステムを考えるときに必要となるデータベースシステム[TAKI92]の概念について簡単に述べる。

2.1 統合化から分散化へ

データベースシステムとは、実世界をコンピュータ化したものである。学校の例を考えてみる。学校では、先生は、学生の成績を管理するために学生の成績のデータを利用し、事務は学生のクラブ活動を管理するために学生のクラブ活動のデータを利用する。データベースシステムは、学校での成績管理、クラブ活動管理等の学校の業務に必要な学生についてのデータを記憶するシステムである。データの集まりをデータベース、成績管理等の業務を応用とし、業務を行うためにデータベースを利用するプログラムを応用プログラムとする。

コンピュータが登場してから1960年代まで、各応用プログラム毎に、必要とされるデータがファイルとして管理されてきた。ファイルとは、プログラムが必要とするデータをプログラムの操作方法に合う形式で集めたものである。例えば、成績管理では、学生の成績順に学生データを利用するとき、学生のデータを成績順に整列させて記憶したものが成績管理のためのファイルである。このようなシステムをファイルシステムという。コンピュータ利用の拡大にともない、1950年代後半頃から以下の点が問題となってきた。

1. ファイル間でのデータの一致性の保持が困難である。例えば、同じ学生のデータが複数のファイルに存在すると、本人のデータが変化したとき、全ファイル内のデータを更新する必要がある。
2. 複数ファイル内に同一データが冗長に存在するので、記憶負荷が増大する。
3. ファイル毎にデータの表現形式が異なることにより、ファイルシステム間でのデータの共有が困難である。例えば、あるファイルでは、学生の名前をローマ字で示し、他のファイルでは、カタカナで示している場合である。

以上の問題点を解決するために、1960年代初頭に、各応用のファイル内に記憶されていたデータを一箇所に集める必要が生じてきたが、これがデータベースのはじまりと言える。

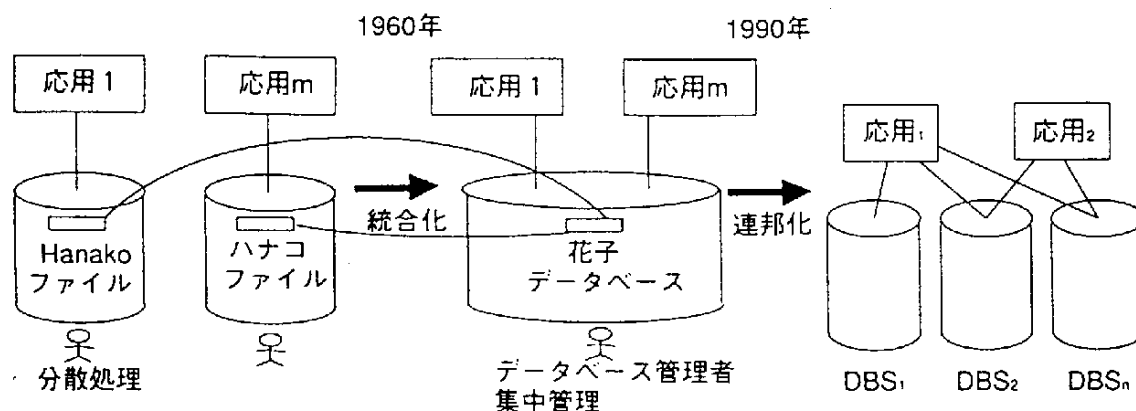


図2.1 データベースシステムの歴史

例えば、複数の応用で利用されていた学籍番号、住所、クラブ等の種々の学生データを、各学生毎に一つのデータを作ることである[図2.1]。これを統合化(integration)という。

1960年代から1970年代の情報システムでは、大型汎用コンピュータを中心として、利用者は、端末を通じてデータベースシステムを利用していた。1980年代から、ワークステーションが普及し、これらはローカルエリアネットワーク(LAN)といった通信ネットワークにより相互接続されている。こうした情報システムでは、データベースシステムはワークステーションにも構築されてきている。また、データベースシステムの利用者インタフェースの機能をクライアント、データベースの基本操作機能をサーバとして、各々を、異なったワークステーションに分散したクライアントサーバモデルのシステムが利用されてきている。これまで、各応用は、一つのデータベースを利用していたが、クライアントから複数のデータベースサーバの利用が可能となった。このように、データベースシステムは、1980年代から、再度分散型の形態となってきた。ファイルシステムのように、各応用毎に完全に分散した形態ではなく、各応用単位に管理されてきたデータベースシステムが、情報システム全体としての緩やかな管理に従っていく形態である。これは、ファイルシステムの完全分散化と、データベースシステムの完全集中化に対して、この中間にあるもので、連合化(federation)[SHET90]ともいわれる。本報告書では、ワークステーション等の信頼性の高くないコンピュータを複数用いて、データベースシステムのサービスを高い信頼性をもって提供するフォールトトレランスシステムについて考える。

2.2 リレーショナルモデル

利用者からデータベースシステムがどのようにみえるかを示したものがデータモデルである。現在の主要なデータベースシステムは、リレーショナルモデル[CODD70]に基づいている。リレーショナルモデルは、データ構造、データ構造に対する操作演算、データ構造が保つべき制約としてのインテグリティ制約により与えられる。

2.2.1 データ構造

リレーショナルモデルのデータ構造は、テーブル(table)から構成される。テーブルの集まりを、リレーショナルモデルのデータベースとする。テーブルの例を図2.2に示す。ここで、従業員の情報を示すテーブルEmpは、従業員の番号(eno)、名前(ename)、給料(sal)、所属している部の番号(dno)に関する情報を持っている。これらを属性またはカラムとする。テーブル名と属性の構成をテーブルのスキーマ(schema)とする。もう一つのテーブルDeptは、部についての番号(dno)、名前(dname)の情報を持っている。テーブル内の各情報、例えば、<1, A, 150, 1>を行(または、組)とする。

Emp	eno	ename	sal	dno
	1	A	150	1
	2	B	200	1
	3	C	300	3
	4	A	100	2
	5	D	280	3
	6	E	450	1
	7	F	130	2
	8	D	310	2
	9	G	200	1
	10	H	280	1

Dept	dno	dname
	1	DDD
	2	EEE
	3	FFF
	4	GGG

図2.2 テーブルの例

2.2.2 操作演算

リレーショナルモデルの基本操作演算を代数演算という。テーブルを行の集合として考え、集合に対する和(union)、積(intersection)、差(difference)、直積(direct product)といった集合演算に加えて、制限、射影、結合演算といったリレーショナルモデル固有の演算がある。

1. 制限(restriction)

テーブルから、ある条件を充足する行の集合を求める演算である。例として、 $\text{Emp}[\text{sal} \geq 200 \wedge \text{eno} < 5]$ は、テーブルEmp[図2.2]から、給料(sal)が200以上で、従業員番号(eno)が5より小さいものを導出する演算である。この結果、図2.3に示すテーブルが得られる。

Emp	eno	ename	sal	dno
	2	B	200	1
	3	C	300	3

図2.3 制限の例

2. 射影(projection)

テーブルから、一部のカラムを取り出す演算である。例として、 $\text{Emp}[\text{dno}]$ は、Empからdnoの値だけを導出する演算である。この結果を図2.4に以下に示す。ここで、射影の結果もテーブルであることから、冗長な行が含まれていないことに注意されたい。

dno
1
2
3

図2.4 射影の例

3. 自然結合(natural join)

二つのテーブルを結合する演算である。例として、EmpとDept間の自然結合 $\text{Emp} * \text{Dept}$ の結果として導出されるテーブルを図2.5に示す。テーブルEmpとDept内で、共通の属性dnoの値が同じ行が結合される。

eno	ename	sal	dno	dname
1	A	150	1	DDD
2	B	200	1	DDD
3	C	300	3	FFF
4	A	100	2	EEE
5	D	280	3	FFF
6	E	450	1	DDD
7	F	130	2	EEE
8	D	310	2	EEE
9	G	200	1	DDD
10	H	280	1	DDD

図2.5 自然結合の例

2.2.3 SQL

いま述べてきた結合、射影等の代数演算は、リレーショナルモデルの基本演算であり、コンピュータのアセンブラに対応した演算である。これに対して、C言語等の高級言語に対応したものとして、SQL(structured query language)[DATE87]がある。SQLは、第一階論理に基づいた非手続き的な高水準言語であり、国際標準言語である。SQLは、スキーマの定義、消去を行うデータ定義機能と、テーブル内の行の操作を行うデータ操作機能を持つ。まず、図2.2に示したテーブルEmp(eno, ename, sal, dno)のスキーマは、以下のcreate文により定義できる。属性eno、sal、dnoのデータ型は整数(int)であり、enameは文字列(char)である。また、enoは、空値を取れない。

```
create table Emp
```

```
(eno int NOT NULL, /* 属性enoのデータ型は整数で、空値が許されない*/
```

```
ename char(36),sal int,dno int)
```

次に、ここまで示してきた二つのテーブルEmpとDept（図2.2）に対するSQLによるデータ操作（検索と更新）の例を説明する。

1. DDD部員のなかで、給料が200以上の者とその給料を求めよ。

```
select ename, sal from Emp, Dept
```

```
where dname = "DDD" and Dept.dno=Emp.dno and sal >= 200
```

Dept.dnoは、dnoがテーブルDeptの属性であることを示す。また、SQLでは以下のようにも書ける。inの右側にはselect...という検索式が書かれている。このような検索式を入れ子形式という。このinは英語の関係代名詞を示し、dnoを修飾している。

```
select ename, sal from Emp
```

```
where sal >= 200 and dno in(select dno from Dept where dname="DDD")
```

2. DDD部員のなかで、給料が200以上の者とその給料を、給料の昇順に求めよ。

```
select ename, sal from Emp
```

```
where sal >= 200 and dno in (select dno from Dept where dname="DDD")
```

```
order by sal
```

3. 部毎の平均給料を求めよ。

```
select dno, avg(sal) from Emp
```

```
group by dno
```

4. DDD部員の給料を1.5倍にする。

```
update Emp set sal= sal*1.5
```

```
where dno in(select dno from Dept where dname="DDD");
```

5. DDD部員を消去する。

```
delete from Emp
```

```
where dno in(select dno from Dept where dname="DDD");
```

6. 組<11, K, 210, 1>テーブルEmpに追加する。

```
insert into Emp(eno, ename, sal, dno)
```

```
values(11, K, 210, 1)
```

7. DDD部員として、Kを追加する。

```
insert into Emp(eno, ename, sal, dno)
```

```
select 11, "K", 210, dno from Dept where dname="DDD"
```

2.2.4 視 野

リレーショナルモデルでは、データベース内のテーブルを基本テーブルとする。利用者は、これらの基本テーブル内の一部を利用する。利用者が利用の度に、テーブルの結合をSQLにより行うことは、容易でない。このために、利用者が必要とするデータを、基本テーブルから集めた仮想的なテーブルを定義する機能がある。この仮想的テーブルを視野(view)とする。例えば、テーブルEmpとDept[図2.2]の上に、従業員名と彼の属している部名を与える視野は以下のように定義できる[図2.6]。

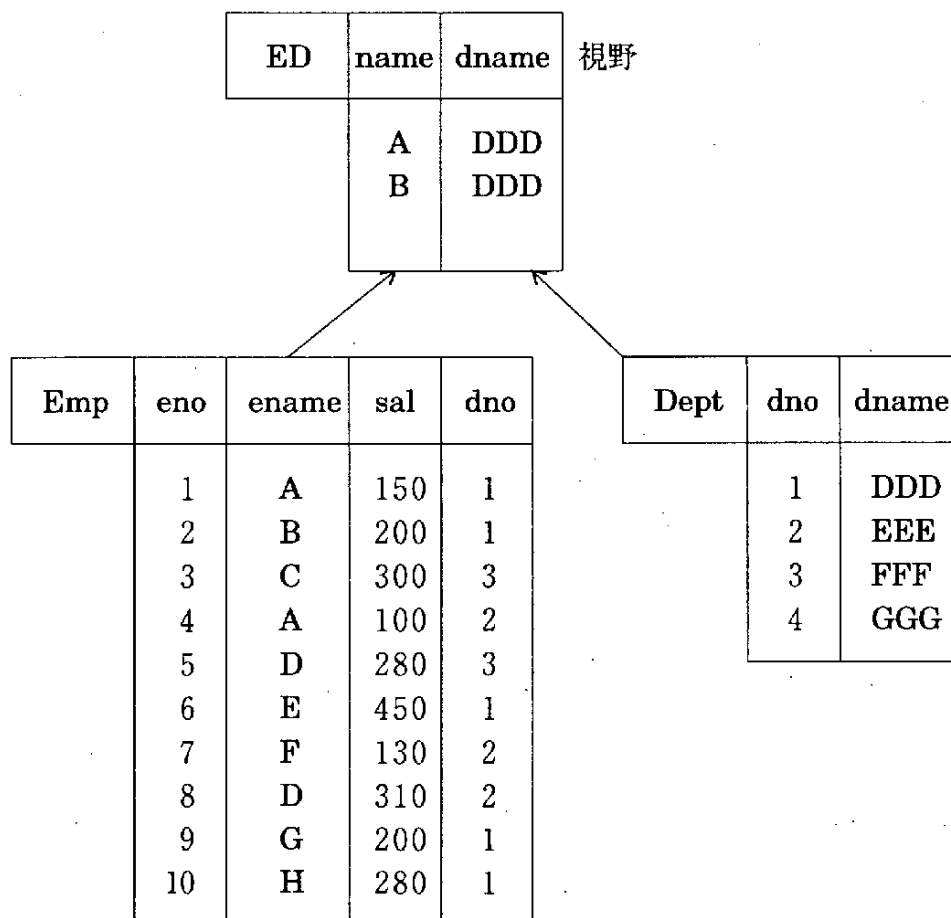


図2.6 視野の例

```
create view ED(name, dname)
as select ename, dname from Emp, Dept
where Emp.dno=Dept. dno
```

検索演算「DDD部の部員を求めよ」は、EDを用いて次のように書ける。

```
select name from ED where dname="DDD"
```

ここで、利用者は、**Emp**と**Dept**の結合を行う必要がない。データベースシステムは、このSQLから、視野の定義文を用いて、以下のSQLを作成し、実行する。

```
select ename from Emp, Dept
where Emp.dno=Dept.dno and dname="DDD"
```

2.2.5 インテグリティ制約

データ構造が保つべき条件を、インテグリティ制約という。リレーショナルモデルのデータ構造が保つべきインテグリティ制約について考える。まず、キー(key)の概念がある。

【キー】 キーとは、テーブル内の各行を一意に識別できる極小の属性集合である。

例えば、図2.2の**Emp**で、**eno**がキーとなる。属性集合{**eno**,**sal**}の値も、各行で一意であるが、これから**sal**を除いてもキーとなるので、{**eno**,**sal**}は極小ではない。このために、{**eno**,**sal**}はキーではない。テーブルが複数のキーを持つとき、その一つを主キーとする。キーについて、以下のインテグリティ制約がある。

【実体インテグリティ制約】 キーの値は、空値であってはならない。

Deptの**dno**もキーである。**Emp**も**dno**を持つ。**Emp**の**dno**は、キーではないが、**Dept**の**dno**はキーである。このような**Emp**の**dno**を外来キー[図2.7]とする。外来キーと主キーの間には、以下のインテグリティ制約がある。

【参照インテグリティ制約】 外来キーの**Emp**の**dno**の値は、**Dept**の**dno**の値として存在するか、または空値でなければならない。

外来キー					主キー		
Emp	eno	ename	sal	dno	Dept	dno	dname
	1	A	150	1		1	DDD
	2	B	200	1		2	EEE
	3	C	300	3		3	FFF
	4	A	100	2		4	GGG
	5	D	280	3			
	6	E	450	1			
	7	F	130	2			
	8	D	310	2			
	9	G	200	1			
	10	H	280	1			

図2.7 主キーと外来キーの例

2.2.6 特 徴

リレーショナルモデルは、以下の特徴を持っている。

1. 簡単なテーブル形式をしたデータ独立なデータ構造を持つ。データ構造は、テーブルだけである。
2. 非手続き的データ操作言語SQLを持つ。操作対象のデータを、第一階述語論理により、記述できる。
3. 閉包性がある。データベース内のテーブルら導出された結果はテーブルであり、再びデータ操作言語で操作できる。この性質は、データ構造を柔軟に変更できることを意味している。
4. CAD、個人用といった非定形的な業務に適する。

3章 分散データベースシステムにおける計算モデル

ワークステーションの高性能化、低価格化と、通信ネットワークの発展により、データベースシステムの構成が変化してきている。一つは、データベースシステムの利用者インタフェースの機能と、データベースに対する基本操作機能を分離し、おのこの機能をクライアントとサーバとして、異なったワークステーションに分散するものである。さらに、独立に設計され運用されてきた複数のデータベースサーバを利用することも可能となってきた。ここでは、こうした複数のデータベースサーバから構成される分散データベースシステムについて述べる。

3.1 クライアントサーバモデル

データベースシステムの機能は、大別して、利用者インタフェースと、データベースに対する基本操作の二つがある。利用者インタフェースでは、利用者に対して容易なデータベースの操作と、応用プログラムの開発を行えることが必要となる。ここでは、ウィンドウ等を用いたインタフェースを利用者に提供される。これに対して、データベースの基本操作では、複数の利用者からのデータ操作要求に対して、データベースのインテグリティ制約を保ちながら要求される性能を提供する必要がある。利用者インタフェースの機能を持つシステムをクライアントとし、後者のデータベースの基本操作機能を持つシステムをサーバとして分離する。クライアントとサーバを異なったコンピュータに配置する。これにより、以下を達成できる。

1. データベースシステムの機能をクライアントとサーバに機能分散することにより、各コンピュータの負荷を低減できる。
2. クライアントを通じて、利用者に容易なデータベースシステムのインタフェースを提供できる。

このように、現在のデータベースシステムは、クライアントとサーバから構成される。これを、クライアントサーバモデルのデータベースシステムという。クライアントサーバモデルのデータベースシステムでは、各サーバは複数のクライアントからのデータ操作要求の処理を行う。さらに、利用者はクライアントを通じて複数のデータベースサーバを利

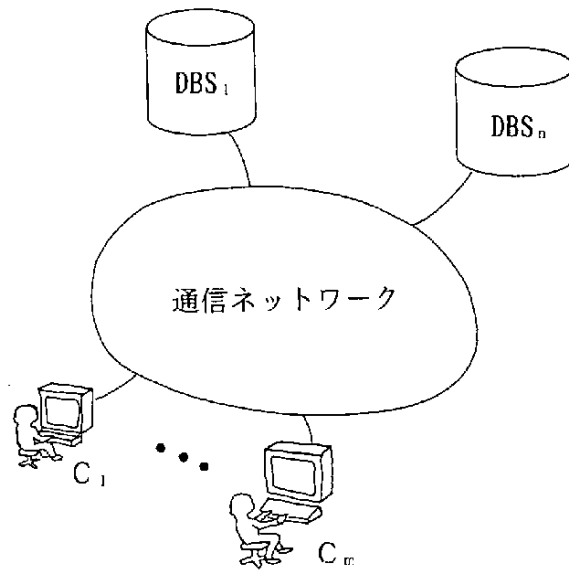


図3.1 システムモデル

用できる。このように、複数のデータベースサーバとクライアントから構成されるシステムを分散データベースシステムとする。分散データベースシステムは、複数のデータベースサーバ $DBS_1, \dots, DBS_n$ とクライアント $C_1, \dots, C_m$ から構成される[図3.1]。

各データベースサーバ DBS_i は、リレーショナルモデルのデータベース DB_i を持ち、これに対するに基本的なデータ操作、即ちSQLを実行する。クライアント C_j は、利用者からのデータ操作要求を受け取り、サーバ DBS_j にSQLで表現されたデータ操作演算 op_j を送信する。サーバ DBS_i は、 C_i からの op_i を受け取り、これを実行し、結果を C_i に返す。

各データベース DB_i は、テーブルから構成される。これを物理テーブルとする。分散データベースシステム内の物理テーブルの集合を物理データベース PDB とする。利用者が物理データベースを操作するためには、各テーブルがどのデータベースサーバに存在するか、また同じデータが複数のテーブルに存在する場合どのテーブルを操作すべきかといったことを知る必要がある。これらのことと独立に分散データベースシステムを利用できることが望ましい。これを分散の独立性という。

データベース DB_i 内のテーブルがインテグリティ制約を満足しているとき、 DB_i は正しいとする。

3.2 データの分散方法

分散データベースシステムの目的の一つは、利用者にデータの分散の独立性を提供することである。利用者から見える分散データベースシステムのデータベースは、テーブルの集合である。これを論理テーブルとし、その集まりを論理データベース LDB とする。論理テーブル内の値は、実際にはあるデータベースサーバ DBS_i 内の物理的テーブル内に格納される[図3.1]。ここでは、論理テーブルをどのように物理テーブルとして格納するかについて考える。これをデータの分散方法とする。データの分散方法としては、以下がある。

[分散方法]

- (1) 断片化。
- (2) 冗長化。

論理的テーブルを、射影演算と制限演算により分割した各要素を断片(*fragment*)とする。各断片を一つの物理的テーブルとして、あるデータベースサーバに格納する。断片化には、大別して、垂直断片化と水平断片化とがある。

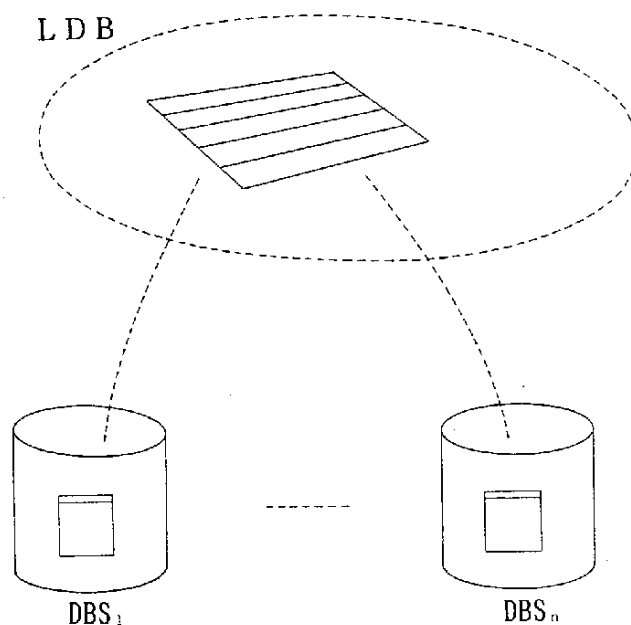


図3.2 論理テーブルと物理テーブル

3.2.1 垂直断片化

まず、垂直断片化について考える。図3.3に垂直断片化の例を示す。論理的テーブル **Emp** を射影断片 **Emp1 = Emp[eno, ename]** と **Emp2[eno, dno]** に分割し、各々をデータベースサーバ **DBS₁** と **DBS₂** に記憶した例である。ここで、**Emp1** と **Emp2** を自然結合した結果は **Emp** とならねばならない（即ち、**Emp = Emp1 * Emp2**）。このために、各断片は、**Emp** の主キー **eno** を含む必要がある。例えば、**Emp3 = Emp[ename, dno]** を考え、**Emp1** と **Emp3** の二つの断片に分割するとする。**Emp1 * Emp3** は **Emp** と異なったテーブルになってしまう[図3.4]。

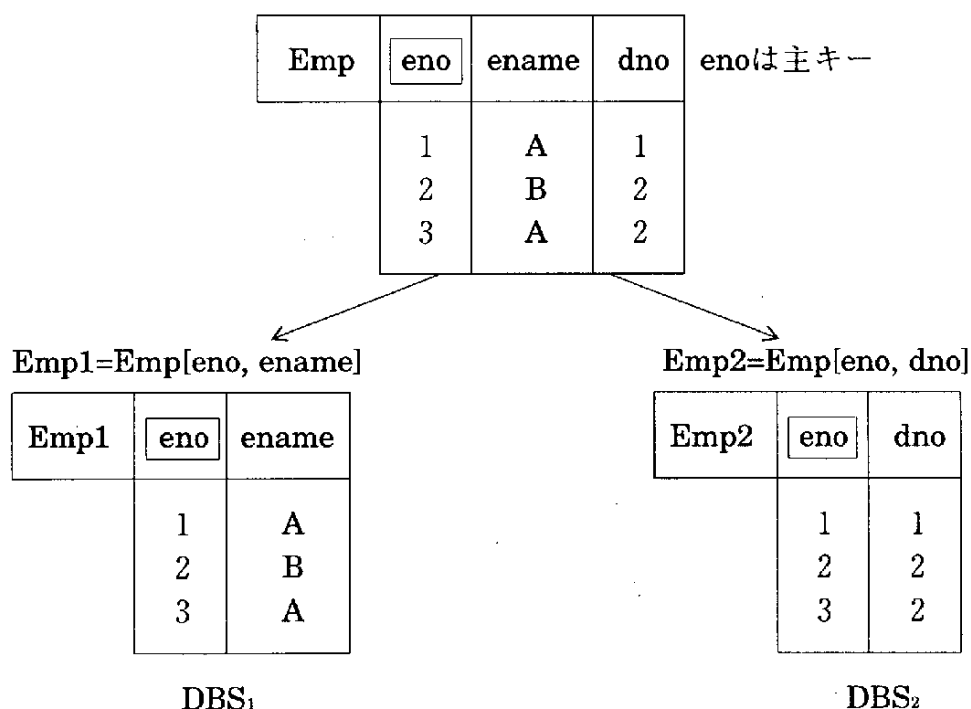


図3.3 垂直断片化

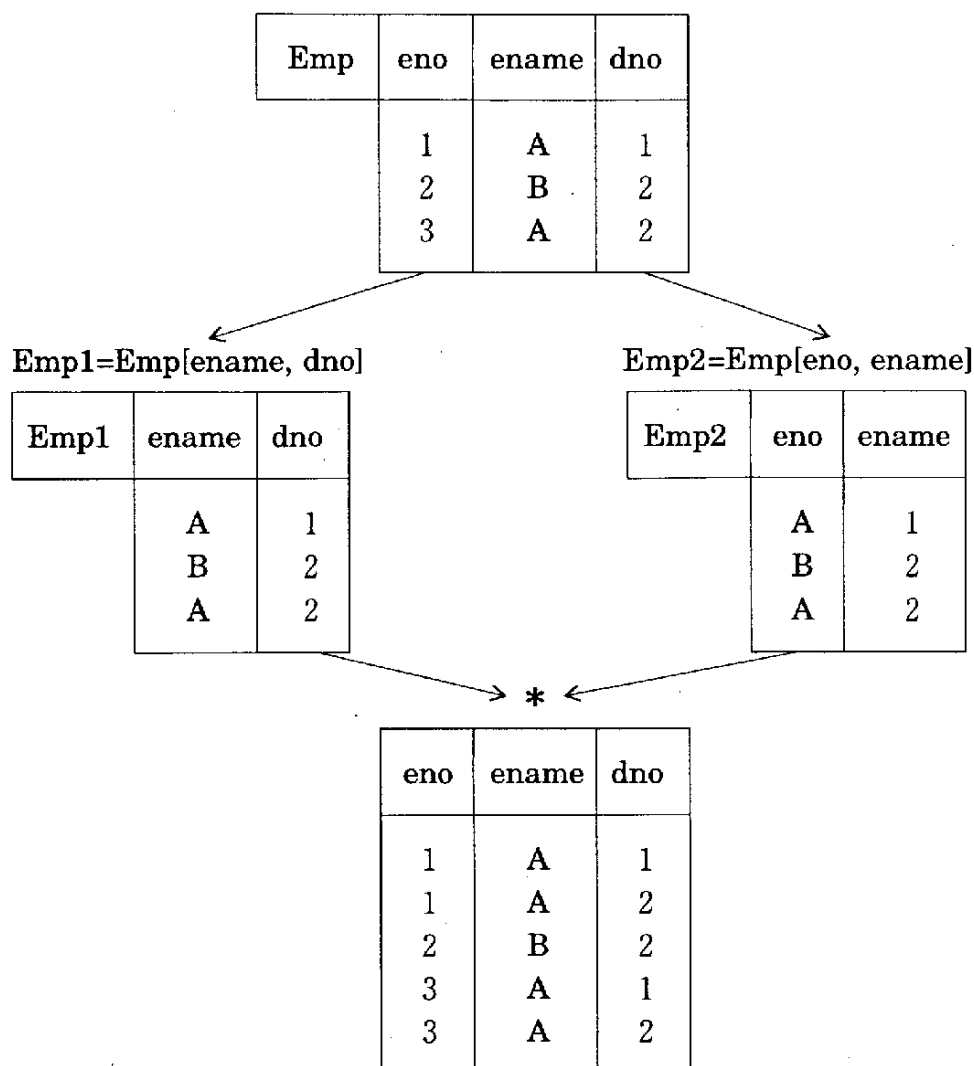


図3.4 正しくない垂直断片化

ここで、どのような属性ごとに断片とするかが問題となる。例えば、図3.3で、**ename**のみを検索する応用は、*DBS₁*内の断片**Emp1**を検索すればよい。**dno**のみを検索する応用は、*DBS₂*内の**Emp2**を検索する。このように、負荷を二つのデータベースサーバに分散できる利点がある。しかし、**ename**と**dno**を利用する検索に対しては、**Emp1**と**Emp2**の両方を結合することが必要である。例えば、*DBS₂*から**Emp2**を、*DBS₁*に送信する必要がある。**Emp2**の転送には通信時間がかかることから、断片化せずに**Emp**を検索した方が検索時間が短くなる。このように、垂直断片化するためには、各応用が複数の断片を利用しなくてよいように断片を定める必要がある。

3.2.2 水平断片化

論理的テーブルの射影を断片としていたが、行の集合である制約を断片とすることもできる。これを水平断片化とする。例えば、図2.2のEmpを、**dno=1**の行の集合 **Emp1=Emp[dno=1]**と**dno=2**の**Emp2=Emp[dno=2]**に分割し、各々を異なったデータベースサーバDBS1とDBS2に記憶する[図3.5]。**dno=1**の行についての検索はDBS1に対して行えて、**dno=2**についての検索はDBS2に対して行える。このように負荷を二つのサーバに分散できる。さらに、**Emp**全体を探索する場合に、**Emp1**と**Emp2**を並列に探索できる。

実際には、垂直と水平の断片化の組み合わせを断片として用いることができる。

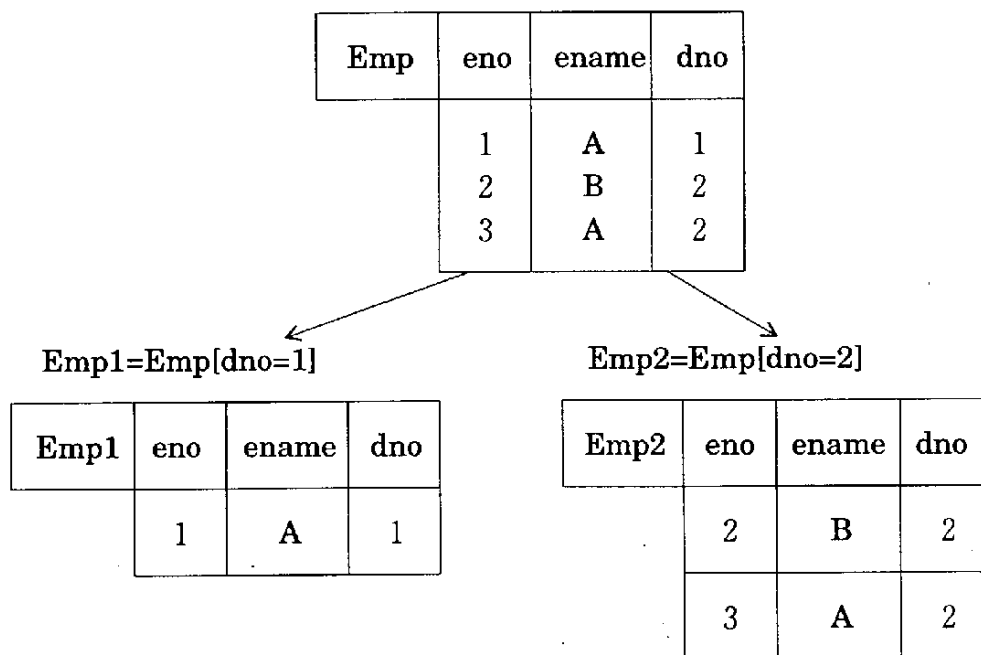


図3.5 水平断片化

3.2.3 冗長化

断片を複数のデータベースサーバに記憶することを断片の冗長化とする [図3.6]。このように、断片を冗長化したものを、断片の複製またはレプリカ(**replica**)とする。冗長化の利点として、以下がある。

[利点]

- (1) 利用者の必要とする断片が、自分の近くのデータベースサーバに存在するので通信

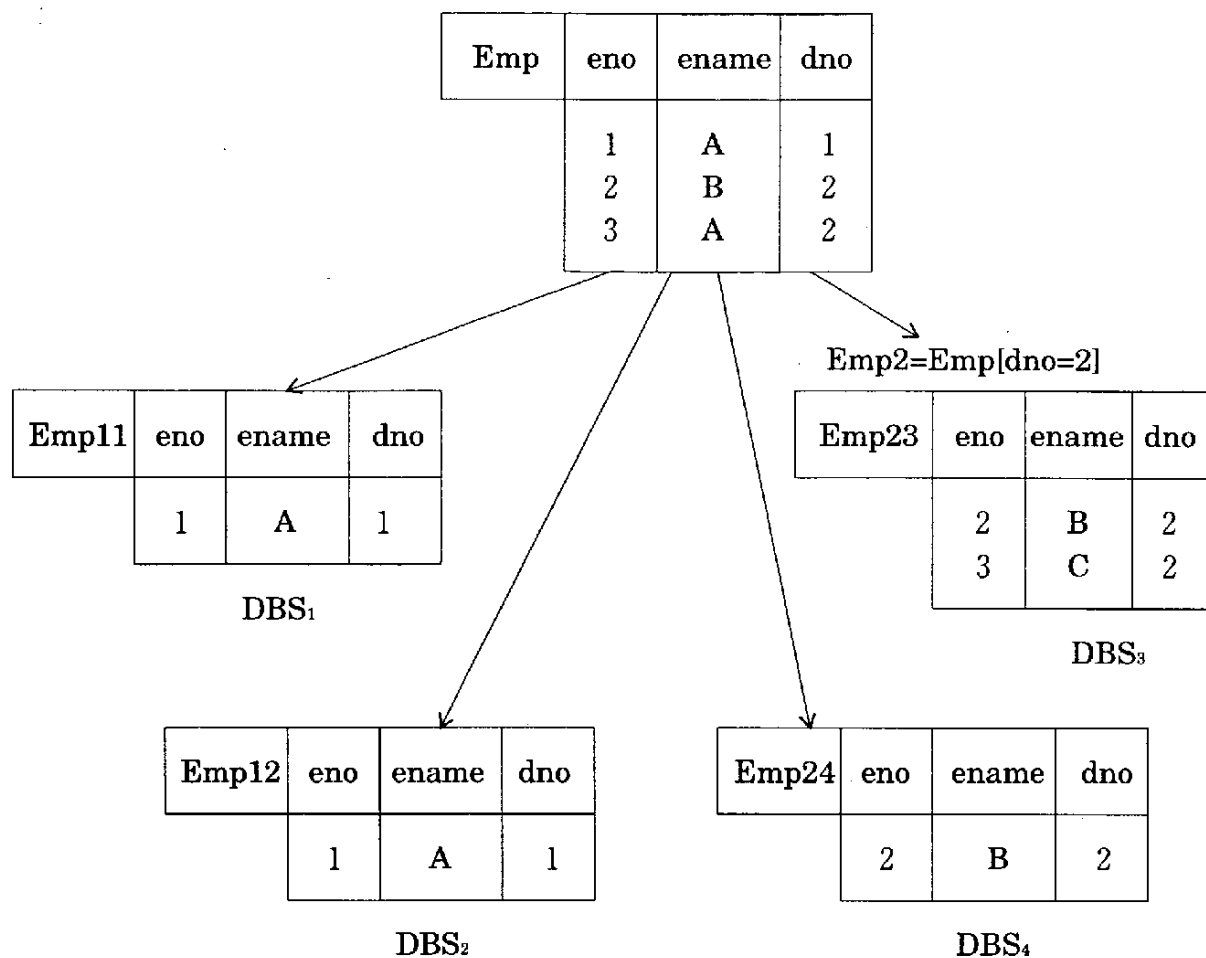


図3.6 冗長化

コストを低減できる。

- (2) 複数のデータベースサービスに負荷を分散できる。複数の検索要求があるとき、これらを複数のレプリカに振り分けることにより、複数のデータベースサービスに検索要求を分散できる。
- (3) データベースサーバの障害に対して、断片の他の複製がデータベースサーバに存在する信頼性と可用性を向上できる。

以上のように、テーブルの断片を多重化することにより、システムの可用性 (availability) を向上できるとともに、負荷分散によりシステムのスループットを向上できる。さらに、遠隔のコンピュータを操作する必要がなく、応答性も向上できる。

一方、冗長化を行うことに対して、以下の欠点がある。

【短所】

- (1) 更新演算に対して、レプリカ間の一致性を保つための制御が必要となる。
- (2) 複数のレプリカを記憶するための記憶量が増大する。

テーブルの値を更新する場合には、対応する断片の全レプリカを更新する必要がある。テーブルが複数のデータベースサーバに分散していない場合には、このテーブルに対してのみ更新を行えばよかった。しかし、複数のレプリカが存在していることから、これらのレプリカの全てを更新する必要がある。このためには、全レプリカの更新の同期をとるための通信が必要となる。

3.2.4 視野

断片化と冗長化は、論理テーブルを、どのように複数のデータベースサーバに格納するかの方法であった。即ち、トップダウンの設計である。これに対して、各データベースサーバ内にあるテーブルから、どのように論理テーブルを与えるかを考える。このとき、物理テーブルに対する視野として、論理テーブルを与える。例えば、データベースサーバ DBS_1 がテーブル **Emp** を持ち、 DBS_2 が **Dept** を持つ場合を考える。これに対して、図2.6に示すような視野 $DE(name, dname)$ を定義する。利用者は、**Emp** と **Dept** の所在と独立に、視野 DE に対して検索を行える。**Emp** と **Dept** に対する検索をどのように実行するか、例えば、**Dept** を DBS_2 から DBS_1 に送信して、 DBS_2 で **Emp** と結合を行うかは、システムが行う。 DE を用いて、更新を行うことには、問題がある。例えば、 DE に対して、 $name="A"$ で $dname="DDD"$ である行の更新を行うとする。このとき、**Emp** から $ename="A"$ の行を消去し、**Dept** から $dname="DDD"$ である行を消去すると、 DE から $dname="DDD"$ のすべての行が消えてしまう。同様に、 DE に $ename="A"$ で、 $dname="HHH"$ の行を追加すると、**Dept** に新しく $dbame="HHH"$ なる行を追加せねばならない。しかし、**Dept** の主キー dno の値が未定であるために、実体インテグリティ制約を乱すことから、**Dept** に追加を行えない。

4章 障 害

分散処理システムは、利用者に、ある仕様を満足するサービスを提供するシステムである。システムが仕様を満足するサービスを提供できなくなったとき、システムは故障(failure)したという。システムの障害をもたらすシステム要素の障害を、障害(fault)という。システム内の障害によって、利用者からみて正しくない状態が誤り(error)である。障害が生じて、サービスを提供できるシステムをフォールトトレラント(fault tolerant)であるという。フォールトトレランスとするための一つの方法としてシステムの構成要素を多重化することがある。これに対して、フォールト回避(fault avoidance)とは、高信頼な要素からシステムを構成することにより、フォールトが起きないようにすることである。システムの信頼性(reliability)とは、フォールトに対してどの程度システムが頑強であることを示している。システムを、フォールトトレランスと回避により、システムを高信頼とすることができる。信用性(dependability)とは、システムが利用者に対するサービスをどの位仕様通りに提供するかを示した概念である。信用性は、信頼性、可用性(availability)、安心性(safety)、安全性(security)の概念を包含した概念である。信頼性は、システムがどの位長くサービスを提供できるかを示す。可用性は、どの位直ちにシステムを利用できるかを示している。安心性は、システムがカストロフィをどの位起こさないかを示している。安全性は、利用権のない利用者が不正に情報を利用することをどの位防止できるかを示している。フォールトトレランスは、障害が起きたとしても、仕様を満足するサービスを提供する機能である。

[フォールトトレランス機能]

- (1) 誤り処理：誤りを除去することである。これには、誤りを検出し、除去する機能からなる。
- (2) フォールト処理：障害が、さらに誤りを起こさせないようにする機能である。これには、障害診断(diagnosis)、不活性化(passivation)がある。診断は、観察された誤りの原因を見つけることである。不活性化は、診断された障害が再度起きることを防止することである。

分散データベースシステムシステムを、フォールトトレランスにするための方法として、

システムの構成要素を多重化することがある。一つの要素が障害しても、残りの要素によりサービスを提供することにより、システムをフォールトトレランスとするものである。システムは、複数のコンピュータ $C_1, \dots, C_m (m > 1)$ から構成されたとする。あるプロセス p のレプリカ（複製）(replica)を $p_1, \dots, p_n$ とする。各レプリカ p_i を、 C_i で実行する。ここで、コンピュータの障害に対して、プロセス p の処理を継続することを考える。ここで、以下の仮定が成り立つとする。

[仮定] 各コンピュータの障害は、他のコンピュータと独立に起きる。

以上の仮定を満足する障害を、ハイゼンバグ(heisenbug)という。

分散データベースシステムは、通信ネットワークで結合されたクライアントとデータベースサーバから構成される。ここで、クライアントとサーバを総称してプロセスとする。このように、分散システムはプロセスと通信ネットワークから構成されることから、システムの障害としてプロセス障害とネットワーク障害がある。まず、プロセスの障害 [BARB93, FISC85] について考える。

[プロセスの障害]

(1) 停止(stop-by-failure)：プロセスが停止する障害である。

(2) オミッション(omission)障害：プロセスが、要求に対して、応答したり、しなかったりする障害である。

(3) コミッション(comission)障害：プロセスが、要求に対して、プロトコルに従わない応答を行う障害である。

プロセスの停止障害とは、プロセスが一定時間以上停止する障害である。停止した後復旧しない障害を永久停止とする。一定時間とは、後述するコミットメントプロトコルの処理時間以上の時間である。一過的(transient)障害とは、繰り返し発生しない障害であり、プロセスを再実行することにより復旧できる。停止して直ちに復旧する障害（間欠(intermittent)障害）は停止障害ではなく、オミッション障害である。

(2)と(3)は、プロセスが誤動作する障害であり、ビザンチン障害[GRAY86]という。いわゆるプログラムが「嘘」をつく障害である。

分散システムでは、複数のプロセス間での合意をとることが必要となる。合意をとるた

めの手順を合意プロトコルとする。後述するコミットメントプロトコルは、合意プロトコルの例である。合意プロトコルでは、システム内のプロセスの障害を考慮する必要がある。まず、プロセスの停止障害があるもとの合意プロセスについて考える。プロセスの停止は、以下のようにして検出される。

〔停止プロセスの検出〕 分散処理システムで、各プロセス p_i が、他のプロセス p_j が停止しているか、動作中であるかは、 p_j からメッセージが届くかどうかにより検出される。

一定時間以上の間、 p_j から一つもメッセージを受信しないならば、 p_i は、 p_j が停止しているものとする。この判断を p_i が行える前提には、 p_j が、 p_i が想定する速度以上で動作していることが前提となる。このとき、 p_i と p_j は、同期的(synchronous)であるとする。これは、以下のように、形式的に定義できる。

〔同期的プロセス〕 プロセス p_i がある演算を実行しているとき、他のプロセス p_j は、少なくとも一つの演算を実行しているとき、 p_i と p_j は、同期的である。

同期的プロセスについて、停止障害が起きるもとの合意を行うプロトコルの例として、コミットメント制御がある。これについては、後述する。例えば、コンピュータが過負荷となり、プロセス p_j の速度が低下したことにより、 p_i からのメッセージに対して一定時間以内に応答できないとする。 p_i は、 p_j が動作しているにもかかわらず停止しているものと判断してしまう。言い換えると、プロセスが同期的でない場合には、各プロセスは他のプロセスの停止をタイムアウトにより検出できない。同期的でないプロセスを、非同期的(asynchronous)とする。非同期的プロセスに対して以下の結論が示されている[FISC85]。

〔定理〕 プロセスが非同期的であるとき、一つでもプロセスが停止するならばプロセスで合意を行えない。

5章 トランザクション

分散データベースシステムでの処理の基本となるトランザクション(transaction)について考える。

5.1 トランザクションの定義

利用者が、複数のデータベースサーバ $DBS_1, \dots, DBS_n$ 内のデータを操作する問題を考える。各 DBS_i 内のデータベース DB_i をオブジェクトの集合とする。オブジェクトとは、各 DBS_i の基本的な操作単位である。リレーショナルデータベースシステムでは、テーブル、断片、行等がオブジェクトである。何をオブジェクトとするかは、各サーバにより決まる。こうしたオブジェクトの単位を粒度(*granularity*)という。クライアントから各 DBS_i にSQLが発行されるが、ここでは、 DBS_i の演算を抽象化して考える。各 DBS_i のオブジェクト x に対する基本操作としては、以下の演算がある。

[基本操作演算]

- (1) $r_i[x]$ =オブジェクト x の値の読み出す(read)演算。
- (2) $w_i[x]$ =オブジェクト x に値を書き込む(write)演算。

read演算 $r_i[x]$ は、 x に対するSQLのselect文に対応し、 $w_i[x]$ は、*update*、*insert*、*delete*文に対応する。クライアントは、これらの操作演算をサーバに対して発行する。例として、銀行AからBに送金することを考える。送金を行う場合には、Aの口座とBの口座の二つの更新を行う必要がある。両方の更新を正しく行えたときのみ、送金を完了できる。いずれか一方の更新を行えないときには、送金を行えない。このような複数の操作演算の実行系列をトランザクションという。

以上の基本操作演算に加えて、トランザクションの終了を行うための演算として以下のものがある。

[コミットとアボート演算]

- (1) $c(commi)$: トランザクションを正常終了(コミット)させる演算である。

(2) $a(\text{abort})$: トランザクションを異常終了(アボート)させる演算である。

アボート a は、この実行する前までに実行された更新演算を無効とする演算である。これに対して、コミット c は、これまでに実行された更新演算を完了する演算である。例えば、 A の口座の更新後に、 B の口座の更新を行えないことがわかったとき、アボート a を実行することにより、 A の口座の更新を無効とできる。 c と a は、おのこのSQLの *commit-work* と *abort-work* 文の実行を示す。

以上をまとめると、トランザクション T とは、分散データベースシステム内のオブジェクトに対する一つ以上の基本演算の系列の実行状態である。但し、以下の性質を充足するものである。

[トランザクションの性質(ACID)]

(1) 原子性(atomicity): T 内の全操作演算が実行されるか、全くされないかのいずれかである。

(2) 一貫性(consistency): T を単独で実行させたときに、データベースのインテグリティ制約を保つ。

(3) 独立性(isolation): T の中間結果を他のトランザクションは見れない。

(4) 耐久性(durability): T が正常終了した時、更新結果はデータベースから消失しない。
次に、トランザクションの例について考える。

[例] トランザクションの例を考える。3つのデータベースサーバ DBS_1 、 DBS_2 、 DBS_3 があるとする。各データベースサーバ DBS_i は、各個人の名前(name)とその人の口座の残金(account)を示すテーブル $Bank(\text{name}, \text{account})$ を持つ。ここで、ある個人 A が、 DBS_1 と DBS_2 内の自分の口座から、おのこの x と y 円を引き出して、その合計額($=x+y$)を DBS_3 内の B 氏の口座に振り込むとする。これは、SQLにより以下のように書ける。以下で、 $DBS_i.Bank$ は、 DBS_i 内のテーブル $Bank$ を示す。

```
select account into t1 from DBS1.Bank where name ="A";
```

```
if t1.account-x<0, then abort-work; ... (1)
```

```
update DBS1.Bank set account=account-x where name="A";
```

```
select account into t2 from DBS2.Bank where name="A";
```

```
if t2.account-y<0, then abort-work; ... (2)
```

```

update DBS2.Bank set account=account-y where name="A";
update DBS3.Bank set account=account+x+y where name ="B";
commit-work; ... (3)

```

ここで、*abort-work*はトランザクションのアボート（異常終了）を行わせる演算であり、これまでの更新結果を無効とする。例えば、この例では、(2)で、データベースサーバDBS₁のBankを更新した後に、DBS₂のBankが残高不足のために更新出来なかった場合に、DBS₁の更新を無効としている。*commit-work*はトランザクションのコミット（正常終了）を示す。正常終了すると更新が完了する。このプログラムが実行する基本演算の系列がトランザクション*T*である。正常終了した場合のトランザクション*T*を以下に示す[図5.1]。ここで、 a_1 と a_2 は、おのこのDBS₁とDBS₂内のAについてのオブジェクト（行）を示す。また、 b はDBS₃内のBについてのオブジェクトを示す。

$r_1[a_1]; w_1[a_1]; r_2[a_2]; w_2[a_2]; r_2[b]; w_3[b]; c;$

すべての操作演算を正しく行えるときにだけ、*T*は正常終了する。これが原子性である。

トランザクション*T*が正常終了したときには、データベースサーバDBS₁、DBS₂、DBS₃のテーブルBank内の値は更新される。ここで、データベースサーバが障害を起こしても、復旧した場合に、Bank内のaccountの値は更新された値でなければならない。この値を変更するためには、変更を行うためのトランザクションを実行する必要がある。これ

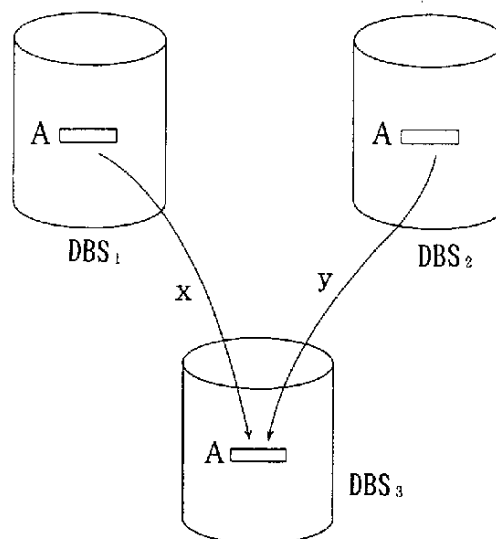


図5.1 トランザクション

が耐久性である。さらに、 DBS_1 と DBS_2 の更新後で、 B の更新を行う前に、他のトランザクションが A の口座を操作することができないことが独立性である。 T を単独で実行した場合 A の口座の値を B に移しているので、データベース内の残金の合計は変わらない、即ちインテグリティ制約が保たれている。これが、一貫性である。

トランザクション T は、クライアント C で実行される。 T 内の各演算は、 C からデータベースサーバ DBS_1 、 DBS_2 、 DBS_3 に送信され、各データベースサーバで実行される。各 DBS_i で実行される T の演算の系列を副トランザクション T_i とする。例えば、図5.1で、 T の副トランザクションとして、以下がある。

$$T_1=r_1[a_1];w_1[a_1];$$

$$T_2=r_2[a_2];w_2[a_2];$$

$$T_3=r_3[b];w_3[b];$$

5.2 実行履歴

$m(\geq 1)$ 個のトランザクション $T_1, \dots, T_m$ が、 $n(\geq 1)$ 個のデータベースサーバ $DBS_1, \dots, DBS_n$ 内のオブジェクトを操作する場合について考える。各 DBS_i は、複数のトランザクションからの演算が一つずつ直列に実行される。この実行の履歴を、 DBS_i のローカル履歴 L_i とする。ローカル履歴 $L_1, \dots, L_n$ の集合を $T_1, \dots, T_m$ の履歴 L とする。

〔例〕データベースサーバ DBS_1 がオブジェクト x と y を持ち、 DBS_2 が z を持つとする〔図5.2〕。次の二つのトランザクション T_1 と T_2 が DBS_1 と DBS_2 に対して実行されるとする。

$$T_1:r_{11}[x];w_{12}[z];r_{11}[y];c;$$

$$T_2:r_{22}[z];w_{21}[x];r_{21}[y];c;$$

次に、 DBS_1 と DBS_2 に対するローカル履歴 L_1 と L_2 の例を以下に示す。

$$L_1:r_{11}[x];w_{21}[x];r_{11}[y];r_{21}[y];$$

$$L_2:r_{22}[z];w_{12}[z];$$

これらは、 DBS_1 と DBS_2 で、 T_1 と T_2 をインタリーブ実行した履歴である。

任意の二つの演算 op_1 と op_2 に対して、 op_2 が op_1 の終了後に実行されるとき、 $op_1 < op_2$ と書く。このとき、 op_1 は、 op_2 に先行するという。

トランザクション集合 $T=\{T_1, \dots, T_m\}$ に対する履歴 L の性質について考える。

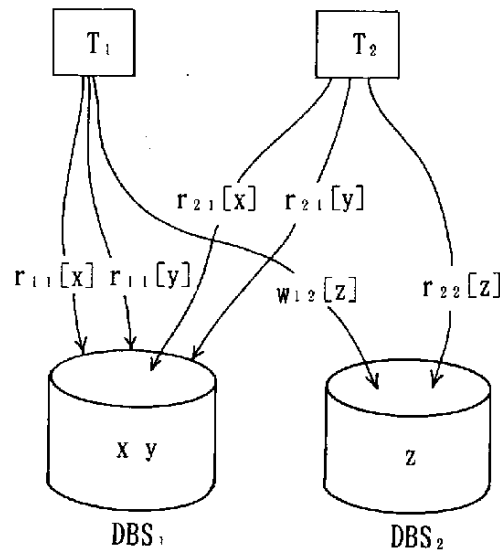


図5.2 履歴

〔正しい履歴〕 インテグリティ制約を充足しているデータベースを正しいとする。正しいデータベースに対して、履歴 L を実行させた結果、正しいデータベースをもたらすならば、履歴 L は正しい。

まず、トランザクション間のコミット/アボートの関係について考える。ここで、 x を DBS_j 内のオブジェクトとする。また、 $r_j[x]$ 、 $w_j[x]$ 、 a_j 、 c_j を、各々トランザクション T_j の x に対する*read*、*write*、*abort*、*commit*演算とする。

〔トランザクション間の読みだし関係〕 L をトランザクション $T_1, \dots, T_m$ の履歴とする。 T_j が T_k からオブジェクト x を読むとは以下の条件を満足することである〔図5.3〕。

- (1) L_j 内で、 $w_k[x]$ が $r_j[x]$ に先行する。即ち、 $w_k[x]$ が $r_j[x]$ より先に DBS_j で実行される。
- (2) $r_j[x]$ の前に、アボート a_k は実行されていない。
- (3) $w_k[x]$ の後で、 $r_j[x]$ に先行して $w_h[x]$ が存在するならば、 $r_j[x]$ の実行前に T_h はアボートしている。

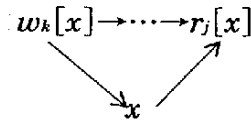


図5.3 読みだし関係

即ち、トランザクション T_k が書いた x の値を、 T_j が読んでいるとき、 T_j は T_k が x を読むという。

次に履歴の性質について考える。

[復旧可能履歴] L を履歴とする。このとき、 L 内で、トランザクション T_j が T_k からオブジェクト x を読み、 c_k を L が含んでいるならば、 $c_k < c_j$ であるとき、 L は復旧可能(recoverable)という。

$L_1: \dots w_k[x] \dots c_k \dots r_j[x] \dots c_j \dots$

$L_2: \dots w_k[x] \dots r_j[x] \dots c_j \dots a_k \dots$

$L_3: \dots w_k[x] \dots r_j[x] \dots c_k \dots c_j \dots$

L_1 では、トランザクション T_j は、 T_k が書いた値をコミット後に読んでいるので復旧可能な履歴である[図5.4(a)]。これに対して、 L_2 と L_3 では、 T_j は、 T_k がコミットする前に T_k が書いた値を読んでいる。 L_2 では、 T_j がコミットした後に、 T_k がアボートしている。 T_j は、すでにコミットしているのでアボートできない。結果として、 T_j は存在しない値を読んだことになってしまう。従って、このような履歴では、トランザクションのアボートに対して復旧を行えなく、復旧不可能という[図5.4(b)]。

履歴 L_3 では、復旧可能であるが、トランザクション T_j は T_k の書いた値を読んでいる。したがって T_k がアボートすると、 T_j もアボートする必要がある。一つのトランザクションのアボートにより、他のトランザクションもアボートすることを連鎖的アボートという。

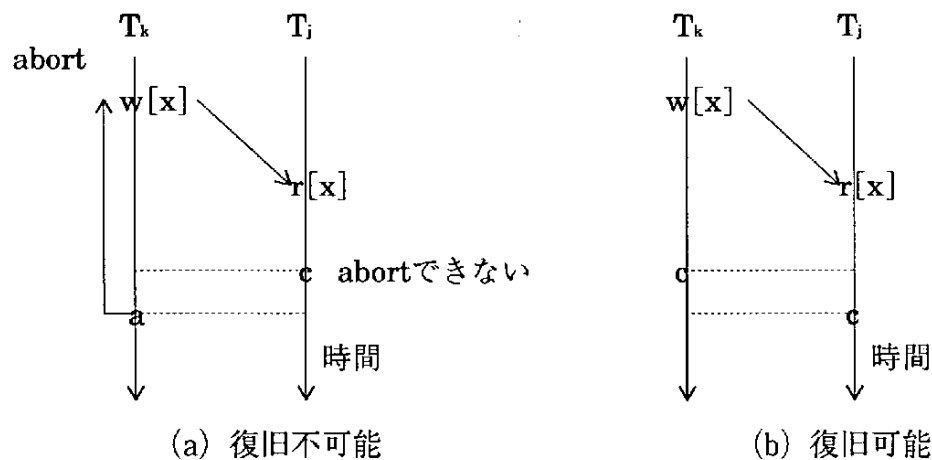


図5.4 復旧可能性

〔連鎖的なアボートを起こさない履歴〕 トランザクション T_j が T_k から履歴 L 内で x を読むならば、 $c_k < r_j[x]$ であるとき L は連鎖的なアボート(cascading abort)を起こさないという。

$L_4: \dots w_k[x] \dots r_j[x] \dots a_k \dots c_j \dots$

$L_5: \dots w_k[x] \dots c_k \dots r_j[x] \dots c_j \dots$

履歴 L_4 は、 $r_j[x]$ の後にトランザクション T_k がアボートしたならば、 T_j もアボートしなければならないので連鎖的なアボートを起こす履歴である。 L_5 は、連鎖的なアボートを起こさない。このように、連鎖的なアボートを起こさない履歴は、復旧可能である〔図5.5〕。

ここまでは、トランザクション間での w と r の関係を考えてきた。次に、トランザクション間での w 演算の間の関係について考える。

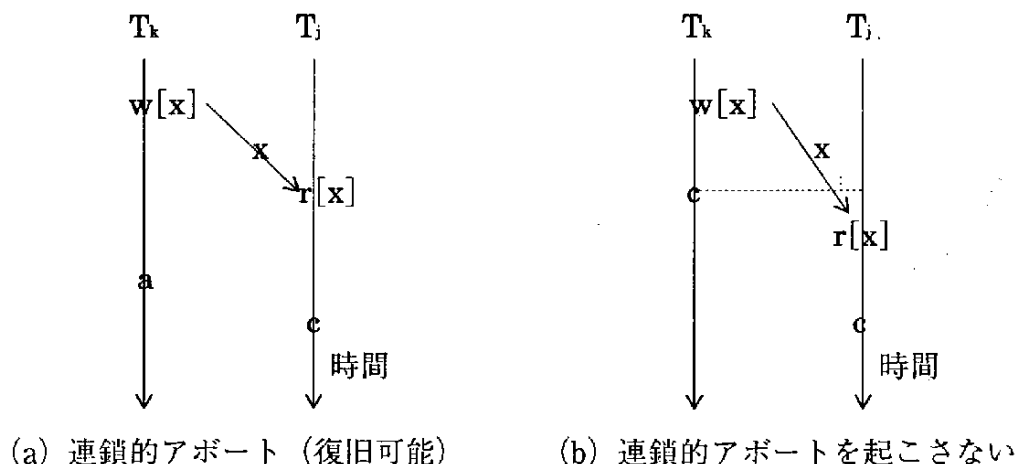


図5.5 連鎖的アボート

[厳格な履歴] $w_k[x] < op_j[x]$ ならば、 $a_k < op_j[x]$ 又は $c_k < op_j[x]$ (ここで、 $op_j[x]$ は $r_j[x]$ 又は $w_j[x]$ である) であるとき、履歴 L は厳格(strict)であるという。

$L_6: \dots w_j[x] \dots w_k[x] \dots c_k \dots r_j[x] \dots c_j \dots$

$L_7: \dots w_k[x] \dots c_k \dots w_j[x] \dots r_j[x] \dots c_j \dots$

履歴 L_6 は、 T_j がコミットする前に x を書いているので、厳格な履歴ではない。 L_7 は、厳格な履歴である。例からわかるように、厳格な履歴は復旧可能であり、かつ連鎖的なアボートを起こさない[図5.6]。

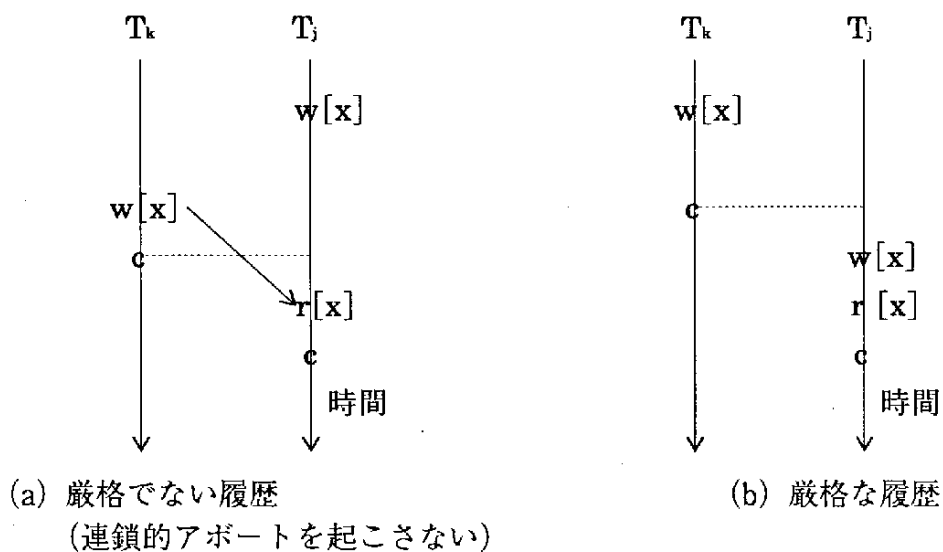


図5.6 厳格履歴

以上から、厳格な履歴は連鎖的なアボートを起こさず、連鎖的なアボートを起こさない履歴は復旧可能であることがわかる。しかし、この逆は成り立たない。このことは、厳格な履歴は、連鎖的なアボートを起こさない履歴よりも制約が厳しく、トランザクションの並行度が減少する。同様に、連鎖的なアボートを起こさない履歴は、復旧可能な履歴よりも、並行度が減少する。

5.3 正しくない履歴

$op_u[x]$ をトランザクション T_i の演算で、 DBS_i 内のオブジェクト x に対する演算とする。 op は $r(read)$ または $w(write)$ である。 x に対する $op_u[x]$ と $op_w[x]$ を考える。

【競合演算】二つの演算の実行順序により演算結果が異なるとき、 $op_j[x]$ と $op_k[x]$ は競合するという。

例えば、 $r[x]$ の次に $w[x]$ を実行した場合と、 $w[x]$ の後に $r[x]$ を実行した場合では、 x の値は同じでも $r[x]$ する値が異なる。したがって、 $r[x]$ と $w[x]$ は競合する。同様に、 $w[x]$ と $w[x]$ も競合する。競合する演算の実行順序を各サーバでどのような順序とするかが重要な問題となる。

【例】二つのデータベースサーバ DBS_1 と DBS_2 が、おのおのオブジェクト a と b を持つとする。インテグリティ制約として、“ $a=b$ ”という関係があるとする。トランザクション T_1 は、 a と b の値に1加え、 T_2 は1減じるとする。

$T_1:r_{11}[a];/*a=a+1*/w_{11}[a];r_{12}[b];/*b=b+1*/w_{12}[b];c;$

$T_2:r_{21}[a];/*a=a*2*/w_{21}[a];r_{22}[b];/*b=b*2*/w_{22}[b];c;$

明らかに、トランザクション T_1 と T_2 を、おのおの単独で実行した場合には、 $a=b$ である。しかし、 DBS_1 と DBS_2 のローカル履歴 L_1 と L_2 〔図5.7〕を考えると、 $a \neq b$ となる。これは、トランザクションが、他のトランザクションの中間結果を読むことによりおきるので、汚い読みだし(dirty read)という。従って、 L_1 と L_2 から成る履歴 L は、正しくない。

$T_1:r[a];/*a=a+1*/w[a];r[b];/*b=b+1*/w[b];c;$

$T_2:r[a];/*a=a+2*/w[a];r[b];/*b=b+2*/w[b];c;$

T_1	T_2	a	b
$r[a];$		10	10
$/*a=a+1*/$			
$w[a];$		11	10
	$r[a];$	11	10
	$/*a=a+2*/$		
	$w[a];$	22	10
	$r[b];$	22	10
	$/*b=b+2*/$		
	$w[b];$	22	20
$r[b];$		22	20
$/*b=b+1*/$			
$w[b];$		22	21

時間
↓

図5.7 汚い読み出し

例] 次に、以下の二つのトランザクション T_1 と T_2 を考える。

$T_1: w_{11}[a]; /*a=20*/w_{12}[b]; /*b=20*/c;$

$T_2: w_{21}[a]; /*a=30*/w_{22}[b]; /*b=30*/c;$

このローカル履歴を図5.8に示す。この結果、 $a \leq b$ となってしまう。これは、競合するwrite演算の順序が問題である。これを、紛失更新(lost update)という。

$T_1: r[a]; /*a=a+1*/w[a]; c;$

$T_2: r[a]; /*a=a+1*/w[a]; c;$

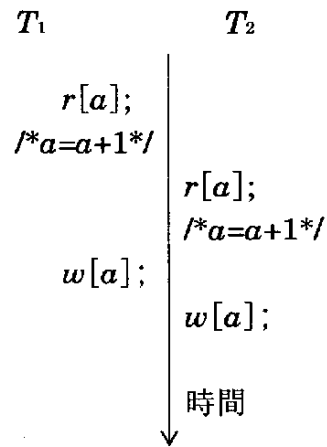


図5.8 紛失更新

以上の例からわかるように、複数のトランザクションの競合する演算の実行順序を制御する必要がある。このための制御を、同時実行制御という。

6 章 同時実行制御

複数のトランザクションが、複数のデータベースサーバ内のオブジェクトを操作する問題について考える。このとき、各データベースサーバ内のデータベースがインテグリティ制約を保った状態に保つとともに、システム全体のスループットを向上させる必要がある。システムのスループットを向上させるためには、複数のトランザクションがデータベースサーバでインタリーブ実行することと、複数のデータベースサーバで並列実行する方法がある。しかし、任意にインタリーブ実行し、並列実行すると、データベースのインテグリティ制約が保たれなくなる場合がある。分散システムのスループットを向上させながら、データベースのインテグリティ制約を保つための制御が、同時実行制御[BERN87]である。

6.1 直列可能性

トランザクション $T_1, \dots, T_m$ が、データベースサーバ $DBS_1, \dots, DBS_n$ 内のオブジェクトを更新する場合を考える。複数のトランザクションが複数のサーバにより並列に実行されることの正しさの根拠として、直列可能性(serializability)[BERN87, ESWA76]がある。あるトランザクションが完了してから、次のトランザクションが一つずつ実行される場合、これを直列実行とする。その履歴は、直列であるとする。各トランザクションが完了したとき、分散データベースシステム内のデータベースを正しい状態、即ち、インテグリティ制約の保たれた状態に保つ。このことから、複数のトランザクションが直列に実行された場合も、データベースを正しい状態に保つ。複数のデータベースサーバで、トランザクションが並列実行された結果が、ある直列実行した結果と同じであることをもって、この並列実行の正しさとしている。履歴 L に対するトランザクション間の順序関係 $\ll_L$ を定義する。

【トランザクション間の順序関係】二つのトランザクション T_i と T_k に対して以下が成り立つとき、 $T_i \ll_L T_k$ とする。即ち、ある履歴 L_h で、

- (1) T_i 演算 op_{jh} と T_k の演算 op_{kh} を含み、 op_{jh} と op_{kh} が競合し、
- (2) op_{jh} が op_{kh} よりも、 DBS_h で先行して実行される。

例えば、図5.7で、 DBS_1 内のオブジェクト a については、 T_1 と T_2 の競合演算は、 T_1 のものが T_2 のものより、先に操作しているので、 $T_1 \ll_L T_2$ である。順序関係 $\ll_L$ を用いて、 $T_1, \dots, T_n$ の実行を示す履歴 L が直列可能であることを定義する。

〔直列可能性〕 トランザクション集合の履歴 L について、 $\ll_L$ が全順序関係ならば、 L は直列可能である。

〔例〕 図5.2で示した例を考える。データベースサーバ DBS_1 がオブジェクト x と y を持ち、 DBS_2 がオブジェクト z を持つとする〔図6.1〕。 DBS_1 と DBS_2 に対する次の二つのトランザクション T_1 と T_2 について考える。

$T_1: r_{11}[x]; w_{12}[z]; r_{11}[y]; c$

$T_2: r_{22}[z]; w_{21}[x]; r_{21}[y]; c$

DBS_1 と DBS_2 に対するローカル履歴 L_1 と L_2 の例を以下に示す。

$L_1: r_{11}[x]; w_{21}[x]; r_{11}[y]; r_{21}[y];$

$L_2: r_{22}[z]; w_{12}[z];$

副履歴 L_1 と L_2 から構成される履歴 L は、直列可能ではない。何故ならば、データベースサーバ DBS_1 で、オブジェクト x に対する競合演算は、トランザクション T_1 の $r_{11}[x]$ が T_2 の $w_{21}[x]$ より前に実行されている。しかし、 DBS_2 では、逆に、 T_2 の $r_{22}[z]$ が T_1 の $w_{12}[z]$ に先行しているからである。即ち、 $T_1 \ll_L T_2$ で $T_2 \ll_L T_1$ である。しかし、以下に示すローカル履歴 M_1 と M_2 の履歴 M は直列可能である。 DBS_1 の x については、 T_1 の $r_{11}[x]$ が T_2 の $w_{21}[x]$ に先行している。 DBS_2 のオブジェクト z については、 T_1 の $w_{12}[z]$ が T_2 の $r_{22}[z]$ に先行している。一方、オブジェクト y については、 T_2 の $r_{21}[y]$ が T_1 の $r_{11}[y]$ に先行しているが、これらは競合しない。従って、 DBS_1 と DBS_2 における競合演算は、 T_1 が T_2 よりも先行している($T_1 \ll_M T_2$)。このことは、 T_1 を実行した後に、 T_2 を直列に実行した場合と同じ結果が得られる。

$M_1: r_{11}[x]; w_{21}[x]; r_{21}[y]; r_{11}[y];$

$M_2: w_{12}[z]; r_{22}[z];$

直列可能性の判定のために、履歴 L に対する以下の順序グラフ (precedence graph) を考える。

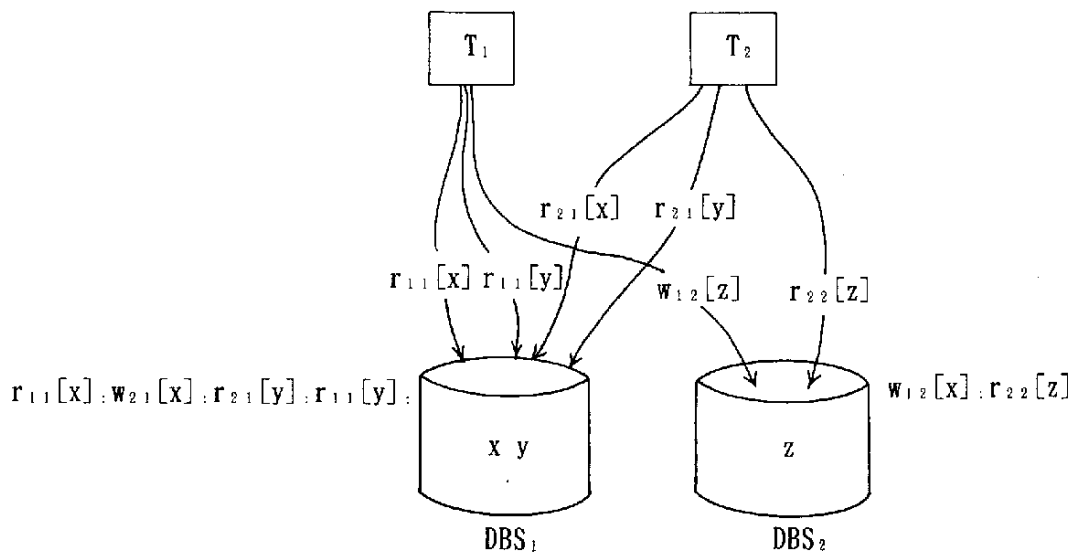


図6.1 直列可能性

〔順序グラフ〕履歴 L に対して、以下の手順により得られる有向グラフ G を L の順序グラフとする。

- (1) L 内の各トランザクション T_i に対して、節点 T_i を設ける。
- (2) 任意の T_i と T_j に対して、 T_i の演算 op_i と T_j の演算 op_j が競合し、 L 内で、 op_i が op_j より先行して実行されている($op_i < op_j$)とき、節点 T_i から T_j に有向辺 $T_i \rightarrow T_j$ を設ける。

例えば、図6.1に示した履歴 L と M の順序グラフを図6.2に示す。図からわかるように、履歴が直列可能であるときは、その順序グラフは、巡回閉路を含まない。

〔直列可能性〕履歴 L の順序グラフ G が巡回閉路を含まないとき、 L は直列可能である。

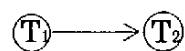


図6.2 順序グラフ

履歴の順序グラフを用いると、二つの履歴が同値であることを以下のように定義できる。

〔履歴の同値性〕 二つの履歴 L_1 と L_2 の順序グラフが同じとき、 L_1 と L_2 は同値である($L_1 \equiv L_2$)。

6.2 二相ロック

6.2.1 二相ロックプロトコル

任意のトランザクション集合 $T=\{T_1, \dots, T_m\}$ ($m \geq 1$)の履歴を直列可能とする代表的方法としては、二相ロック[ESWA76]がある。オブジェクト x に対して、新しく以下の演算を導入する。

〔ロック演算〕

- (1) $l[x]$ =オブジェクト x をロックする演算。
- (2) $u[x]$ = x のロックを解放する演算。

これらの演算は、以下の規則に従って、適用される。

〔ロック規則〕

- (1) 各トランザクション T_i は、オブジェクト x を操作する前に、ロック演算 $l[x]$ により x のロックを行う。他のトランザクション T_j が x をロックしている場合は、 T_i は T_j のロックの解放 $u[x]$ を待つ。
- (2) トランザクション T_i は、終了前に、 T_i が獲得した全ロックの解放を行う。

トランザクションが、ロック規則に従ってオブジェクトをロックしても、得られる履歴は、直列可能とはならない。例を考える。

〔例〕 図5.7の例を考える。ここで、トランザクション T_1 と T_2 は、以下のようにオブジェクト a と b をロックするとする。

$T_1: l_{11}[a]; r_{11}[a]; w_{11}[a]; u_{11}[a]; l_{12}[b]; r_{12}[b]; w_{12}[b]; u_{12}[b]; c;$

$T_2: l_{21}[a]; r_{21}[a]; l_{21}[a]; w_{21}[a]; r_{22}[b]; w_{22}[b]; u_{21}[a]; u_{22}[b]; c;$

この履歴 K を図6.3に示す。図からわかるように、ロックを行っても、図5.7と同じ履歴となり、直列可能ではない。

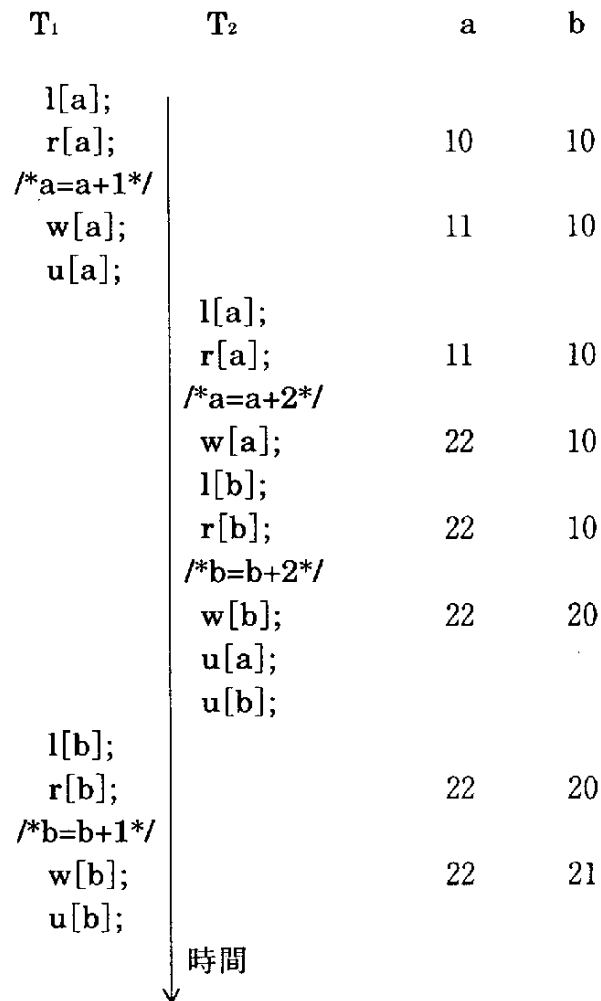


図6.3 ロック演算と直列可能性

履歴を直列可能するためのロック方法として、二相ロック方式がある。これを以下に示す。まず、オブジェクトが複数のデータベースサーバに冗長に存在しない場合を考える。ここで、 T_i をトランザクションとする。

[二相ロック方式]

- (1) T_i 内の各演算 op_{ij} に対して、 DBS_j 内のオブジェクト x をロック($L_{ij}[x]$)する。
- (2) T_i がロックの解放演算を行ったならば、これ以後ロック演算を行わない。

図5.7の二層ロックトランザクションの実例を図6.4に示す。

T ₁	T ₂	a	b
l[a];		10	10
r[a];		10	10
w[a];		10	10
l[b];		11	10
u[a];		11	10
	l[a];	11	10
	r[a];	11	10
		11	10
	w[a];	22	10
r[b];		22	10
w[b];		22	11
u[b];		22	11
	l[b];	22	11
	r[b];	22	11
	w[b];	22	22
	u[a];	22	22
	u[b];	22	22
	時間 ↓		

図6.4 2PLトランザクションによるログ

二相ロックトランザクションの例を以下に示す。

[例] 図6.1に示すデータベースサーバに対する以下の二つのトランザクション T_1 と T_2 を考える。これらは、二相ロック形式である。

$T_1: l_{11}[x]; r_{11}[x]; l_{12}[z]; w_{12}[z]; l_{11}[y]; u_{12}[z]; r_{11}[y]; u_{11}[y]; u_{11}[x]; c;$

$T_2: l_{22}[z]; r_{22}[z]; l_{21}[x]; w_{21}[x]; l_{21}[y]; r_{21}[y]; u_{21}[y]; u_{21}[x]; u_{22}[z]; c;$

トランザクション T_1 と T_2 では、ロックの解放演算が実行された後にロック演算が実行されていない。したがって、 T_1 と T_2 は、二相ロックトランザクションである。次のトランザクション T_3 は二相ロック形式ではない。何故ならば、オブジェクト x のロックを解放演算 $u_{11}[x]$ により解放した後に、オブジェクト z と y のロックが行われているからである。

$T_3: l_{11}[x]; r_{11}[x]; l_{12}[z]; u_{11}[x]; w_{12}[z]; l_{11}[y]; u_{12}[y]; r_{11}[y]; u_{11}[y]; c;$

各トランザクション T_i は、複数のデータベースサーバに演算を発行し、実行される。こ

ここで、 DBS_j でのトランザクション T_i の操作演算の系列を、 T_i の副トランザクション T_{ij} とする。 T_1 と T_2 の副トランザクションは、以下のようになる。

$$T_{11}:l_{11}[x];r_{11}[x];l_{11}[y];r_{11}[y];u_{11}[x];u_{11}[y];c;$$

$$T_{12}:l_{12}[z];w_{12}[z];u_{12}[z];c;$$

$$T_{21}:l_{21}[x];w_{21}[x];l_{21}[y];r_{21}[y];u_{21}[y];u_{21}[x];c;$$

$$T_{22}:l_{22}[z];r_{22}[z];u_{22}[z];c;$$

これらの各副トランザクション T_{ij} は、二相ロック形式である($i, j=1, 2$)。副トランザクション T_{11} と T_{21} の DBS_1 でのインタリーブ実行を示すローカル履歴 N_1 と、 T_{12} と T_{22} の DBS_2 でのインタリーブ実行を示すローカル履歴 N_2 を以下に示す。

$$N_1:l_{11}[x];r_{11}[x];l_{11}[y];u_{11}[x];l_{21}[x];u_{11}[y];w_{21}[x];l_{21}[y];r_{21}[y];u_{21}[x];u_{21}[y];$$

$$N_2:l_{12}[z];w_{12}[z];u_{12}[z];l_{22}[z];r_{22}[z];u_{22}[z];$$

副履歴 N_1 と N_2 から成る履歴 N では、どの競合演算もトランザクション T_1 のものが T_2 よりも先に実行されている。即ち、 $T_1 \ll T_2$ である。従って、 T_1 の実行後に、 T_2 を実行した場合を示す以下の直列履歴と同値である。

$$S_1:l_{11}[x];r_{11}[x];l_{11}[y];r_{11}[y];u_{11}[x];u_{11}[y];l_{21}[x];w_{21}[x];l_{21}[y];r_{21}[y];u_{21}[y];u_{21}[x];$$

$$S_2:l_{12}[z];w_{12}[z];u_{12}[z];l_{22}[z];r_{22}[z];u_{22}[z];$$

二相ロックトランザクションについて、以下の結果が得られている。

[定理] トランザクション集合 T 内の各トランザクションどの副トランザクションも二相ロック形式ならば、 T から得られる履歴は直列可能である。

即ち、どのトランザクションも二相ロック形式ならば、これらのトランザクションを複数のデータベースサーバで実行した結果は正しい。二相ロック方式では、ロックの解放をいつ行うかが問題となる。オブジェクト x のロックの解放をトランザクション T_i の終了前に行うと、他のトランザクション T_j が x を T_i の終了前に操作できる。 T_i がアボートした場合には、 T_j もアボートされる必要がある。こうした連鎖的アボートを防ぐための方法として、トランザクションの終了時に獲得した全てのロックを一括して解放する方法がある。この方法を行うと、履歴を厳格なものとなる。

[厳格な二相ロック方式]

- (1) T_i 内の各演算 $op_{ij}[x]$ に対して、データベースサーバ DBS_j 内のオブジェクト x をロック($l_{ij}[x]$)する。
- (2) T_i がコミットまたはアボートする時に、 T_i のロックをすべて解放($u_{ij}[x]$)する。

6.2.2 ロックのモード

オブジェクト x を、あるトランザクション T_i が $read$ する場合を考える。他のトランザクション T_j も x を $read$ するとき、 T_i が x をロックしているために T_j が x を利用できない。しかし、 $read$ 演算はどのような順序で実行されても結果は同じである。このために、各演算 op に対して、ロックモード $mode(op)$ を考える。

[ロックのモード] オブジェクト x に対する二つの演算 op_1 と op_2 をどのように実行しても結果が同じであるとき、 op_1 と op_2 は交換可能(commutable)とする。交換可能な演算 op_1 と op_2 のロックモード $mode(op_1)$ と $mode(op_2)$ は伴立である(compatible)とする。

演算 op_1 と伴立なモードでオブジェクト x が他の演算 op_2 でロックされているときには、 op_1 は x を操作できる。そうでないときには、 op_2 によるロックの解放を待たねばならない。例えば、 $read$ 演算のロックモード $Rlock$ は互いに伴立であるが、 $write$ 演算のロックモード $Wlock$ は、 $Wlock$ とも $Rlock$ とも伴立ではない。これらのロックモード間の伴立関係を図6.5に示す。

	<i>Rlock</i>	<i>Wlock</i>
<i>Rlock</i>	○	×
<i>Wlock</i>	×	×

○：伴立 ×：非伴立

図6.5 ロックモードの伴立性

6.2.3 冗長なオブジェクトが存在する場合のロック

次に、オブジェクト x が複数のデータベースサーバに冗長に存在する場合を考える。 x の冗長な物理オブジェクト(レプリカ)を $x_1, \dots, x_m$ とする。例えば、テーブル Emp の断片 $Emp_i = Emp[eno, ename]$ が、二つのデータベースサーバ DBS_1 と DBS_2 に存在する場合であ

る。一つの方法は、レプリカ $x_1, \dots, x_m$ を全てロックしてから、 $x_1, \dots, x_m$ を*read*と*write*することである。しかし、この方法では、 x の全レプリカをロックするために、通信と処理の負荷が大きくなる。このために以下の方法がある。

[冗長オブジェクトのロック方式]

- (1) オブジェクト x の*read*に対しては、あるレプリカ x_j を*Rlock*モードでロックしてから、 $r_v[x_j]$ を実行する。
- (2) x の*write*に対しては、全てのレプリカ $x_1, \dots, x_m$ を*Wlock*モードでロックし、 $x_1, \dots, x_m$ を*write*する。

この方法では、オブジェクト x の*write*を行うトランザクションは x の全レプリカをロックし、更新する必要がある。一方、 x の*read*に対しては、自分のまたは近くのデータベースサーバ*DBS_j*内のレプリカ x_j を*read*できる利点がある。このために、検索中心の応用に適した方法である[図6.6(1)]。

もう一つの方法は、レプリカの中の一つ、ここでは x_1 を主レプリカとして、ロックは主レプリカに対して行う方法である[図6.6(2)]。主レプリカをロックできたならば、あるレプリカを*read*する。*write*を行う場合には、全レプリカの更新を行う。

[主レプリカ方式]

- (1) オブジェクト x の*read*に対しては、 x の主レプリカ x_1 を*Rlock*モードでロックしてから、あるレプリカ x_j を $r_v[x_j]$ する。
- (2) x の*write*に対しては、主レプリカ x_1 を*Wlock*モードでロックし、 x のすべてのレプリカ $x_1, \dots, x_m$ を*write*する。

この方式では、ロックを主レプリカ x_1 に対してのみ行えばよいので、*write*演算を行うときのロックを行うための負荷を減少できる。しかし、すべてのトランザクションが主レプリカ x_1 をロックすることから、主レプリカ x_1 の存在するデータベースサーバに負荷が集中する問題がある。

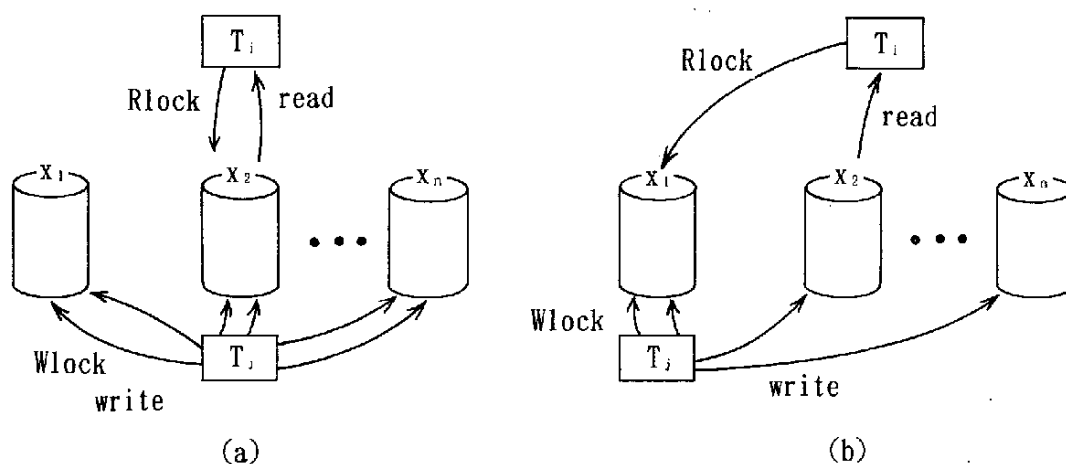


図6.6 二相ロック方式

他に、コーラムに基づいた方法がある。オブジェクト x は、 $m(\geq 1)$ 個のレプリカ $x_1, \dots, x_m$ を持つとする。各レプリカ x_i は、版番号 $version(x_i)$ を持つ。オブジェクト x は、*read*コーラム r -quorumと*write*コーラム w -quorumという二つの整数値を持つ。ここで、両値は、以下の制約を満足している。

$$r\text{-quorum} + w\text{-quorum} > m.$$

まず、トランザクションがオブジェクト x を*read*する場合を考える。 x のレプリカの集合を、 $po(x) = \{x_1, \dots, x_n\}$ とする。

[read方式]

- (1) オブジェクト x のレプリカ集合 $po(x)$ 内の r -quorum個のレプリカを検索する。これらのレプリカの集合を読みだしレプリカ集合 $R(x)$ とする。集合 $R(x)$ を、読出しについてのコーラムとする。
 - (2) この $R(x)$ 内で、版番号が一番大きいレプリカ x_i を求め、この値を読む。
- 次に、オブジェクト x を*write*する場合を考える。

[write方式]

- (1) $po(x)$ 内の w -quorum個のレプリカを検索する。この中で、版番号が一番大きなレプリカ x_i を見つける。これらのオブジェクトの集合を $W(x)$ とし、書込みについてのコラムとする。
- (2) $version(x_i)$ を増加させ、 $v=version(x_i)$ とする。
- (3) 書込みについてのコラム内の全レプリカに、書き込まれる値 t と新しい版番号 v を送る。
- (4) x_j が、新しい値 t と版番号 v を受信したとする。 $version(x_j) < v$ ならば、レプリカ x_j の値を t として、 $version(x_j)$ を v とする。

r -quorumと w -quorumの条件は、 $read$ を行うレプリカと $write$ を行うレプリカの集合は、少なくとも一つのレプリカが共通に含まれていることを意味している。このために、 $read$ は、最新のオブジェクトの値を読める。 $write$ に対しても、全レプリカを $write$ する必要がなく、書込みコラム内のレプリカだけを $write$ すればよい。

6.3 デッドロック

二相ロック方式は、オブジェクトをロックすることにより、他のトランザクションにこのオブジェクトを利用させないようにする同期方式である。また、トランザクションは、操作するオブジェクトを一つずつ順番にロックを行う。このために、トランザクション間にデッドロックが生じる可能性がある。

6.3.1 デッドロックの定義

デッドロックとは、複数のトランザクションが他のトランザクションのロックの解放を待ちあっている状態である。トランザクション間のデッドロックは以下に示す待ちグラフにより検出できる。

[待ちグラフ]

- (1) 各トランザクション T_i に対して、節点 T_i を設ける。
- (2) T_i が T_j によりロックされているオブジェクトの解放を待っているならば、 T_i から T_j

に有向辺を設ける($T_i \rightarrow T_j$)。

三つのトランザクション T_1 、 T_2 、 T_3 があり、おのおのがオブジェクト x 、 y 、 z をロックしているとする。このとき、 T_1 が y を、 T_2 が z を、 T_3 が x のロックを行おうとしているとする。ここで、 T_1 は T_2 の y の解放を待ち、 T_2 は T_3 の z の解放を待ち、 T_3 は T_1 の x の解放を待つ。この関係を示す待ちグラフを、図6.7に示す。

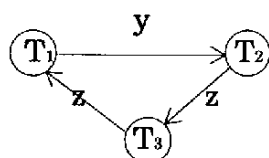


図6.7 デッドロック

図6.7に示す状態は、トランザクション T_1 、 T_2 、 T_3 が互いに待ちあっていることから、デッドロック状態である。図6.7をみると、待ちグラフ内に巡回有向閉路が存在していることがわかる。

【デッドロック】 システムの待ちグラフが巡回閉路を含むならば、かつこのときに限りシステムはデッドロックしている。

次に、図6.8に示すように、データベースサーバ DBS_1 がオブジェクト x と y を持ち、 DBS_2 が z を持つ場合を考える。図6.7に示したように、三つのトランザクション T_1 、 T_2 、 T_3 がおのおのオブジェクト x 、 y 、 z をロックしているとする。このとき、 T_1 が y を、 T_2 が z を、 T_3 が x のロックを行おうとしているとする。 DBS_1 を見てみると、 x と y を利用している T_1 、 T_2 、 T_3 間には、巡回有向閉路は存在しない。 DBS_2 も同様である。しかし、分散データベースシステムとしては、図6.7に示すようにデッドロックとなっている。このように、分散データベースシステムでは、各データベースサーバ内でデッドロックとなっていなくても、システム全体としてみたときにデッドロックとなっていることがある。分散データベースシステムでデッドロックを検出するためには、システム全体の待ち関係を把握するために、データベースサーバ間、トランザクション間での通信が必要となる。

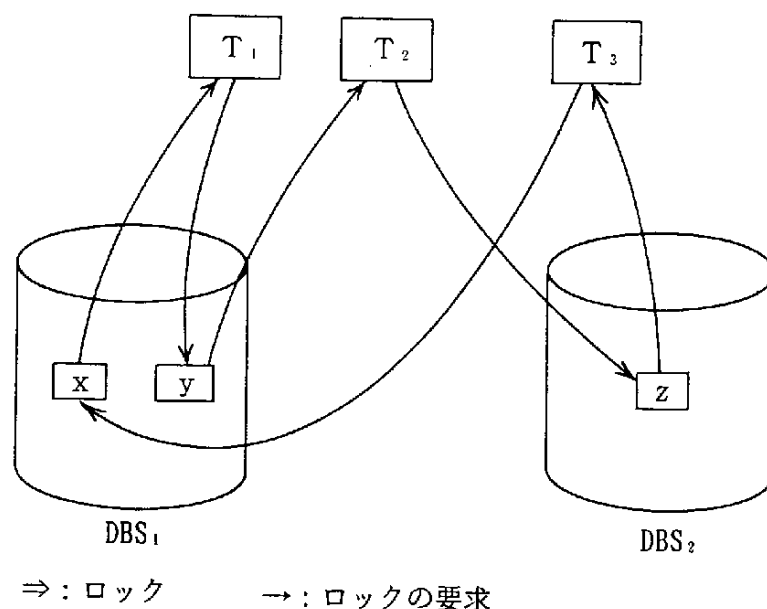


図6.8 分散データベースシステムのデッドロック

各トランザクションが、一度にいくつのオブジェクトを要求するかについて、以下のモデルがある。

- (1) 単一オブジェクトモデル。
- (2) 積モデル。
- (3) 和モデル。
- (4) (C)モデル。

単一オブジェクトモデルでは、トランザクションは、一時に高々一つのオブジェクトを要求する。待ちグラフで、各節点からの出力辺は高々一つである。通常のトランザクションは単一オブジェクトモデルである。これに対して、積、和、(C)モデルでは、トランザクションは一時に複数のオブジェクト $x_1, \dots, x_n (n \geq 1)$ を要求する。積モデルでは、プロセスは要求した全オブジェクト $x_1, \dots, x_n$ が獲得されるまで実行を停止して待つ。従って、待ちグラフ内で各節点の出力辺は複数となる場合がある。また、デッドロックしているプロセスは複数の有向閉路に含まれ得る。しかし、デッドロックの定義は単一オブジェクトモデルのものと同じで、グラフ内の巡回閉路に含まれるトランザクションがデッドロックして

いる。和モデルでは、トランザクションは複数のオブジェクト $x_1, \dots, x_n$ を要求するが、この中の一つでも獲得できれば実行を続けるものである。例えば、ホテルの予約を行う時に、幾つかのホテルに予約の要求を行ってどれか一つでも予約できればよい場合である。和モデルのトランザクションのデッドロックの定義は少し複雑なものとなる。待ちグラフで、有向閉路に含まれるトランザクションがデッドロックしているとは限らない。各トランザクション T_i は、複数のオブジェクトを要求しているので、 T_i からのあるオブジェクトへの出力辺が巡回閉路に含まれても、他のオブジェクトが割り当てられれば、 T_i は実行できるからである。従って、和モデルのトランザクションについてのデッドロックは、以下のよう

に定義できる。

【結び】 待ちグラフ内の以下の条件を充足する節点の部分集合 K を結び(knot)とする。

- (1) K 内の任意の二つの節点 T_i と T_k に対して、 T_i から T_k への有効経路があり、また T_k から T_i へも有効経路がある。
- (2) K 内の節点だけが(1)の条件を満足する。

図6.9に結びの例を示す。

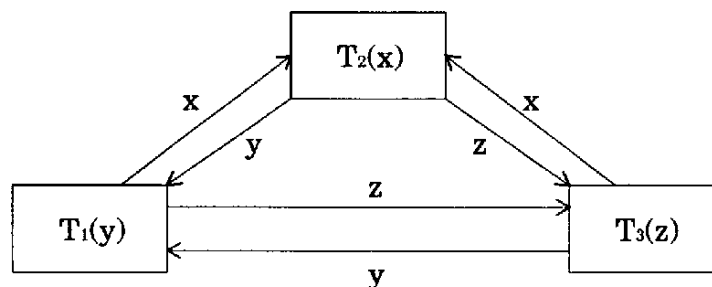


図6.9 結び

【和モデルのデッドロック】 トランザクション T_i が結びに含まれるとき、 T_i はデッドロックしている。

(7)モデルは、積と和モデルの一般化したモデルである。トランザクションは n 個のオブジェクトを要求するが、このなかの r 個のオブジェクトが獲得されたならば実行を続けるものである。単一オブジェクトモデルは(i)モデル、積モデルは(ii)モデル、和モデルは(iii)モデルである。

6.3.2 デッドロックの解決方法

デッドロックの解決方法としては、以下がある。

- (1) 検出方法
- (2) 防止方法

検出方法では、デッドロックを検出して、デッドロック状態のあるトランザクションをアボートしてデッドロックを解消する。防止方法では、デッドロックが生じないようにする方法である。

6.3.3 検出方法

検出方法では、デッドロックを検出してデッドロック状態を解消する方法である。この方法は、商用のシステムで広く用いられているものである。

【デッドロック検出解除方法】

- (1) 現在の状態に対するトランザクション間の待ちグラフ G を作り、巡回閉路を探す。
- (2) 巡回閉路が見付かれば、この閉路内のあるトランザクション T を（生贄として）選択し、 T をアボートさせる。

デッドロックが生じていなくても、デッドロック検出手順がデッドロックが生じていると見なしてしまう状態を偽デッドロック（falseまたはphantomデッドロック）という。ただし、以下の性質が示されている[BERN87]。

【定理】 各トランザクションが二相ロック形式であり、頻繁にアボートしないならば、偽デッドロックは生じない。

デッドロックの検出解除方法では、以下が問題となる。

1. 検出の間隔

周期的にデッドロック検出を行う方法が用いられる場合が多い。このとき、周期をどの位にするかが問題となる。周期を短くすると、デッドロック検出のための負荷が増大してしまう。一方、周期を長くすると、デッドロックが長時間生じていることになる。

2. 強制終了されるトランザクションの選択

デッドロックが検出されたとき、どのトランザクションを強制終了させるかが問題となる。実行されている時間の一番短いもの、長いもの、ロックしているオブジェクト数が最少のもの等を選択する方法がある。現在のデータベースシステムでは、システムがデッドロックを検出し、デッドロック状態のトランザクションにこのことを通知する方法が用いられている。デッドロックに対して、どのように対処するかは各トランザクションの問題となる。

分散データベースシステムでのデッドロックの検出方法としては、集中型と分散型がある。集中処理方式では、デッドロック検出を行う指揮プロセスCが存在する。

[集中処理方式] [図6.10]

- (1) 各トランザクション p_i は、オブジェクト x の獲得を要求するとき、指揮プロセスCに要求を送信する。 x を解放するときにも、解放要求をCに送信する。
- (2) Cは、 x の獲得または解放要求を受けたならば、待ちグラフに変更を加える。獲得要求に対して、待ちグラフを調べる。グラフが、巡回閉路を含むならば、デッドロックが生じている。

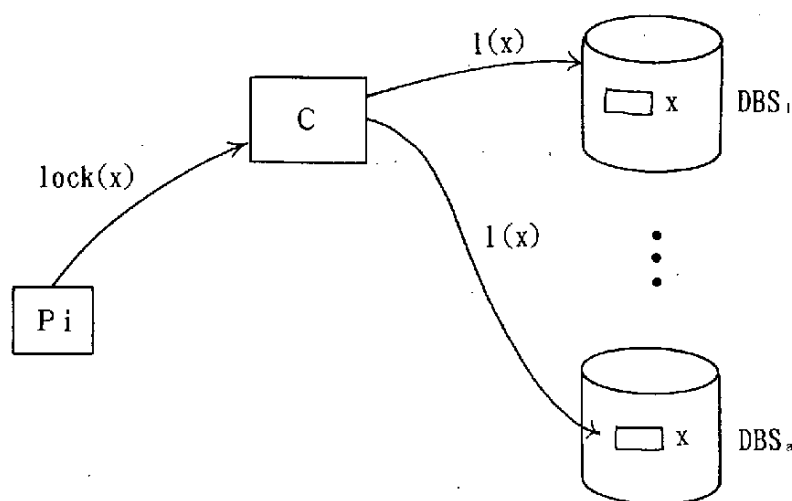


図6.10 集中型制御

集中処理方式は簡単であり、実装が容易であるが、以下の問題点がある。

1. 各トランザクション T_i は、指揮プロセス C と通信を行わねばならない。このために、 C に負荷が集中することになる。

2. 指揮プロセス C が障害を起こすと、他の全てのプロセスが待ちとなる。

他に、待ちグラフを指揮プロセス C に集める方法も考えられる。

(1) 各データベースシステムサーバ C_i は、 C_i 内のオブジェクトについての待ち G_i を持つ。

C_i は、周期的または指揮プロセス C からの要求により、 G_i を C に送信する。

(2) C は、 $G_1, \dots, G_n$ から分散データベースシステム全体の待ちグラフ G を作成する。 G が巡回閉路を含むならば、デッドロックが起きていると判断する。

各データベースサーバと指揮プロセス C 間の通信遅延があるために、指揮プロセス C に届けられた待ちグラフがそのサーバの現在の状態を示していずに、送信前の状態を示している。このために、この手順では擬デッドロックが生じる場合がある。各サーバの待ちグラフを同時に獲得できるならば、偽デッドロックは生じない。しかし、分散システムでは、同時に各コンピュータの状態を獲得することはできない。

次に、指揮プロセスの存在しない場合について考える。トランザクションが待ち状態になったときに、デッドロック検出手順が起動される。各データベースサーバは、自分の持つオブジェクトについての待ちグラフを管理している。このグラフ内の情報がデッドロック検出アルゴリズムの実行中に、他のサーバに送信される。まず、[CHAND83]の検出手順について述べる。この手順では、各トランザクションは同時に複数のオブジェクトを要求できる積モデルである。各トランザクションが、あるオブジェクト x を要求し待つときに、検出手順が起動される。待ちとなったトランザクションは、プローブ(probe)というメッセージを x が割り当てられているトランザクションに送信する。プローブは、三つ組 $\langle i, j, k \rangle$ として与えられる。ここで、 i は待ちの状態となったトランザクション T_i を示す。 j はプローブを送信したトランザクション T_j を示す。 k は、プローブの宛先のトランザクション T_k を示す。

[Chandy, Misra, Haasの手順] [CHAND83]

(1) オブジェクト x_j がトランザクション T_i に割り当てられているとする。ここで、 T_i が、 x_j を要求して待ちとなったとする。このような各 x_j について、 T_i は、プローブ $\langle i, i, j \rangle$

を T_i に送信する。

(2) T_i が、プローブ $\langle j, k, i \rangle$ を受信したとする。 $j=i$ であるとき、 T_i がはじめに送信したプローブが T_i に返ってきたことを示す。したがって、 T_i は、デッドロックしていることがわかる。

(3) (2)で、 $j \neq i$ であるとする。トランザクション T_i が要求して、割当てを待っている各オブジェクト x_h について考える。各オブジェクト x_h は、トランザクション U_h に割り当てられているとする。 T_i は、プローブ $\langle j, i, h \rangle$ を T_h に送信する。

図6.11に、5つのトランザクション p_0, p_1, p_2, p_3, p_4 の例を示す。 p_0 がオブジェクト x を待ちとする。 p_0 は、プローブ $\langle 0, 0, 1 \rangle$ を、 x が割り当てられている p_1 に送信する。 p_1 は、 y と v を待っているとする。 p_1 は、プローブを受信したならば p_2 にプローブ $\langle 0, 1, 2 \rangle$ を送信する。また、 p_4 に $\langle 0, 1, 4 \rangle$ を送信する。このようにして、 p_1 は、 p_3 からプローブ $\langle 0, 3, 1 \rangle$ を受信する。ここで、 p_0 は、デッドロックしていることがわかる。

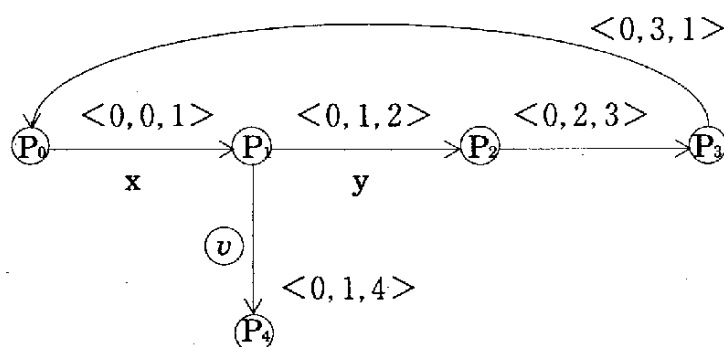


図6.11 分散型デッドロック検出

6.3.4 防止方法

防止方法では、デッドロックが生じないようにトランザクションにオブジェクトをロックさせる方法である。このための方法として以下がある。

- (1) オブジェクトを順序付けて、トランザクションにこの順にロックさせる。
- (2) トランザクションを全順序付ける。

(1)のオブジェクト順序付け方法では、オブジェクトをロックする順番をあらかじめ決めておく方法である。例えば、三つのオブジェクト a, b, c に対して、各トランザクションはこの順にオブジェクトのロックを行わせる方法である。この方法では、順位の高いオ

ブジェクトにロック要求が集中することになり、トランザクション実行の同時実行性を減少させてしまう点が問題となる。

次に、(2)の方法について述べる。各トランザクション T に対して、 T が起動された時刻を T の時刻印 (time stamp) $ts(T)$ として与える。ここで、 x をオブジェクト、 T_1 と T_2 をトランザクションとする。いま、オブジェクト x は T_2 によりロックされているとし、新たに T_1 が x をロックしようとしている場合を考える[図6.12]。

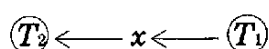


図6.12 待ち関係

このとき、以下の二つの方法がある。

[横取り(preemption)方法]

- (1) $ts(T_1) < ts(T_2)$ ならば (T_1 が T_2 より古い)、 T_2 をアボートさせ、再実行させる。
- (2) $ts(T_1) > ts(T_2)$ ならば、 T_1 を待たせる。

[非横取り(nonpreemption)方法]

- (1) $ts(T_1) < ts(T_2)$ ならば (T_1 が T_2 より古い)、 T_1 をアボートさせ T_1 に新しい時刻印を与えて再実行させる。
- (2) $ts(T_1) > ts(T_2)$ ならば、 T_1 を待たせる。

以上のデッドロック防止方法を用いると、デッドロックは生じない。しかし、あるトランザクションがロックしているオブジェクトを他のトランザクションがロックしようとしたときに、トランザクションのアボートと再実行が生じる場合がある。従って、トランザクションがどの位再実行されるかが問題となる。横取り方法では、トランザクションは時刻印の小さい(即ち、古い)ものが優先して実行される。これに対して、非横取り方法では新しいトランザクションが優先して実行される。従って、オブジェクト x を操作しようとする古いトランザクションがあるとき、これらより新しいトランザクションが x を獲得してしまうと、古いトランザクションはアボートされてしまう。このために、横取り方法の方が、一般的に非横取り方法よりもアボートされるトランザクションは少ない。

7章 コミットメント制御

トランザクションが複数のデータベースサーバを更新する場合には、更新の原子性、即ち、全てのデータベースサーバ内のデータが更新されたか、または全く更新されなかったかのいずれかであることを保障する必要がある[図7.1]。このための制御がコミットメント制御である。ここでは、コミットメント制御について考える。

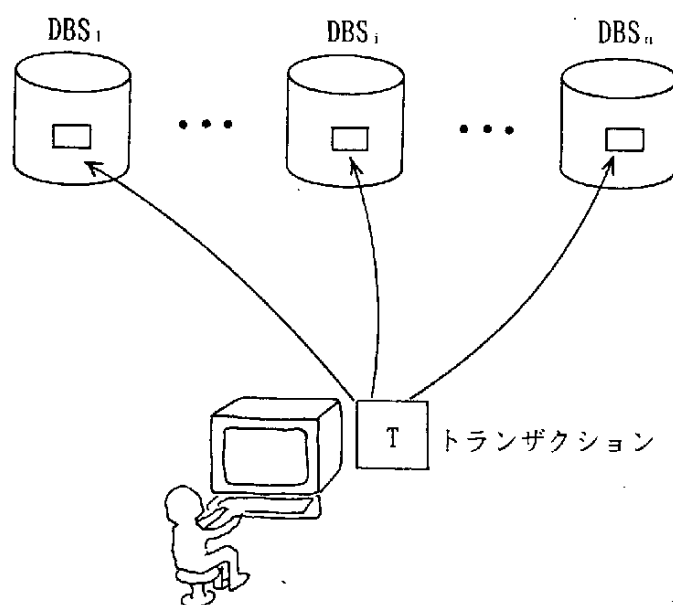


図7.1 コミットメント制御

7.1 障 害

コミットメント制御では、分散システムの障害について、以下の仮定を設けている。

〔障害についての仮定〕コミットメント制御では、プロセス停止のみを障害とする。

障害していないプロセスを動作中プロセスであるとする。プロセスがすべて障害（即ち、停止）する障害を全体障害とする。少なくとも一つのプロセスが動作している障害を部分障害とする。

通信ネットワークは、TCP/IP、OSI等の通信プロトコルによりメッセージが紛失したり、文字化け等の通信障害に対しても頑強であり、メッセージを宛先に送信順に届けることができる。通信ネットワークが処理できない障害として、ネットワークの分割がある。

〔ネットワーク障害〕 ネットワーク分割とは、プロセスが複数のグループに分割され、各グループ内のプロセスは互いに通信を行えるがグループ間では通信を行えない状態である。

あるグループ内のプロセスが、他のグループ内のプロセスは障害していると考えて独自にコミットメントの処理を行うとグループ間で異なった決定を行ってしまう可能性がある。図7.21は、プロセスA、B、C、D、Eから構成されるシステムを示している。ここで、CとD間の通信リンクが切断すると、A、B、Cと、DとEとの二つのグループに分割される。Aは、プロセスDとEは停止しているものとみなし、一方、Dは、A、B、Cが停止していると考ええる。

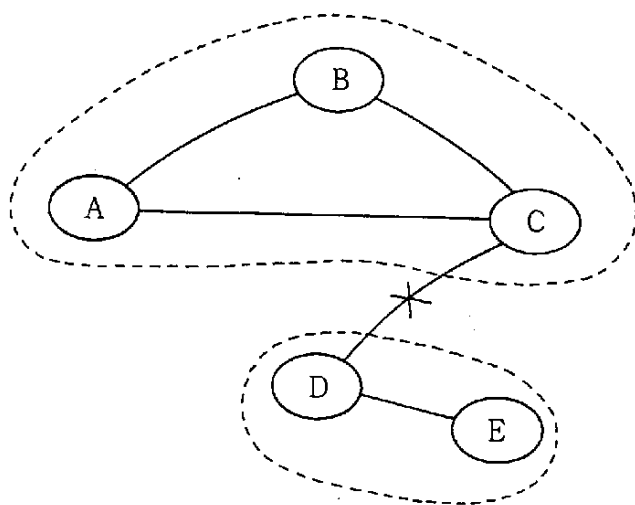


図7.2 ネットワーク分割

7.2 二相コミットメント

コミットメント制御のためのプロトコルである二相コミットメント(two-phase commitment)[SKEE81,SKEE83]プロトコルについて述べる。例として、プロジェクトのメンバでテニスに行くことを考える。ここで、プロジェクトの全員が出席するときだけテニスに行くとする。プロジェクトのリーダーは、全員に電話をかけ出欠を確認する。ここで、一人でも欠席者いた場合には、出席の意志表示を行ったメンバにテニスに行かないことを電話する。全員が出席ならば、リーダーは全メンバに再度電話を行い、テニスに行くことを通知する。このように、全員出席するときのみテニスに出掛けることの決定を行うためには、出欠の意志表示を求めることと、テニスに出掛けるかどうかの決定を通知することの二つの手順が必要となる。さらに、リーダーから電話を受け、テニスに出席すると意志表示したメンバはリーダーからの電話を待つ。この状態が不確定状態である。このとき、例えば、他のグループからゴルフに誘われてもゴルフに行くことはできない。各メンバは、ただリーダーからの電話を待つことだけができる。

以上から、コミットメントを行うためのプロトコルを考える。まず、プロセスがトランザクション T のコミットについて、諾又は否を他のプロセスに示すことを意志表示とする。全プロセスがコミットかアボートかの合意に達することを決定とする。

【不確定状態】 プロセスが諾表示を行った後に、全体の決定を待っているとき、そのプロセスは不確定状態にある。

プロセスがコミットメント制御の処理を続行するために障害プロセスの復旧を待たねばならないとき、このプロセスはブロックしているとする。プロセスがブロックしないための条件として以下がある。

【非ブロック条件】 ある動作中プロセスが不確定状態にあるならば、動作中か障害中かに関わらずコミットを決定しているプロセスはないとき、コミットメント手順は非ブロックである。

二相コミットメント制御では、各トランザクション T に対して、一つの指揮プロセス C と $n(\geq 1)$ 個の関係プロセス $P_1, \dots, P_n$ がある。ここで、指揮プロセス C は、 T が実行されてい

るクライアントであり、関係プロセスは P_i 、データベースサーバである。各 P_i は、プロセスの障害に対しても、障害を受けない安定記憶としてのログ L_i を持つとする。二相コミットメントの基本手順を以下に示す[図7.3、図7.4]。

[二相コミットメント(2PC)プロトコル]

- (1) 指揮プロセス C は、トランザクション T の操作演算を全関係プロセス $P_1, \dots, P_n$ に送信し、各 P_i は操作演算を実行する。
- (2) 指揮プロセス C は、 T からコミット演算(c)を受けたとき、全関係プロセス $P_1, \dots, P_n$ に**VOTE-REQ**(VR)を送信する。
- (3) 各関連プロセス P_i は、**VOTE-REQ**を受信したとき、 T をコミットできるならば**YES**を指揮プロセス C に送信する。このとき、 P_i での T による更新結果を安定記憶ログ L_i に記憶する。コミットできないならば、**NO**を C に送信してアボートする。
- (4) C は、全関係プロセスから**YES**を受信したならば、**COMMIT**を送信し、 T をコミットする。ある P_i から**NO**を受信したならば、**YES**を受信した全関係プロセスに**ABORT**を送信し、 T をアボートする。
- (5) 各 P_i は、 C から**COMMIT**を受信したならば T をコミットする。このとき、ログ L_i 内の T の更新データを用いてデータベースの更新を行う。また、**ABORT**を受信したならばアボートする。このとき、データベースの更新を行わず L_i 内の T の更新データを消去する。

図7.3は、全関係プロセスがトランザクションをコミットする場合を示している。各関係プロセス P_i で、**VOTE-REQ**を受信して、**YES**を送信してから、**COMIT**または**ABORT**を受信するまでの状態が、不確定状態である。 P_i が不確定状態にあるとき、コミットの意志表示を他に通知しているので、意志表示に反するアボートを行えない。即ち、 P_i は、指揮プロセス C からの**COMMIT**又は**ABORT**の通知を待つことだけができる。 C が、**VOTE-REQ**を送信してから、関係プロセスが**YES**または**NO**が指揮プロセスに届くまでを、投票相とする。次に、 C が**COMMIT**または**ABORT**を関係プロセスに送信する相を、決定相とする。このように、二つの相から構成されていることから、二相コミットメントプロトコルといわれる。図7.4には、関係プロセス P_n が、トランザクション T をアボートする場合を示している。

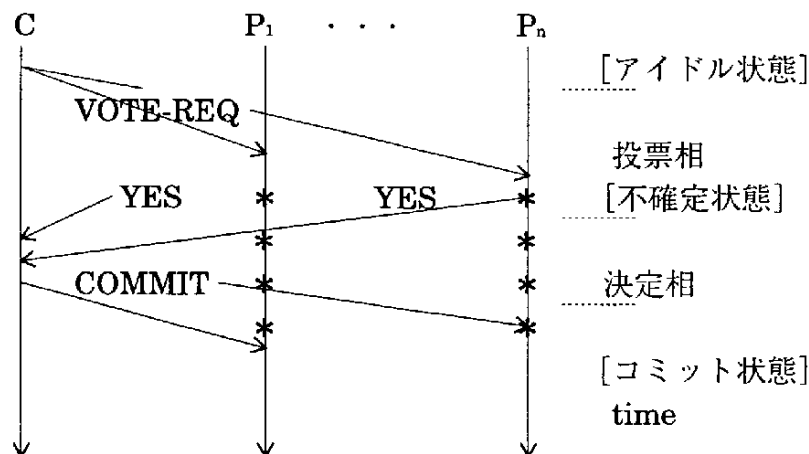


図7.3 2 P Cの基本手順（コミット）

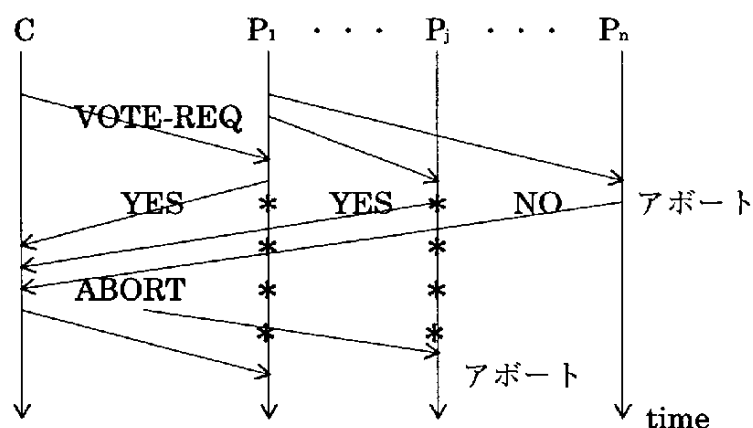


図7.4 2 P Cの基本手順（アボート）

7.3 終結プロトコル

指揮プロセスCが障害を起こした場合を考える。*VOTE-REQ*を送信した後に、Cが障害を起こした場合には、関係プロセスは待ち続けてしまう。この問題を解決するために、関係プロセスによりコミットメント制御の終了を行う手順が必要になる。これを終結プロトコル(termination protocol)という。

〔終結プロトコル〕 指揮プロセスが障害した場合に、関係プロセス間の通信により、コミットまたはアボートの決定を行うためのプロトコルを終結プロトコルという。

しかし、指揮プロセスCがどの関係プロセスにもCOMMITまたはABORTを送信する前に障害したならば、全てのプロセスをブロックして指揮プロセスCの復旧を待つ必要がある[図5.33(1)]。一方、図5.33(2)に示すように、一部の関係プロセス P_j にCOMMITまたはABORTの意志決定を送信した後に障害を起こした場合には、不確定状態のプロセスは P_j に問い合わせることにより決定を行うえる。このための終結プロトコルについて考える。

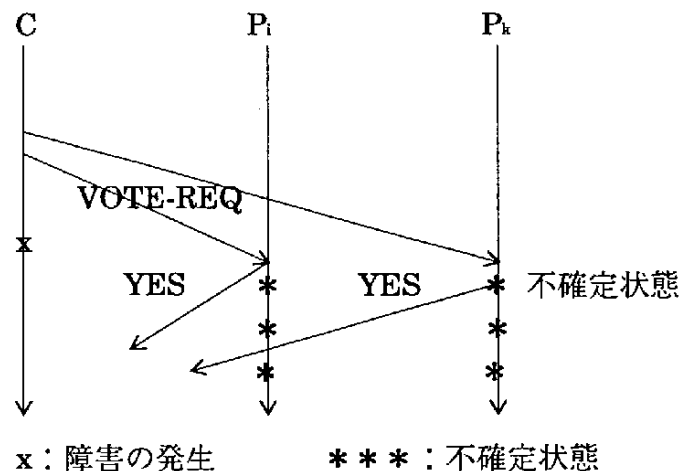


図7.5 ブロック状態

[協調的終結プロトコル] 不確定状態にある関係プロセス P_j が、タイムアウトしたとき以下の手順を行う[図5.34]。

- (1) 関係プロセス P_j は、すべてのプロセスに**STATE-REQ(SR)**を送信する。
- (2) **STATE-REQ(SR)**を受信した動作中のプロセスは、自分の状態（コミット、アボート、不確定）を送信する。
- (3) P_j は、状態についての応答を受信したならば、以下の終結規則に従う。

終結プロトコルが用いる終結規則を以下に示す。

[終結規則]

- TR1：すべての動作中プロセスが不確定状態ならば、 P_j は不確定状態のままで待つ。
- TR2：あるプロセス P_k がコミット(*Committed*)状態ならば、 P_j はコミットする。
- TR3：ある P_k がアボート(*Aborted*)状態ならば、 P_j はアボートする。

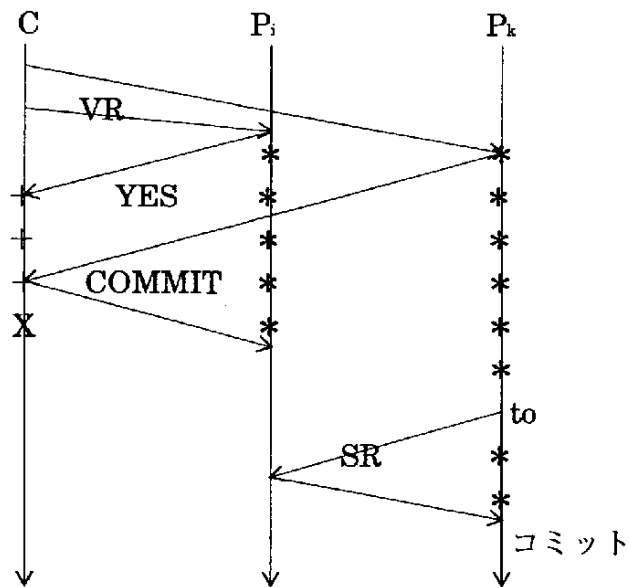


図7.6 終結プロトコル

図7.6では、関係プロセス P_k がタイムアウトにより、他の関係プロセス P_j に *STATE-REQ*を送信する。 P_j はコミットしているので、このことを P_k に通知する。 P_k はトランザクション T をコミットする。

7.4 障害復旧

各関係プロセス P_i は、停止障害から復旧したとき、ログ L_i 内に停止したときの状態が記憶されているとする。 P_i は L_i の状態に復旧する。このとき P_i が不確定状態であるならば、 P_i だけではコミットまたはアボートの決定を行えない。即ち、独立復旧を行えない。 P_i が復旧したとき次の復旧プロトコルを用いる。

[復旧プロトコル]

- (1) P_i は不確定状態であれば、終結プロトコルにより決定を行う。
 - (2) P_i がYES又はNOを送信する前の状態（アイドル状態）であれば、アボートする。
- ネットワーク障害に対しても、同様にブロックする。

7.5 評価

二相コミットメント(2PC)プロトコルの性質について考える。

[二相コミットメントプロセスの性質]

- (1) プロセス障害とネットワーク障害に対して頑強である。すなわち、プロセスの障害とネットワークの障害に対して、全プロセスで同一の決定に導ける。
- (2) しかし、指揮プロセスの障害とネットワーク障害に対して、ブロックする可能性がある。

次に、性能について考える。ラウンドを最遠隔のプロセスにデータ単位が届くまでの遅延時間とする。このとき、二相コミットメントプロトコルの実行時間は以下となる。

[実行時間]

- (1) 障害のない場合 3 ラウンド。
- (2) 障害のある場合 5 ラウンド。

次に、送信されるメッセージ数について考える。関係プロセスの数を n とする。また、プロセス障害が起きている場合に、終結プロトコルを起動するプロセス数を m とする。このときメッセージ数は以下となる。

[メッセージ数]

- (1) 障害のない場合： $3n$ 。
- (2) 障害のある場合：最悪の場合を考える。終結プロトコルを一つのプロセスが起動しこれが終了した後に、次のプロセスが終結プロトコルを起動していく場合である。まず、はじめのプロセスが終結プロトコルを起動した時、 $n-1$ 個の $STATE-REQ$ を送信し、 $n-m$ 個のプロセスから応答を得る。次のプロセスが終結プロトコルを起動した場合には、 $n-1$ 個の $STATE-REQ$ を送信し、 $n-m+1$ 個のプロセスから応答を得る。このように、送信されるメッセージ数は以下となる。

$$(n-1)m + \sum_{j=1, \dots, m} (n-m+j-1) = 2(n-1)m - m^2/2 + m/2。$$

二相コミットメントプロセスを実現するためには、各データベースサーバ P_i でコミット

する機能に加えて、*VOTE-REQ*を受信した時に副トランザクションの更新状態をログ L_i に退避する機能が必要となる。この機能がないと二相コミットメントを行えない。

7.6 三相コミットメント

二相コミットメントプロトコルは、指揮プロトコルの障害に対して関係プロセスがブロックする問題があった。この問題を解決するために三相のコミットメントプロトコル[SKEE83]が示されている。三相のコミットメント(3PC)プロトコルの基本手順は以下のようである。

[三相コミットメント(3PC)プロトコル] [図7.7]

- (1) 指揮プロセス C は、*VOTE-REQ*を全関係プロセス $P_1, \dots, P_n$ に送信する。
- (2) 関係プロセス P_i が*VOTE-REQ*を受信したならば、*YES*または*NO*を指揮プロセス C に送信する。*NO*を送信したときは、アボートする。
- (3) 指揮プロセス C が全関係プロセスから*YES*を受信すれば、*PRE-COMMIT*を全関係プロセスに送信する。一つでも*NO*を受信したならば、*ABORT*を全関係プロセスに送信してアボートする。
- (4) 関係プロセス P_i が*PRE-COMMIT*を受信したならば、*ACK(acknowledgment)*を指揮プロセス C に送信する。*ABORT*を受信したならばアボートする。
- (5) 指揮プロセス C は、全関係プロセスから*ACK*を受信したならば、*COMMIT*を送信して停止する。
- (6) 関係プロセス P_i は、*COMMIT*を受信したならば、コミットして停止する。

二相コミットメントは、不確定状態のプロセスが、指揮プロセス C からの*COMMIT*または*ABORT*を待っていて、タイムアウトしたときにコミットもアボートもできないためにブロックしてしまう。これに対して、三相コミットメントプロトコルは、不確定状態のプロセスがタイムアウトした場合にはアボートできる。三相コミットメントプロセスはブロック問題を生じない利点があるが、処理時間が5ラウンド、送信されるデータ単位数が $5n$ となることから性能面で問題がある。

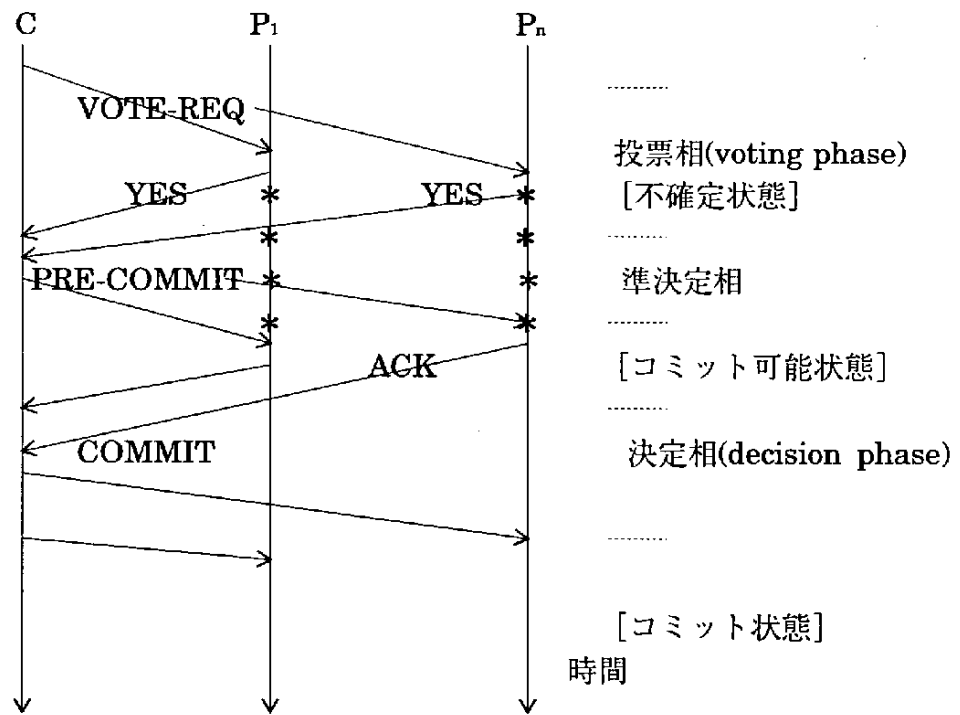


図7.7 三相コミットメントプロトコルの基本手順

8 章 分散質問処理

分散データベースシステムでは、複数のデータベースサーバ内のテーブルに対する検索を行う場合がある。まず、例を考えてみる。

〔例〕図2.2に示した二つのテーブルEmpとDeptを考える。ここで、EmpはデータベースサーバDBS₁にあり、DeptはDBS₂にあるとする。ここで、二つのテーブルに対する以下のSQLを考える。

```
select ename, dname from DBS1.Emp, DBS2.Dept
where DBS1.Emp.dno = DBS2.Dept.dno and sal > 200 —— (1)
```

まず、(1)のSQLを、DBS₁とDBS₂内のテーブルに対する以下のSQLを実行する。

```
select ename, dno from Emp into T1 where sal > 200 —— (2)
select dname, dno from DBS2.Dept into T2 —— (3)
```

これを処理するためには、中間結果のテーブルT₁またはT₂を移動させる必要がある。この方法として以下が考えられる〔図5.40〕。

- (1) T₁をDBS₁からDBS₂に移動し、T₂とDBS₂で結合を行う。
- (2) T₂をDBS₂からDBS₁に移動し、DBS₁でT₁結合を行う。
- (3) T₁とT₂を、第三のサーバDBS₃に移動し、DBS₃で結合を行う。

この中で、サーバ間での通信時間と各サーバでの処理時間を最小とするものが選択される。例えば、通信時間を最小とすることを考えよう。テーブルT₁とT₂の大きさを比較してみると、T₂の方が小さい。従って、T₂を、DBS₂からDBS₁に送信して、DBS₁でT₁とT₂の結合を行う。

分散質問処理とは、複数のデータベースサーバ内のテーブル間のSQL質問を通信時間と処理時間を最小とするように、各データベースサーバでの処理とデータベースサーバ間の通信を行わせることである。分散結合処理の一つの技術として、テーブル間の準結合(semi-join)がある。まず、準結合について説明する。RとSを二つのテーブルとする。RのSに対する条件pについての準結合は以下のようなものである。ここで、 Ω_R は、テーブルRの属性集合とする。

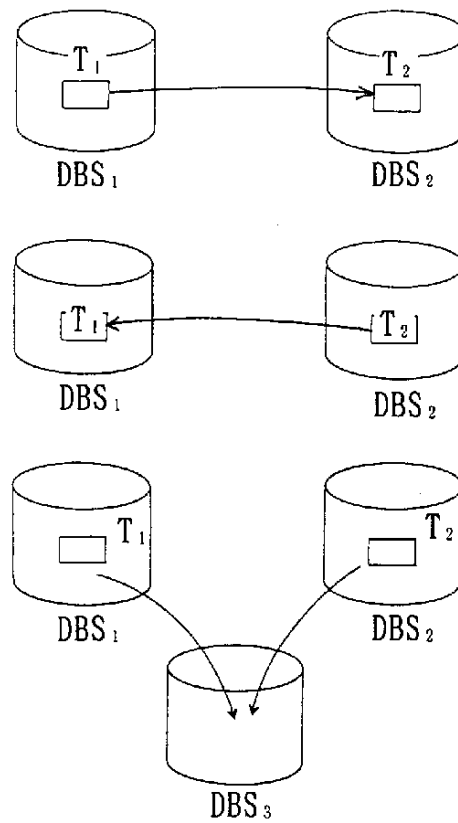


図8.1 分散質問処理

$$R[p > S] = \{r \mid r \in R \wedge \exists s \in R[p]S\} = (R[p]S)[\Omega_R].$$

ここで、準結合 $R[a=a > S]$ を $R[>S]$ と書く。これは、 $(R*S)[\Omega_R]$ である。例えば、 $\text{Emp}[>\text{Dept}]$ は、 $(\text{Emp}*\text{Dept})[\text{eno}, \text{ename}, \text{sal}, \text{dno}]$ である。 DBS_1 にあるテーブル Emp と DBS_2 にある Dept 間の自然結合 $\text{Emp}*\text{Dept}$ を考える。例で述べたように、 Emp または Dept を他方のデータベースサーバに送信して、結合を行う方法である。これに対して、準結合を用いると以下のような処理が考えられる。

[準結合によるデータベースサーバ間の結合処理] [図5.42]

- (1) Emp の dno の値 $\text{Emp}[\text{dno}] (= T_1)$ を求める。 DBS_1 は、 T_1 を DBS_2 に送信する。
- (2) DBS_2 は、 T_1 を受信したならば、 Dept と T_1 との結合 $T_2 = \text{Dept} * T_1$ を行う。 $T_3 = T_2[\text{dno}]$ を、 DBS_1 に送信する。
- (3) DBS_1 は、 T_3 を受信したならば、 $T_4 = \text{Emp} * T_3$ を行う。

この時点で、 DBS_1 の $T_4[\text{dno}]$ と DBS_2 の $T_2[\text{dno}]$ は、同一である。従って、 T_4 と T_3 を利用

者のいるクライアントに送信し、 $T_1 * T_2$ により質問の解を得れる。ここで、任意のテーブル T に対して、 $|T|$ は、 T の大きさとする。以上の準結合により送信されるデータ量は、 $|T_1| + |T_3| = |Emp[dno]| + |(Dept * Emp)[dno]|$ である。ここで、 $|T_3| \leq |T_1|$ であるので、 $|T_1| + |T_3| \leq 2|Emp[dno]|$ である。一方、 Emp を送信する場合は送信量は、 $|Emp|$ である。従って、 $|Emp[dno]| \leq |Emp|$ ならばテーブルを転送するよりは、準結合を行った方が送信量を減少できる。以上は、 $Emp > Dept$ を示したが、 $Dept > Emp$ も同様に行うことができる。

次に、二つのテーブル R と S に自然結合演算を適用した結果得られるテーブル $T = R * S$ の基数の見積もり方法について考える。ここで、 R 内の属性 $a \in \Omega_R$ に対して、以下の記法を導入する。

$|R|$ = テーブル R の行数。

$dom(a)$ = 属性 a の定義域。

以上を用いて、 R 内の属性 a に対する選択度(selectivity) σ_{Ra} を以下のように定義する。

$$\sigma_{Ra} = |R[a]| / |dom(a)|.$$

自然結合 $T = R * S$ に対して、 $T[a]$ の見積もりについて考える。ここで、以下の仮定を設ける。

[仮定] テーブル内の属性の値の分散は互いに独立である。

この仮定が成り立つとすると、 $|T[a]|$ の期待値は以下ようになる。

$$|T[a]| = |dom(a)| *$$

$$(|R[a]| / |dom(a)|) * (|S[a]| / |dom(a)|) = |R[a]| * |S[a]| / |dom(a)|$$

属性 a のテーブル R での選択度 σ_{Ra} は、 a の定義域 $dom(a)$ 内の任意の値 v が R に含まれる確率を与えている。従って、 $dom(a)$ 内の任意の値が自然結合の結果テーブル T 内に含まれると確率は、二つのテーブル R と S の両方に含まれる確率であるので $\sigma_{Ra} * \sigma_{Sa}$ である[図8.2]。従って、 $|T[a]|$ は $|dom(a)| * \sigma_{Ra} * \sigma_{Sa}$ である。

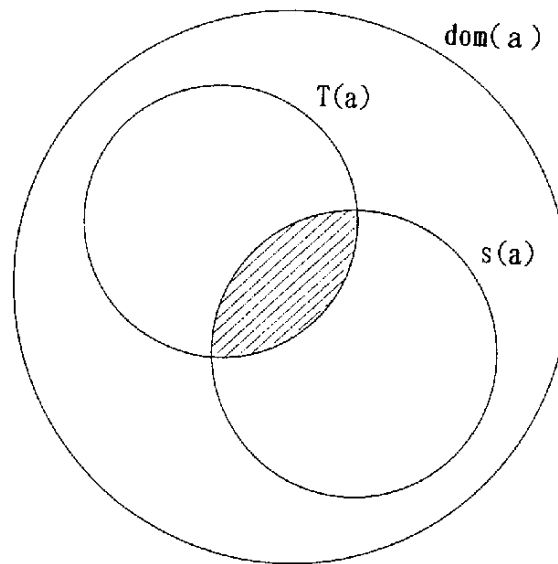


図8.2 定義域

例として、次のようなテーブル R と S を属性 a により自然結合 $T=R*S$ する場合を考える。
 $|R[a]|=100$, $|S[a]|=50$, $|dom(a)|=500$ 。このとき、 $\sigma_{Ra}=100/500=0.2$ 、 $\sigma_{Sa}=50/500=0.1$
 であり、 $|T[a]|$ の期待値は $500*(100/500)*(50/500)=5000/500=10$ である。次に、自然結合
 $T=R*S$ の $|T|$ の見積もりについて考える。属性値の分散についての仮定が成り立つとす
 ると、以下の式により $|T|$ の期待値を与えることができる。

$$|T|=|R|*|S|*\sigma_{Ra}*\sigma_{Sa}$$

9 章 多重化

フォールトトレラント分散型データベースシステムのシステムモデルについて述べる。

9.1 システム構成

データベースシステムは、クライアントとサーバから構成され、利用者はクライアントを通してサーバに操作要求を送信し、サーバは操作要求を受信したならば、これを実行し実行結果をクライアントに返す。分散型データベースシステム [CERI84, OSZU91, JDDBS] は、複数のサーバとクライアントから構成される。本論文では、クライアントとサーバをプロセスとする。ここで、分散型データベースシステムを通信ネットワークにより相互接続されたプロセス $\{P_1, \dots, P_n\}$ から構成されたとする。また、このような形態を論理システム $D = \{P_1, \dots, P_n\}$ とする [図9.1]。各プロセスは、SQL [DATA87] を入力し、SQL を実行して、結果を出力するものとする。

分散データベースシステム D 内のあるプロセスに障害が起こると、利用者はシステムを使用できなくなる場合がある。特に、サーバのプロセスに障害が起きた場合、そのサーバの提供するデータベースが使用不能となる。このために、サーバを多重化し各々異なったコンピュータに配置する。データベースサーバのプロセスを多重化したものをレプリカとする。フォールトトレラント分散データベースシステムでは、あるレプリカが障害しても残りのレプリカによりサービスを継続して提供できる。各プロセス P_i のレプリカを $P_{i1}, \dots, P_{im_i}$ ($m_i \geq 1$) とする。レプリカの集合 $\{P_{i1}, \dots, P_{im_i}\}$ を P_i のレプリカ集合とし、 $P_i = \{P_{i1}, \dots, P_{im_i}\}$ と書く。各プロセスのレプリカ集合 $\{\{P_{11}, \dots, P_{1m_1}\}, \dots, \{P_{n1}, \dots, P_{nm_n}\}\}$ を D の物理システム P_D とする。

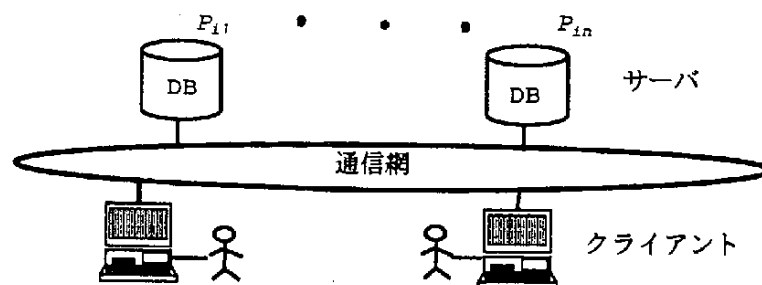


図9.1 論理システムD

9.2 障害

分散データベースシステムDの被る障害について考える。まず、障害とは、要求された機能をシステムのハードウェア、ソフトウェアまたは、その両方が遂行できなくなる偶発的条件である。分散データベースシステムにおける障害には、通信ネットワークの障害とプロセスの障害がある。本論文では、通信ネットワークは、高信頼であるとする。即ち、通信プロトコルにより、メッセージの紛失やネットワークの分割などの障害は起きないと仮定する。

プロセス障害には、停止障害とビザンチン障害[LAMP82]がある。停止障害はプロセスがある時間以上停止する障害であり、ビザンチン障害はプロセスが誤動作する障害である。ある一つのレプリカが停止障害を起こしたとき、レプリカが二つ以上あれば残りのレプリカが計算を継続できる。また、ある一つのレプリカがビザンチン障害を起こしたときは、レプリカを三つ以上作り、多数決を行なうことによって障害を起こしたレプリカを検出できる。障害したレプリカは切り離され、残りのレプリカにが、計算を継続する。

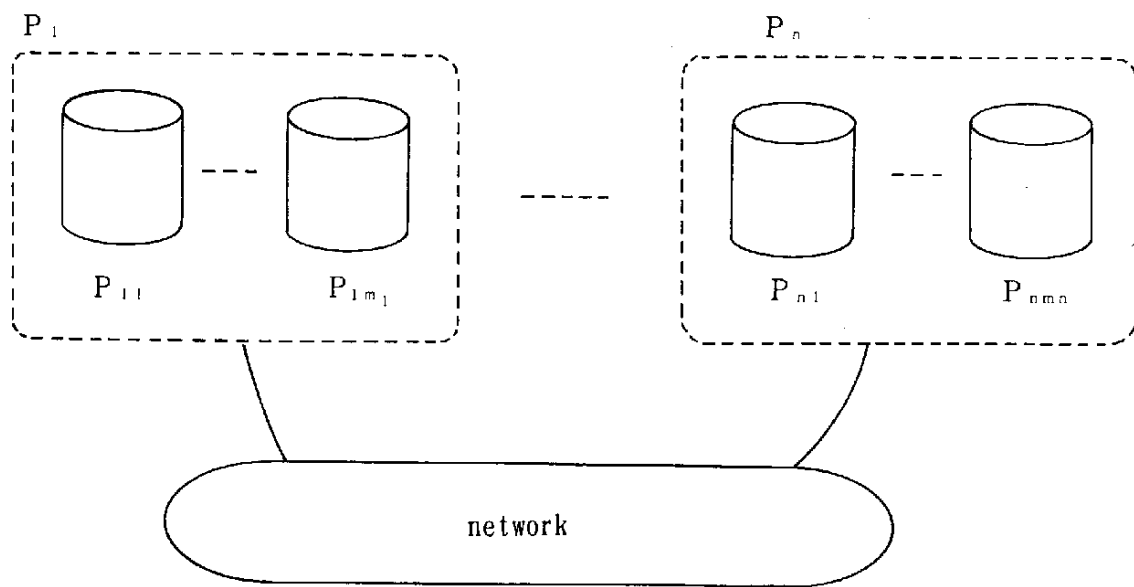


図9.2 物理システム P_D

9.3 多重化方式

レプリカを複数作成して、システムをフォールトトレラントとする方法には、以下の多重化方法がある[POWE94]。

[多重化方式]

1. 能動的多重化 (active replication)
2. 受動的多重化 (passive replication)
3. 準能動的多重化 (semi-active replication)

プロセス P_i のレプリカグループを $\{P_{i1}, \dots, P_{im_i}\} (m_i \geq 1)$ とする。

(1) 能動的多重化

全てのレプリカ $P_{i1}, \dots, P_{im_i}$ が、同じ入力を受け、同じ計算を行ない、同じ出力を行う[図9.3-(1)]。各レプリカ P_{ij} は、全てのレプリカの出力値を比較し多数決を行なうことにより、ビザンチン障害を起こしたレプリカを検出できる。よって、ビザンチ

ン障害に頑強である。全レプリカが同じ計算を行なうために、 P_i の計算は、決定論的である必要がある。

(2) 受動的多重化

入力を受け、計算を行ない、出力するレプリカ P_{i1} を指揮レプリカとする。受動的多重化では、指揮レプリカ P_i のみ入力を受け、計算し、出力を行なう[図9.3-(2)]。他のレプリカ $P_{i2}, \dots, P_{im_i}$ は、何も行わない。他のレプリカ P_{ij} の内部状態は、指揮レプリカ P_{i1} がチェックポイントを取り、このときの内部状態を P_{ij} に送信することによって更新される。 P_{i1} が障害を起こしたとき、他のレプリカの1つが指揮レプリカとなり、最新のチェックポイントから計算を再開する。このため復旧に時間がかかる。受動的多重化では、指揮レプリカの停止障害のみ対処できる。しかし、 P_{i1} がビザンチン障害を起こしたときには、これを検出できない。また、指揮レプリカのみ計算を行い出力を行うので、指揮レプリカの計算は決定論的である必要がない。

(3) 準能動的多重化

入力を受け、計算し、出力を行うレプリカ P_{i1} を指揮レプリカとする。準能動的多重化では、指揮レプリカ P_i のみ入力受け出力を行う[図9.3-(3)]。他のレプリカ $P_{i2}, \dots, P_{im_i}$ は、入力を受け、計算を行うが出力は行わない。他のレプリカの内部状態は、自らの計算によって更新されるか、チェックポイントによって指揮レプリカの内部状態と同じ状態に更新される。準能動的多重化では、受動的多重化と同様に指揮レプリカのビザンチン障害を検出できず、指揮レプリカの停止障害のみ適応できる。指揮レプリカの計算は、受動的多重化と同様に指揮レプリカのみ計算を行い、出力を行うため決定論的である必要がない。指揮レプリカが障害を起こしたとき、他のレプリカの1つが指揮レプリカとなり計算を継続する。他のレプリカも指揮レプリカと同じ入力を受け計算を行っている。このため、計算を中断せずに継続できる。よって、チェックポイントまで戻る必要がなく、復旧に時間がかからない。

プロセス P_i の計算量を考える。能動的多重化では、全レプリカ $P_{i1}, \dots, P_{im_i}$ が計算を行ない、全レプリカで入力と出力のための通信が行われる。受動的多重化では、指揮レプリカ P_{i1} のみが計算を行う。他のレプリカ $P_{i2}, \dots, P_{im_i}$ は、計算を行わない。準能動的多重化では全レプリカで計算が行われる。入力のための通信は全レプリカで行われるが、出力のための通信は、 P_{i1} でのみ行われる。ここで、以下を定義する。

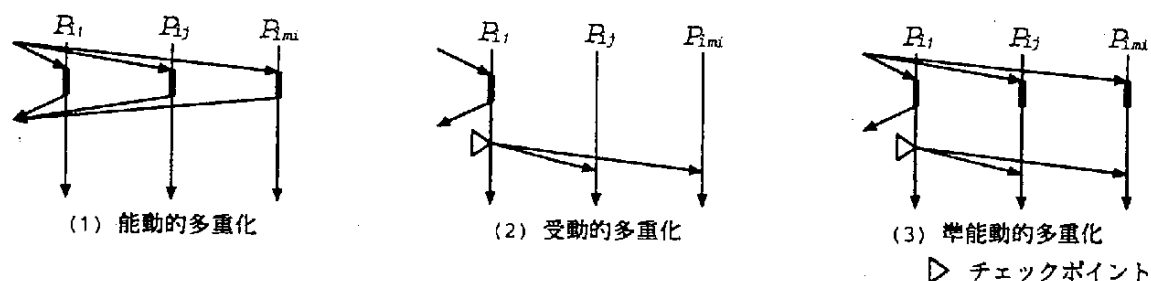


図9.3 多重化

[定義]

- (1) 各入力に対して各レプリカの計算量（時間）を T とする。
- (2) 受動的多重化と準能動的多重化でのチェックポイントは、 P_{i1} で k 回目の入出時に1回チェックポイントを取るとする。
- (3) 指揮レプリカ P_{i1} がチェックポイントを取るための計算時間を T_α とする。他のレプリカ $P_{i2}, \dots, P_{imi}$ がチェックポイントにより受信したデータを計算する時間を T_β とする。
- (4) レプリカがチェックポイントから再開する計算時間を T_γ とする。

1 回の入出力が行なわれた場合の、各多重化方法の全計算時間とメッセージ数、また、平均復旧時間とビザンチン障害に対処できるかを表1に示す。

表9.1 各多重化の特徴

多重化方法	全計算時間	メッセージ数	平均復旧時間	ビザンチン障害
能動的	Tm_i	$2m_i$	0	○
受動的	$T + \frac{1}{k}(T_\alpha + T_\beta(m_i - 1))$	$2 + \frac{1}{k}(m_i - 1)$	$\frac{k}{2}T_\gamma$	×
準能動的	$Tm_i + \frac{1}{k}(T_\alpha + T_\beta(m_i - 1))$	$(m_i + 1) + \frac{1}{k}(m_i - 1)$	0	×

例として、 $m_i=5$ 、 $T=1$ 、 $T_\alpha=0.04$ 、 $T_\beta=0.04$ 、 $T_\gamma=0.002$ とし、 k の値を変化させ、能動的多重化に対する各多重化方法のメッセージ数と全計算時間の比率を示す[図9.4]。図9.4より、能動的多重化が最もメッセージ数が多く、受動的多重化が最もメッセージ数が少な

いことが分かる。また、能動的多重化や準能動的多重化に比べ、受動的多重化の計算時間が少ない。能動的多重化と準能動的多重化では、チェックポイントの計算時間分だけ、準能動的多重化の方が多いが k の値が大きい場合差が少ない。

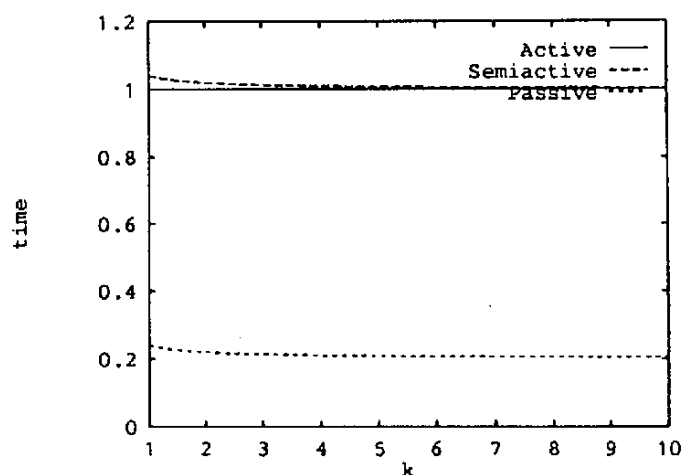
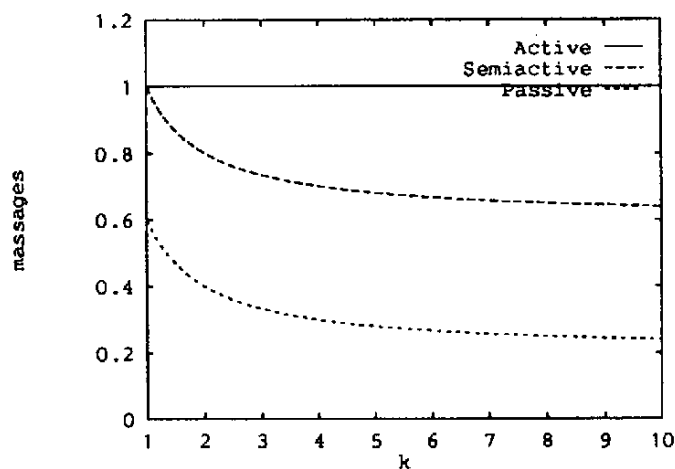


図9.4 能動的多重化に対する、メッセージ数 $\left(\frac{1}{2m_i}\right)$ と計算時間 $\left(\frac{1}{Tm_i}\right)$ の比率

10章 複合多重化

以下では、能動的、準能動的、受動的の多重化を組み合わせた複合的多重化を考える。

10.1 レプリカグループ

レプリカの障害に対して、以下の目的を達成する多重化について考える。

- (1) レプリカの障害に対して論理プロセス P_i の計算の中断、後戻りがない。
- (2) メッセージ数とレプリカの全計算時間を最小化する。
- (3) ビザンチン障害に対する頑強性を提供する。

このために、各プロセス P_i を以下のように多重化することを考える。

- (1) ビザンチン障害に対処するために、複数の指揮レプリカを設ける。これらを能動的に多重化し、同一の計算を行わせる。指揮レプリカが障害しても残りのレプリカが計算を継続できる。また、指揮レプリカが、ビザンチン障害を起こした場合でも、全ての指揮レプリカの出力を比較することにより、障害したレプリカを検出できる。
- (2) 通信コストを減少させるために、障害時に昇格させるレプリカを候補レプリカと関連レプリカとに分ける。候補レプリカは、指揮レプリカが障害した場合に指揮レプリカとなる。候補レプリカは、直ちに、指揮レプリカとなれるように準能動的に多重化する。関連レプリカは受動的に多重化する。

以上から、論理プロセス P_i は、指揮レプリカ C_i 、候補レプリカ D_i 、関連レプリカ E_i の3つのレプリカの部分グループから構成される[図10.1]。 $(P_i = \langle C_i, D_i, E_i \rangle)$ 。

P_i において同時に障害するレプリカの数の上限值 f が定まっていると仮定する。 $2f+1$ 個の指揮レプリカがあれば、多数決を行うことによりビザンチン障害を起こしたレプリカを検出できる。また、候補レプリカは、 f 個用意することによって常に指揮レプリカ数を一定に保つことができる。

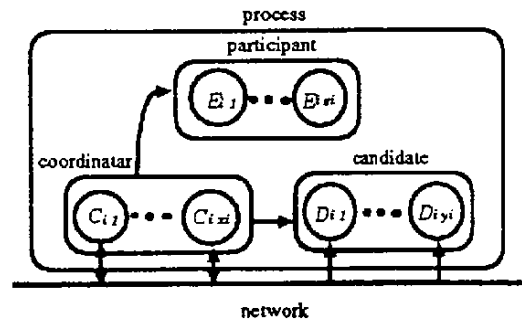


図10.1 レプリカグループ

(1) 指揮(coordinator)レプリカグループ

$C_i = \{C_{i1}, \dots, C_{ixi}\} (xi \geq 2fi+1)$ 。

入力を受け、計算を行い出力を行うレプリカのレプリカグループ。

(2) 候補(candidate)レプリカグループ

$D_i = \{D_{i1}, \dots, D_{iyi}\} (yi \geq fi)$ 。

入力を受け、計算を行うが、出力を行わないレプリカのレプリカグループ。

(3) 関連(participant)レプリカグループ

$E_i = \{E_{i1}, \dots, E_{ixi}\} (zi \geq 0)$ 。

入力、計算、出力を行わないレプリカのレプリカグループ。

10.2 指揮レプリカグループ

指揮レプリカ C_0 は、入力を受け、計算を行ない、出力を行う。すなわち、クライアントからの SQL を受け付け、実行し、結果を返す。指揮レプリカ C_0 は、チェックポイント CK_0 を取る。チェックポイントでは、 C_0 の内部状態がログに記憶される。指揮レプリカが候補レプリカグループと関連レプリカグループに CK_0 での内部状態を送信する。

指揮レプリカ $C_{i1}, \dots, C_{ixi}$ の障害を考える。 C_0 が障害を起こしたとする。 C_0 の停止障害は、一定時間以上応答がないことにより検出される。 C_0 のビザンチン障害について考える。ある C_0 が、他のレプリカ $C_{i1}, \dots, C_{ixi}$ と異なった出力と状態を持つとき、 C_0 がビザンチン障

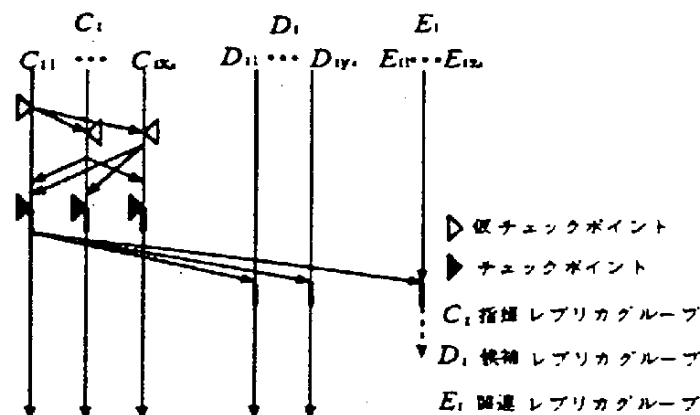


図10.2 チェックポイントの手順

害したとみなせる。

$C_{i1}, \dots, C_{i\alpha}$ 間で、チェックポイントをとるための手順を以下に示す[図10.2]。

[チェックポイント手順]

- (1) 1つの指揮レプリカ C_{i1} が他の指揮レプリカ $C_{i2}, \dots, C_{i\alpha}$ にチェックポイントをとる要求を送る。
- (2) 他の指揮レプリカ C_{ij} は仮チェックポイント TC_{ij} を取る。 TC_{ij} を取れたならば、全指揮レプリカにYESを送信する。 TC_{ij} を取れないならば全指揮レプリカにNOを送信する。
- (3) 各 C_{ij} は、全指揮レプリカからメッセージを受信する。メッセージの過半数がYESの場合を考える。 C_{ij} が、YESを送信していれば、 TC_{ij} をチェックポイント CK_{ij} とし、全候補レプリカと全関連レプリカに内部状態を送信する。また、NOを送信した指揮レプリカ C_{ik} を障害とみなし切り離す。次に、多数決した結果がNOのとき、全指揮レプリカは、 TC_{ij} を消す。
- (4) CK_{ij} を取れなかったならば、チェックポイントをとることを再び試みる。

10.3 候補レプリカグループ

候補レプリカ D_{ij} は、入力を受け、計算を行うが出力を行わない。 D_{ij} の内部状態は、指揮レプリカからのチェックポイントによって、指揮レプリカの内部状態に更新されるか、入力をうけ計算を行うことによって、状態を更新する。候補レプリカは、指揮レプリカと同じく、入力を受け計算を行うため内部状態が指揮レプリカと同じである。そのため指揮レプリカに障害が発生した時、候補レプリカが指揮レプリカに直ちに昇格して、その障害を起こした指揮レプリカの引き継ぎを行う。

10.4 関連レプリカグループ

関連レプリカ E_{ij} は計算を行わない。 E_{ij} の内部状態は、チェックポイントによって指揮レプリカの内部状態に更新される。

10.5 複合的多重化のモデル

複合的多重化では、指揮、候補、関連レプリカがどのように通信を行うかが問題となる。この通信方式について考える。

10.5.1 モデル1（クライアント依存型）

手順を以下に示す[図10.3]。

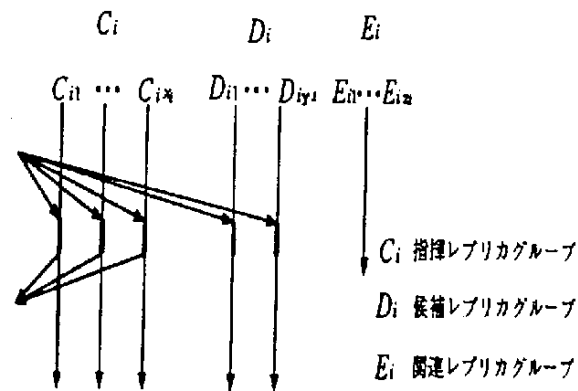


図10.3 モデル 1

[手順]

- (1) クライアントは、メッセージを全指揮レプリカと全候補レプリカに送信する。
- (2) 全指揮レプリカと全候補レプリカは、計算を行う。
- (3) 全指揮レプリカは、出力をクライアントに送信する。

これをクライアント依存型モデルとする。

クライアントは、全ての指揮レプリカからの出力の多数決を行う。 $f+1$ 以上のレプリカと異なった出力を行ったレプリカを障害したとする。クライアントは、障害したレプリカを切り離すことを指揮レプリカに通達する。

モデル1の特徴として、以下がある。

利点

1. メッセージ数が少ない。
2. 指揮レプリカにプロセス障害が発生しても、それに対処できる。

欠点

1. クライアントが障害の検出を行うため、障害した指揮レプリカの切り離しに時間がかかる。

10.5.2 モデル2 (集中型)

手順を以下に示す[図10.4]。

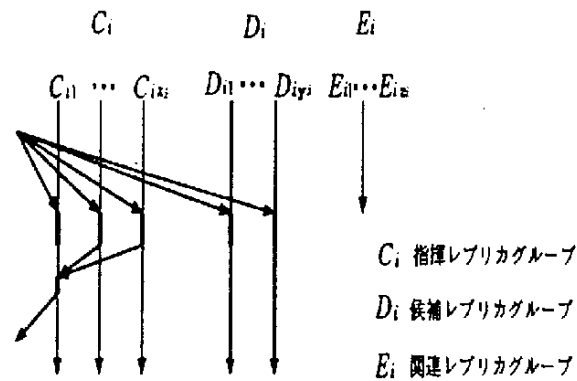


図10.4 モデル2

[手順]

1. クライアントは、メッセージを全指揮レプリカと候補レプリカに送信する。
2. 全指揮レプリカと全候補レプリカは、計算を行なう。
3. 他の指揮レプリカ $C_{i2}, \dots, C_{ik_i}$ は、出力を指揮レプリカ C_{i1} に送信する。
4. 指揮レプリカ C_{i1} は、受信した出力の多数決を行ない障害したレプリカを検出する。

障害したレプリカは切り離される。

5. 指揮レプリカ C_{i1} は、出力をクライアントに送信する。

これを集中型モデルとする。

モデル2の特徴を以下に示す。

利点 メッセージ数が少ない。

欠点 指揮レプリカ C_{i1} に障害が発生するとプロセスが機能しなくなる。

10.5.3 モデル3 (分散型)

手順を以下に示す[図10.5]。

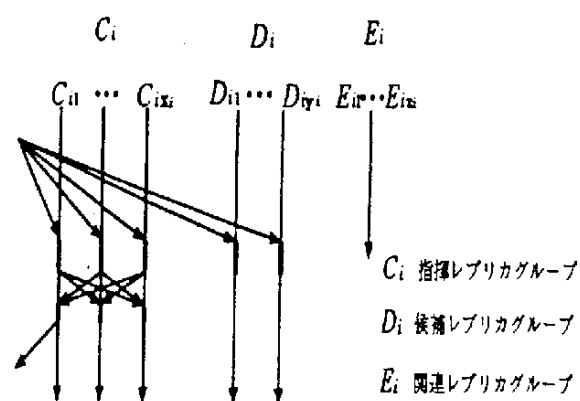


図10.5 モデル3

【手順】

1. クライアントは、メッセージを全指揮レプリカと候補レプリカに送信する。
2. 全指揮レプリカと全候補レプリカは計算を行う。
3. 全指揮レプリカは、自分以外の指揮レプリカに出力を送信する。
4. 各指揮レプリカは、受信した出力の多数決を行ない障害したレプリカを検出する。
障害したレプリカは切り離される。
5. $f+1$ 個の正常な指揮レプリカが、クライアントに出力を送信する。ここで、 f は障害発生数である($f_i \leq f$)。

これを分散型モデルとする。

モデル3の特徴として以下がある。

利点 指揮レプリカにプロセス障害が発生しても、対処できる。

欠点 メッセージ数が多い。

10.5.4 モデル4 (メイン指揮レプリカ型)

モデル2では、1つの指揮レプリカのみ多数決と出力を行うため、そのレプリカが障害してしまうと機能が停止してしまう。そのため、多数決と出力を指揮レプリカを複数作成することによって障害に対する頑強性を持たせるモデルを考える。多数決と出力を行うレプリカをメイン指揮レプリカ $[C_{i1}, \dots, C_{imi}](x_i \geq w_i)$ とし指揮レプリカ内に設ける

手順を以下に示す[図10.6]。

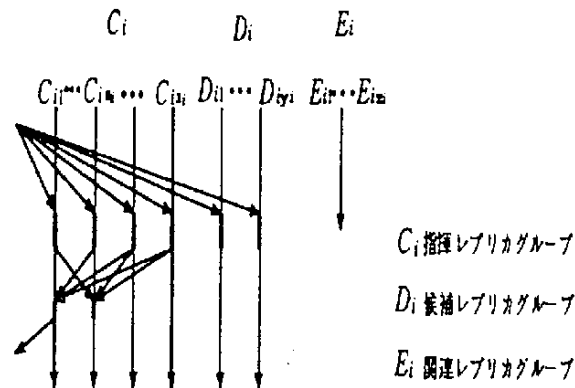


図10.6 モデル 4

[手順]

1. クライアントは、メッセージを全指揮レプリカと候補レプリカに送信する。
2. 全指揮レプリカと全候補レプリカは計算を行う。
3. 全指揮レプリカは、メイン指揮レプリカに出力を送信する。
4. 各メイン指揮レプリカは、受信した出力の多数決を行ない障害したレプリカを検出する。障害したレプリカは切り離される。
5. $f+1$ 個の正常な指揮レプリカが、クライアントに出力を送信する。

これをメイン指揮レプリカ型モデルとする。

モデル 4 の特徴として以下がある。

利点 指揮レプリカにプロセス障害が発生しても対処できる。

欠点 メッセージ数が多い。

10.5.5 レプリカの昇格

レプリカが障害を起こした時の昇格方法を述べる。指揮レプリカ、候補レプリカや関連レプリカが障害を起こした時、各レプリカは障害を起こしたレプリカとして、各レプリカグループより切り離される。そして、候補レプリカが指揮レプリカに、関連レプリカが候補レプリカに昇格する。候補レプリカは、指揮レプリカと同じ入力を受け計算を行っているため、障害を起こした指揮レプリカを切り離すと同時に昇格することができる。関連レ

プリカを候補レプリカに昇格させるときは、他の候補レプリカの内部状態と異なるため、最新のチェックポイントから計算を再開し内部状態を合わせる。障害を起こしたレプリカは、復旧可能ならば関連レプリカに昇格させる。復旧不可能ならば、新しいレプリカのクローンを作成し(cloning)関連レプリカにする[図10.7]。

指揮レプリカが f 個($f \leq f$)障害した場合を考える。正常な指揮レプリカの1つが f 個の候補レプリカと関連レプリカに昇格要求メッセージを送信する。メッセージを受信した候補レプリカは、メッセージを受信したことを指揮レプリカに送信する。また、メッセージを受信した関連レプリカは、候補レプリカと同様に、メッセージを指揮レプリカに送信する。次に、昇格中の関連レプリカは指揮レプリカの状態に自らの状態を更新する。また、指揮レプリカと同様に入力を受けそれを計算し候補レプリカに昇格する。指揮レプリカは、候補レプリカと関連レプリカからの受信メッセージを受信しなければ、そのレプリカを停止障害を起こしているとみなし、そのレプリカをレプリカグループより切り離す。

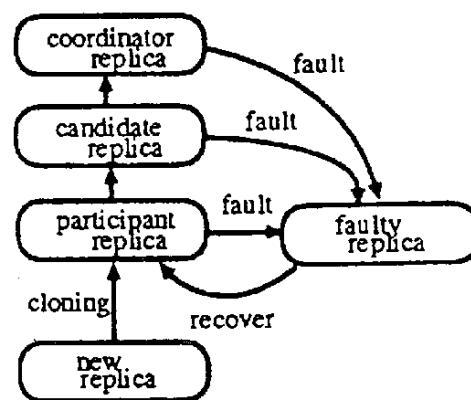


図10.7 レプリカノ昇格

10.5.6 評価

複合的多重化の提案したモデルについて評価する。

複合多重化方式をメッセージ数、計算時間、メッセージ数について評価を行う。各 P_i について、以下のパラメータを考える。

T =入力に対する各レプリカの計算時間。

k =チェックポイントを取る割合。即ち、 k 回の入出力毎にチェックポイントを取る。

T_α =指揮レプリカがチェックポイントを取るための計算時間。

T_β =他のレプリカがチェックポイントにより受信したデータを計算する時間。

T_γ =関連レプリカが候補レプリカに昇格する平均復旧時間。

T_δ =レプリカが受信した出力を多数決し、障害したレプリカを検出するための計算時間。

T_θ =障害と障害の平均故障間隔時間。

各 p_i に対して、以下を考える。

x_i : 指揮レプリカ数。

y_i : 候補レプリカ数。

z_i : 関連レプリカ数。

$$m_i = x_i + y_i + z_i$$

w_i : メイン指揮レプリカ数。

f_i : P_i で同時に障害するレプリカ数の上限値。

f : P_i で同時に障害するレプリカ数。($0 \leq f \leq f_i$)

1 回の入出力と計算が行われた場合のメッセージ数と計算時間を表10.1に示す。

表10.1 各モデルのメッセージ数と計算時間

モデル	メッセージ数	計算時間
モデル 1	$2x_i + y_i$ $+\frac{1}{k}(x_i^2 - x_i + y_i + z_i)$	$(x_i + y_i)T$ $+\frac{1}{k}(x_i T_\alpha + (y_i + z_i)T_\beta)$
モデル 2	$2x_i + y_i$ $+\frac{1}{k}(x_i^2 - x_i + y_i + z_i)$	$(x_i + y_i)T + T_\delta$ $+\frac{1}{k}(x_i T_\alpha + (y_i + z_i)T_\beta)$
モデル 3	$x_i^2 + y_i + 2f + 1$ $+\frac{1}{k}(x_i^2 - x_i + y_i + z_i)$	$((x_i + y_i)T + x_i T_\delta$ $+\frac{1}{k}(x_i T_\alpha + (y_i + z_i)T_\beta)$
モデル 4	$x_i w_i + x_i + y_i - w_i + 2f + 1$ $+\frac{1}{k}(x_i^2 - x_i + y_i + z_i)$	$((x_i + y_i)T + w_i T_\delta$ $+\frac{1}{k}(x_i T_\alpha + (y_i + z_i)T_\beta)$

ここで、例として、 $x_i=2f_i+1$ 、 $y_i=f_i$ 、 $z_i=1$ 、 $w_i=f_i+1$ 、 $f=0$ 、 $k=10$ 、 $T=1$ 、 $T_a=0.04$ 、 $T_p=0.04$ 、 $T_s=0.002$ とし、 f_i を変化させた時の各モデルのメッセージ数と全計算時間を示す[図10.8]。

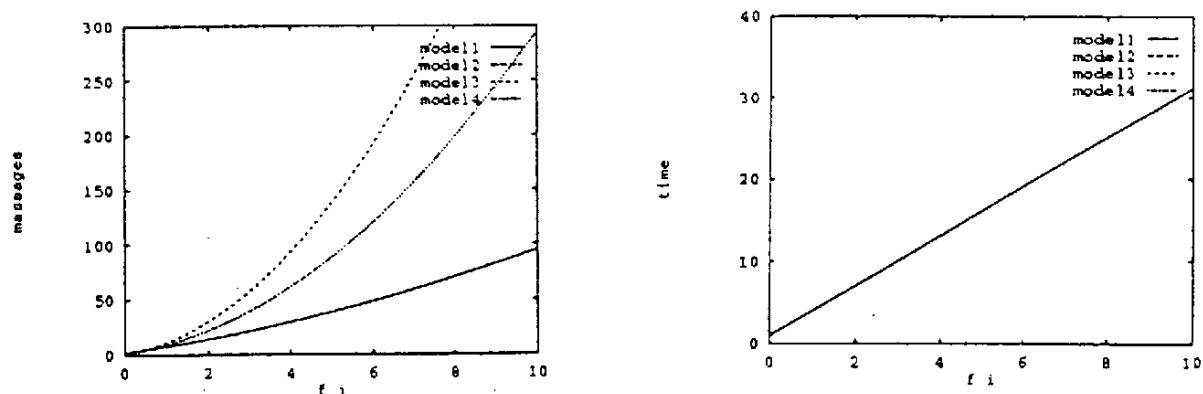


図10.8 各モデルのメッセージ数と計算時間

図10.8より、モデル1とモデル2がメッセージ数が少なく、モデル3がメッセージ数が多い。モデル4のメッセージ数はその中間である。モデル1とモデル2のメッセージ数は同じである。また、各モデルでの計算時間の差はあまりみられない。これは、 T_s の値が他の計算時間に比べ小さいからである。

10.6 各モデルの考察

モデル1は、クライアントが出力を多数決し、障害した指揮レプリカを検出するため、障害した指揮レプリカをレプリカグループより切り離すのに時間がかかる。このため、このモデルは適していない。

モデル2は、指揮レプリカ C_n に多数決と出力を行うため、メッセージ数を減少させることができたが、 C_n に障害が発生した時にシステムが停止してしまう。このモデルも適していない。

モデル3とモデル4は、ビザンチン障害に対処できる。しかし、共にメッセージ数が多い。

モデル4のメイン指揮レプリカ数 w_i を指揮レプリカ数 x_i と等しくすると、モデル4はモデル3と同じになる。 $w_i \leq x_i$ より、モデル4はモデル3より計算時間において $(x_i - w_i)T_s$ 、メッセージ数において $(x_i - w_i)(x_i - 1)$ 少ない。

よって、モデル4が提案したモデルの中で最も適しているといえる。

10.7 モデル3、モデル4と能動的多重化の比較

$x_i=4$ 、 $y_i=2$ 、 $z_i=1$ 、 $w_i=2$ 、 $k=100$ 、 $f=0$ 、 $T=1$ 、 $T_s=0.04$ 、 $T_p=0.04$ 、 $T_s=0.002$ として、モデル3、モデル4と能動的多重化の計算時間、メッセージ数を比較する[図10.9]。

図10.9より、モデル4は能動的多重化よりメッセージ数、計算時間が少ない。モデル3とモデル4の計算時間の差は $(x_i - w_i)T_s$ であり、図10.9ではその差がみられないほど小さい。

以上より、モデル4が提案したモデルの中で最適であり、かつ、能動的多重化に比べてメッセージ数と計算時間が少ないことが分かった。

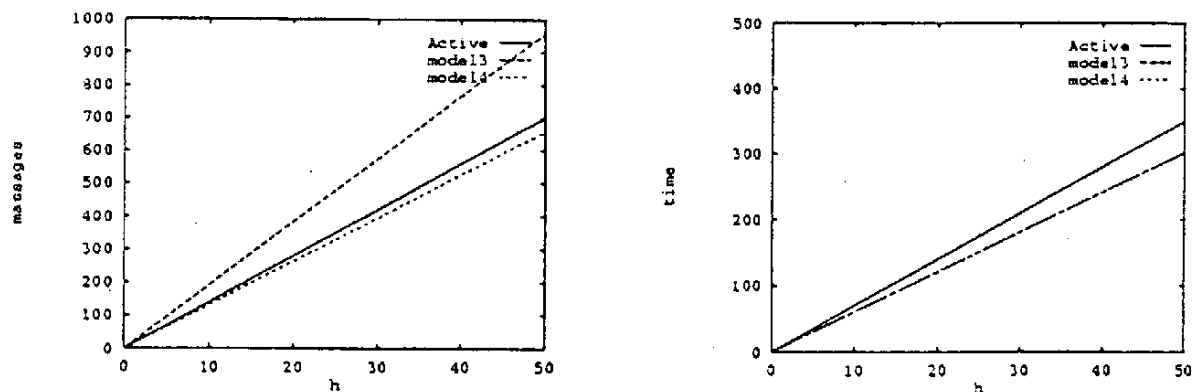


図10.9 メッセージ数と計算時間

第11章 今後の課題

本研究では、光ファイバ等を用いた高速通信ネットワークにより、比較的低信頼なワークステーションを相互接続した分散システムにより、複数のデータベースサーバをどのように高信頼なものとしていくかについて論じた。今後の課題として、以下がある。特に、本報告では、データベースサーバが停止障害するだけでなく、ビザンティン障害を起こす場合についても、システム全体を頑強とする方法について論じた。

1. データベースサーバのレプリカ間の通信方法。ビザンティン障害に対処するためには、データベースサーバを多重化し、これらから送信されてくるメッセージを比較する必要がある。このときの通信では、本報告で述べたように、多くのメッセージが送信される。今後、メッセージ数をより少なくするレプリカ間の通信方式を研究する必要がある。
2. 障害レプリカの診断方法。障害したレプリカをどのように診断し、検出していくかが課題となる。診断には、同じ操作を複数のレプリカに行わせ、その結果を比較する方法がある。データベースサーバに対する演算、例えば、検索演算では、大量のデータが結果として得られる場合がある。これを有効にどのように比較していくかが問題となる。
3. 移動型システムの高信頼化。今後の情報システムでは、従来のデータベースサーバとクライアントのように、通信ネットワーク内の固定の位置に存在するものだけでなく、通信ネットワーク内を移動するサーバとクライアントを考える必要がある。

これらの課題の中で、特に、複数の移動型のデータベースサーバとクライアントを通信ネットワークで相互接続し、信頼性のあるデータベースサービスを提供していくことが重要な課題である。PHS等の移動型の端末の普及により、各個人は任意の場所から通信ネットワーク内の任意のデータベースサーバを利用できる。また、データベースサーバも、固定の位置のものだけでなく、飛行機、自動車、個人等の移動体内に存在しうる。こうした移動体システムを基本として、今後の情報システムは、構築されていくものと考えられる。このとき、移動体内のデータベースサーバの信頼性と可能性をどのように保っていくかは、重要な課題となる。

参考文献

- [BARB93] Barborak, M. and Malek, M., "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, Vol.25, No.2, 1993, pp.171-220.
- [BERN87] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison Wesley*, 1987.
- [BEER89] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," *JACM*, Vol.36, No.2, pp.230-269, 1989.
- [BERN87] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison Wesley*, 1987.
- [CASA93] Casavant, T. L. and Singhal, M., "Distributed Computing Systems," *IEEE Computer Society Press*, 1993.
- [CERI84] Ceri, S. and Pelagati, G., "Distributed Databases: Principles and Systems," *McGraw Hill*, 1984.
- [CHAND83] Chandy, K. M., Misra, J., and Haas, L. M., "Distributed Deadlock Detection," *ACM Trans. Computer Systems*, Vol.1, 1983, pp.144-156.
- [CHAND85] Chandy, K. M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM TOCS*, Vol.3, No.1, 1985, pp.63-75.
- [Codd70] Codd, E. F., "A Relational Model for Large Data Banks," *CACM*, Vol.13, No.6, 1970.
- [DATE87] Date, C. J., "A Guide to THE SQL STANDARD," *Addison-Wesley*, 1987.
- [ESWA76] Eswaren, K. P., Gray, J., Lorie, R. A., and Traiger, I. L., "The Notion of Consistency and Predicate Locks in Database Systems," *CACM*, Vol.19, No.11, pp.624-637, 1976.
- [FISC85] Fischer, M. J., Lynch, N. A., and Paterson, M. S., "Impossibility of Distributed Consensus with One Faulty Process," *JACM*, Vol.32, No.2, 1985, pp.374-382.

- [GRAY78] Gray, J., "Notes on Database Operating Systems," *Lecture Notes in Computer Science*, No.60, 1978, pp.393-481.
- [HEIM85] Heimbigner, D. and McLeod, D., "A Federated Architecture for Information Management," *ACM Trans. on Office Information Systems*, Vol.3, No.3, 1985, pp.253-278.
- [GRAY86] J. グレイ, 渡辺栄一, "フォールト・トレラント・システム", マグロウヒルブック, 1986.
- [HEIM85] Heimbigner, D. and McLeod, D., "A Federated Architecture for Information Management," *ACM Trans. on Office Information Systems*, Vol.3, No.3, 1985, pp.253-278.
- [KNAP87] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol.19, No.4, 1987, pp.303-328.
- [LAMP78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Vol.21, No.7, 1978, pp.558-564.
- [LAMP82] Lamport, L. and Shostak, R., "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems*, Vol.4, No.3, 1982, pp.382-401.
- [MULL89] Mullender, S., "Distributed Systems," *ACM Press*, 1989.
- [ORAC92] "ORACLE7 Relational Database Management System: Cooperative Server Technology," *Oracle*, 1992.
- [SYBA90] Sybase, "Transact-SQL User's Guide," *Sybase*, 1990.
- [OSZU91] Ozsú, M. T. and Valduriez, P., "Principles of Distributed Database Systems," *Prentice-Hall*, 1991.
- [POWE94] Powell, D., Barrett, P., Bonn, G., Chereque, M., Seaton, D., and Verissimo, P., "The Delta-4 Distributed Fault-Tolerant Architecture," in *Readings in Distributed Computing Systems*, *IEEE Computer Society Press*, 1994, pp.223-248.
- [SHET90] Sheth, A. and Larson, J. A., "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, Vol.22, No.3, 1990, pp.183-235.

- [SKEE81] Skeen, D., "Nonblocking Commit Protocols," *Proc. of the ACM SIGMOD*, 1981.
- [SKEE83] Skeen, D. and Stonebraker, M., "A Formal Model of Crash Recovery in Distributed System," *IEEE Trans. Software Eng.*, Vol. SE-9, No.3, 1983.
- [SYBA90] Sybase, "Transact-SQL User's Guide," Sybase, 1990.
- [TAKI92] 滝沢誠, "データベースシステム入門技術解説," ソフト・リサーチ・センター, 1992.
- [TAKI83] Takizawa, M., "Distributed Database System JDDBS," *JARECT*, Vol.7, *Computer Science and Technologies* (Kitagawa, T., ed.), Ohmsha and North-Holland, 1983.

—— 禁 無 断 転 載 ——

平成7年3月発行

発 行 財団法人 データベース振興センター
東京都港区浜松町二丁目4番1号
世界貿易センタービル7階
TEL 03 (3459)8581

委託先 株式会社シネジャーナルプロダクション
東京都新宿区四谷4-21-37
TEL 03 (3355)1594

印 刷 株式会社清光社クリエイト
東京都千代田区神田紺屋町34
TEL 03 (5296)2788

