NAIST-IS-DD0461024

博士論文

データベース接続層の拡張による ICT サービスの信頼性向上の研究

中村 暢達

2007 年 9 月 19 日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

中村 暢達

審査委員：

砂原 秀樹 教授	（主指導教員）
山口 英 教授	（副指導教員）
藤川 和利 准教授	（副指導教員）
河合 栄治 特任准教授	（副指導教員）
中山 雅哉 准教授	（東京大学）

データベース接続層の拡張による ICTサービスの信頼性向上の研究*

中村 暢達

内容梗概

現在の多くの企業活動，市民生活にとって，情報通信技術（ICT）サービスは不可欠であり，ICTサービスシステムの信頼性を最大化することが求められている．信頼性向上のために，システムの構成要素を多重化し，障害発生時に，同等の機能を有する構成要素で代替させることが基本的な対処であり，構成要素の多重化管理が重要となる．

従来の多重化管理は，ストレージもしくはデータベースのレイヤで行われるものであるが，複製の制御は固定的であり，多様かつダイナミックに変化するサービスの信頼性要件に対応することは困難であった．また，データベースの多重化では，サーバ負荷が増大し，性能劣化が大きいという問題があった．

本論文では，このような課題に対し，ICTサービスのシステム構成の多くが，サービスとデータを分離，つまりアプリケーション・サーバとデータベース・サーバとから構成されることに注目し，その汎用的なインタフェースとなっているデータベース接続層を拡張することで，データベース多重化構成を管理する方式を提案する．提案方式は，以下の特長を有する．

1. 多重化管理は，アプリケーション・サーバにおいて実行されるので，データベース・サーバの負荷を増大させることはない．同時接続数が増えた場合には，拡張したデータベース接続層が，データベースへのアクセスを平準化するため，性能低下は発生しない．

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD0461024, 2007年9月19日.

2. アプリケーションの多くは，標準的なデータベース接続 API を利用しており，この API を提供しているライブラリを，本研究で開発したものと置き換え，冗長構成部と接続することで，容易に冗長構成に変更できる．つまり，既存アプリケーションのプログラムソースの変更が不要である．また，データベースのアクセスログをデータベース接続層でバッファ管理することによって，冗長のデータベースの接続，および切り離しを，サービスを停止させることなく行うことができる．

3. データベース接続層で交換されるデータは，一般的に SQL 形式であるので，処理種別などの監視・可視化が容易である．その監視結果に応じて，複製処理を柔軟に制御，あるいはログを柔軟にフィルタリングすることを可能とする．サービスの信頼性要件は多様であるが，その要件に応じて，必要最小限のデータベース処理要求のみを冗長構成部に送るように設定することで，信頼性を確保しつつ，処理性能を最大化することが可能となる．

上記に示した方式を，Java のデータベース接続層である JDBC に適用し，ベンチマークソフトウェア等を利用して，動作評価および機能評価を行った．従来のデータベース複製方式と比べ，提案方式は処理性能に優れ，また多様な信頼性要件に応じての複製制御が可能であることが確認できた．

キーワード

信頼性，耐障害，データベース接続，トランザクション，運用管理

Studies on System Dependability Improvement with Extended Database Connection Layers for ICT Services *

Nobutatsu Nakamura

Abstract

Information and Communication Technology (ICT) Services are indispensable to many social activities at present and it is required to maximize ICT system dependability. In order to improve the dependability, the systems should have redundant hardware/software modules. Once any failure has occurred, they would make failover using the modules. Therefore, the system layer key function is to manage the redundant modules.

Many existing methods are managed at either storage layer or database layer. Their replications are fixed and cannot be controlled corresponding to various and dynamic dependability requirements of current ICT services. In database replication, there is another problem of performance degradation due to the server load increase.

In this paper, I propose a redundant module management technology using an extended database connection layer that is a standard interface between application servers and database servers in common ICT system configurations. The proposed technology has following features;

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0461024, September 19, 2007.

1. Since the redundancy is managed at an application layer, the database server load doesn't increase. While the number of simultaneous connections increases, performance doesn't degrade because my extended database connection layer controls database accesses at application servers.

2. As many applications are using standard database connectivity APIs, the library providing this APIs can be replaced to the newly developed library. Then, upgrading redundant system configuration isn't required to change any application program source. In addition, since database access logs are buffered and managed at the database connectivity layer, a redundant database server can be both attached to and detached from the system without stopping a service.

3. The exchanged data through database connectivity library is SQL and can be monitored easily. The proposed technology enables flexible redundancy management control using the monitored situation. As there are various dependability and performance requirements for ICT service system, the minimum requests are transmitted to the redundant modules corresponding to the requirements. Then, the system performance can be maximized under guarantee of system dependability.

In order to evaluate the proposed method, I extended JDBC; database connection layer implementation in JAVA. I compared the extended JDBC with existing methods using benchmark programs and confirmed good performance and flexible replication for various dependability requirements.

Keywords:

dependability, fault-tolerance, database connectivity, transaction, system management

目 次

1. 緒言 **1**

1.1 研究の背景 1
1.2 研究目的 2
1.2.1 システム構築フェーズの観点 2
1.2.2 地域分散性の観点 3
1.2.3 信頼性観点 4
1.3 解決への基本戦略 7
1.3.1 運用管理の自動化 7
1.3.2 多重化構成 9
1.4 本論文の構成 12

2. 多重化構成技術 **14**

2.1 障害対処 14
2.1.1 障害とは 14
2.1.2 耐障害とは 15
2.2 耐障害のための基本的な多重化手法 17
2.2.1 多重化アーキテクチャ 17
2.2.2 システム/ソフトウェアの多重化 21
2.3 ストレージの多重化 23
2.3.1 ストレージアクセス 24
2.3.2 ドライバレイヤでの多重化 26
2.3.3 ネットワークレイヤでの多重化 28
2.4 データベース・サーバの多重化 29
2.5 アプリケーションの多重化 31
2.6 多重化構成技術のまとめ 33

3. データベース接続層の拡張によるデータベース複製方式 **36**

3.1 データベース複製 37

3.1.1 多重化管理位置の定性的検討 37
3.1.2 多重化管理位置の定量的検討 39
3.1.3 データベース接続層における多重化の特徴 43
3.2 無停止フェイルバック 46
3.2.1 フェイルオーバー 47
3.2.2 短時間停止における無停止フェイルバック 47
3.2.3 長時間停止における無停止フェイルバック 48
3.3 基本試作 49
3.4 基本動作 52
3.4.1 正常時の動作：データ複製 53
3.4.2 障害時の動作：サービスの継続 56
3.5 負荷分散構成 60
3.5.1 順序問題 60
3.5.2 従来方式 63
3.5.3 提案方式：セマフォサーバ 65
3.6 負荷分散構成における性能評価 68

4. ディザスタリカバリ機構への適用 **72**
4.1 遠隔地への同期複製 72
4.2 遠隔地への複製の課題と解決方針 73
4.3 ディザスタリカバリ向けデータベース複製システム 74
4.3.1 システム構成概要 74
4.3.2 データベース複製処理 76
4.3.3 データベースアクセス状況 78
4.3.4 複製処理時の障害 78
4.4 基本動作 80
4.4.1 フェイルオーバー 80
4.4.2 不整合回避 81
4.5 遠隔複製評価実験 83
4.5.1 遠隔複製機構の実装 83

4.5.2 遠隔複製実験システム 84
4.5.3 実験結果 86
4.6 ディザスタリカバリ機構への適用のまとめ 88

5. SVMを用いたデータベース処理要求の複製処理制御 89
5.1 データベース処理要求の複製処理制御 89
5.2 運用管理の自動化に向けての課題 90
5.2.1 ルールベース運用管理 90
5.2.2 運用管理における学習技術の利用 91
5.3 学習型運用管理方式のデータベース複製機構への適用 93
5.3.1 学習型運用管理 93
5.3.2 複製処理のタイミング制御 93
5.3.3 データベース処理要求間隔時間の推定 96
5.4 方式検証実験および結果 100
5.4.1 予備実験 100
5.4.2 アプリケーション実験 105
5.4.3 予測閾値によるシステム制御 109
5.4.4 学習効果の評価 112
5.5 SVMを用いたデータベースアクセスのタイミング制御のまとめ . 112

6. データベース置き換えのためのデータベース接続層の拡張 114
6.1 データベース置き換え支援 114
6.1.1 背景 114
6.1.2 置き換え処理 115
6.2 データベース置き換えの課題と解決方針 116
6.2.1 SQL変換処理の位置 116
6.2.2 SQL変換ルール 117
6.2.3 SQL変換処理の検証 118
6.3 SQL 変換・検証機構の実装 122
6.4 SQL 変換・検証機構の評価 123

6.4.1 評価方法 123
6.4.2 評価結果 123
6.5 データベース置き換えのためのデータベース接続層の拡張のまとめ 124

7. アプリケーションの多重実行のためのデータベースアクセス管理 126
7.1 アプリケーション多重実行 126
7.2 データベースアクセス管理 126
7.3 データベースアクセス管理機構の評価 127
7.3.1 評価環境 128
7.3.2 実験方法 128
7.3.3 実験結果 129
7.4 アプリケーション多重実行に関する考察 131
7.4.1 環境多様型多重実行 131
7.4.2 操作ミスに対する対処 133
7.5 アプリケーションの多重化のまとめ 135

8. 結言 136
8.1 研究の成果 136
8.2 展望 137

謝辞 143

参考文献 145

付録 151

A. TPC-C 151
A.1 TPC-C とは 151
A.2 トランザクション詳細 152

B. SQL 変換・検証の評価アプリケーション 153

図 目 次

1 本論文の対象領域 3
2 Web 三層システムモデル例 5
3 多重化構成の分類 10
4 Fault，Error，Failure（[61] より） 15
5 デュアル構成 18
6 クラスタ構成 19
7 リカバリ・ブロック 22
8 N バージョン 23
9 Linux のファイルシステム構成 26
10 iSCSI システムの構成 27
11 データベース・サーバの多重化 30
12 （上）アプリケーション 1：データベース N 構成（3 章）と (下）アプリケーション N：データベース 1 構成（7 章） 35
13 多重化管理位置の検討 38
14 データ照会による複製検証 44
15 データベース処理ログによる複製検証 44
16 提案するデータベース接続層におけるデータベース複製 45
17 短時間停止における無停止フェイルバック 48
18 長時間停止における無停止フェイルバック 49
19 JDBC の構成 50
20 JDBC ゲートウェイ 51
21 基本動作確認実験環境 52
22 正常時のデータベース複製動作 54
23 データベース処理要求アクセスの様子 55
24 データベース接続エラー画面 56
25 該当データなし画面 57
26 障害時のサービス継続動作 59
27 データベース処理要求の順序問題（正常） 61

28 データベース処理要求の順序問題（異常） 62
29 プロキシ型順序制御 63
30 マルチキャスト型順序制御 64
31 セマフォサーバ 67
32 負荷分散構成評価実験構成 68
33 負荷分散構成評価実験結果 70
34 提案システム構成の概要 75
35 複製処理のアクセス手順 77
36 遠隔複製実験システム 85
37 遠隔複製実験結果 87
38 学習型運用管理方式 94
39 一般的なバッファデータ転送 95
40 改善したバッファデータ転送 95
41 アクセスログのベクトル変換 99
42 New-Order シナリオにおける予測精度の評価 102
43 New-Order シナリオにおける転送回数の評価 102
44 New-Order シナリオにおける RTO の評価 103
45 New-Order シナリオにおける RTO と転送回数との関係を示す図 . 104
46 アプリケーション実験における SLO（最大転送間隔）に対する予測精度 106
47 アプリケーション実験における SLO（最大転送間隔）に対する転送回数 106
48 アプリケーション実験における SLO（最大転送間隔）に対する RTO 107
49 アプリケーション実験における RTO と転送回数との関係 108
50 New-Order シナリオにおける予測閾値に対する転送回数，RTO の評価 110
51 アプリケーション実験における予測閾値に対する転送回数，RTO の評価 111
52 学習量およびテスト量に対する転送回数，RTO の評価 113

53 SQL 検証方式 119
54 データベース置き換え手順 121
55 データベースの重複更新 127
56 データベースアクセス管理 128
57 データベースアクセス管理実験システム構成 129
58 Web クエリの種類と提案方式によるオーバーヘッド処理時間 . . . 131
59 提案方式の基本構成 132
60 耐障害のシナリオ例 133
61 障害管理画面例 134

表 目 次

1 障害の分類 17
2 デュアル構成とクラスタ構成 20
3 Linux のファイルシステム 25
4 多重化管理位置の検討 32
5 多重化管理位置と障害復旧手順 40
6 多重化管理位置と可用性 42
7 負荷分散構成での性能評価実験結果 69
8 主データベース，予備データベース，クライアントの処理状態 . . 81
9 遠隔複製実験結果 86
10 アプリケーションログ（時刻とコマンドのみ抜粋） 97
11 学習量およびテスト量に対する予測精度，転送回数，RTO 112
12 有効メソッド 122
13 評価結果 124
14 各 Web クエリの処理時間 130

1. 緒言

1.1 研究の背景

近年の情報通信技術（ICT）の発展により，企業活動は大きく変化した．顧客，商品，社員の情報はデータベースに格納され，必要な情報は瞬時に参照可能となり，業務の連絡などのコミュニケーションの手段は電子メールやWebページが主流となっている．現在，約9割の企業が業務の効率化・迅速化を目的に，企業内ネットワークを構築している [57]．さらに，業務をグローバルな視点で最適化する目的で，国内，世界規模で，社内システムを連係，統合するようになりつつある．ICTによって，高度に企業活動は効率化されており，ICTを用いないということは，企業活動のスピード，コストパフォーマンスで他社に劣り，企業の競争力に大きな問題となる．

一方，市民生活においては，ICTは，商品のトレーサビリティ，交通事故防止，医療管理など，日常の生活において，安全・安心の社会生活を実現するものとして，大きな期待がある [56]．

こういったICTを利用した情報管理サービス，コミュニケーションサービス（ICTサービス）に企業活動，社会生活はますます依存するようになりつつあり，ICTサービスに不具合があると企業活動が停止し，市民生活にも大きな影響がでる．2007年5月に航空会社の予約搭乗手続きや手荷物管理を担当するチェックイン・システムに障害が起き，この障害のために，122便が欠航し，2万人が影響を受けたとされる．ひとたび，障害が発生すると，業務停止によるビジネス機会の損失だけでなく，企業の信頼性が損なわれ，顧客の企業離れを引き起こし，重大な損失となる可能性がある．最悪のケースでは，企業が存続しえないこともありえる [10]．

このように，ICTサービスは，企業活動，人々の日常生活に不可欠であり，さらなる企業活動の効率化，より豊かな生活の実現のためには，ICTサービスの信頼性向上は重要な課題である，そのため，システムの信頼性向上のために，幅広い取り組みがなされている．

従来，ハードウェア，ソフトウェアの信頼性向上に関しては，多くの研究がな

されてきており，冗長構成，フェイルオーバー機構，フェイルセーフ機構など多くの技術が製品設計，および製造工程で適用されてきている．しかしながら，製品の欠陥をゼロにすることは困難であり，またゼロに近づけることは手間と時間を必要とする．十分なテストがなされないまま，欠陥が含まれる可能性がある状態で出荷されるソフトウェアも少なくなく，出荷後にパッチによるソフトウェアのアップデートは頻繁にある．また，管理者，利用者の操作ミスも不可避である．ユーザインタフェースについても，多くの研究がなされており，操作ミスを最小化する取り組みがなされているが，完全に操作ミスをゼロにすることは困難である．他にも，ハードウェアの寿命，宇宙線による誤作動，自然災害も不可避であろう．

このような不可避の障害に対して，いかに対処し，サービスレベルを維持 (SLA: Service Level Agreement を遵守）するかが，大きな課題となっている．欠陥があることを許容できる耐障害性（Fault Tolerance）と，欠陥が顕在化してもすぐに修復できる回復性（Recoverability）に関する信頼性技術が重要であると考える．

1.2 研究目的

本研究の目標は，上述したようなサービスレベルの信頼性向上であるが,「信頼性」は幅広い意味がある．図 1 に本論文における議論の対象領域を示す．システムの構築フェーズの観点では，設計段階とする．地域分散性の観点では，ICT サービスを提供するサーバをターゲットとし，サーバがローカル（一サイト内での分散構成を含む），もしくは広域二地点分散とする．信頼性の観点では，可用性，保全性にフォーカスする．本節では，このような議論対象のフォーカスについて述べる．

1.2.1 システム構築フェーズの観点

システムの信頼性の要件定義は，どのような ICT サービスを対象とするかに依存する．本研究では，特に対象とする ICT サービスを定めず一般的な ICT サービスを対象とする．そして，信頼性の要件は与えられるものとする．

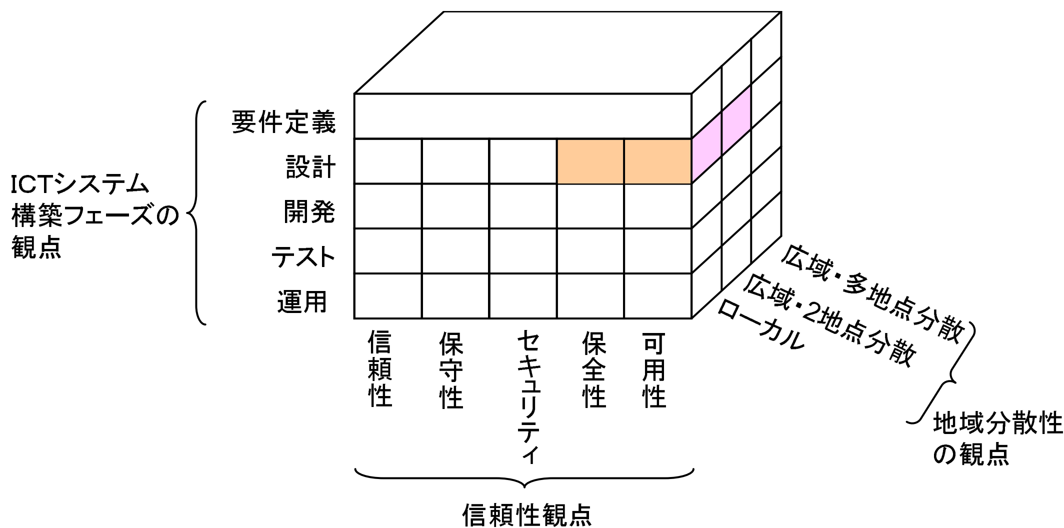


図 1 本論文の対象領域

本論文の議論の中心は，信頼性の要件が与えられた時に，どのようなシステムを設計すればよいか，そして，そのシステム開発を支援する基盤をどのように提供するか，ということにあり，信頼性向上を支援する基盤の提案をする.「開発」以降のシステム構築フェーズにおいても，多くの課題があるが，上記提案に直接関係する事項のみ言及する.

1.2.2 地域分散性の観点

ICT サービスの一般的なアーキテクチャは，図 2 のような三層モデル (3-tier system) である．利用者は，Web ブラウザを使い，ネットワークを介し，Web サーバとクライアント・サーバ通信を行う．ネットワークの途中には，ルータや負荷分散装置がある場合がある．Web サーバから，アプリケーション・サーバ，さらにはデータベース・サーバに要求は変換，転送され，処理結果を利用者の Web ブラウザに返す．運用管理者は，こういったサービスを実際に提供するシステムから管理情報を収集し，監視し，必要であれば，対処操作を行う.

サービス全体の信頼性向上には，利用者の端末，端末〜サーバ間のネットワーク

も大きな要因となるが，本論文では，サービスの根幹となるサーバ，つまりWebサーバ，アプリケーション・サーバ，データベース・サーバおよびストレージを議論対象とする．端末の利用者は一人であるが，サーバの利用者は全員であり，サーバの障害は全利用者に影響し，より高い信頼性が要求されるためである．

サーバの構成としては，一つのサイトのみで構成される場合が通常であるが，事業継続性の観点から，遠隔地にある複数のサイトから構成されることが望ましい．大企業であれば，地方に予備サイトを構成し，中小企業であれば，データセンタ事業者を活用し，アウトソーシングすることで予備サイトを構成する等が考えられる．最近では，国外やインターネットを利用して多地点に分散した予備体制を構築するケースも増えてきているが，東京・大阪のような2地点分散が地域分散の基本であり，本論文では2地点分散までを議論対象とする．広域・多地点分散については，今後の課題として，終章において簡単に言及する．

1.2.3 信頼性観点

コンピュータシステムにおける信頼性評価の指標では，従来RASISと呼ばれる５つの観点がよく知られる．つまり，Reliability（狭義の信頼性），Availability（可用性），Serviceability（保守性），Integrity（保全性）， Security（機密性）の５つの観点である． Reliabilityは，故障をできるだけおこさないことである．Availabilityは，常にシステムが稼動状態にあることである．Serviceabilityは，障害復旧が容易であることであり，Maintainabilityとも呼ばれる．Integrityは，データが矛盾を起こさず一貫性を保つことである． Securityは，機密性が高く，不正なアクセスがなされにくいことである．

これら５つを総合した広義の信頼性は，ディペンダビリティ（Dependability）とも呼ばれる．Dependabilityは，Safety（事故を未然に防ぐ度合い），Performability（処理が瞬間的であれば，障害に遭いにくい），Testability（テストの容易さ）を含めるものもある [40].

近年は，ウィルス感染，情報漏えい，個人情報保護などの観点で，Securityの重要性が高まりつつあるが，Reliability，Availability，Serviceabilityは，サービスを提供するシステム基盤の基本事項として重要である．Securityは，システム

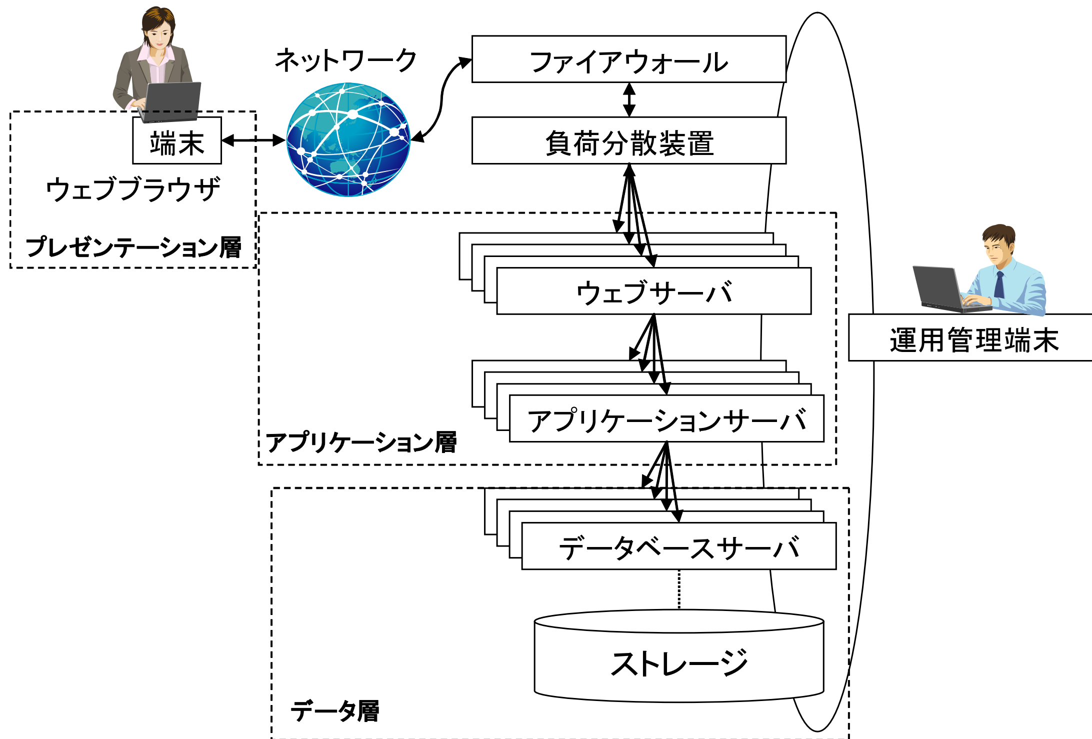


図 2　Web 三層システムモデル例

そのものだけでなく，運用面での課題も大きく，また OS，システムソフトウェアに欠陥があり，そのために Security の問題があったとしても，ベンダ側が責任を負うことは少ない．それに対して，サービス事業者と運用管理者，さらには運用管理者とシステムベンダとの間では，信頼性に関する SLA を取り交わすのが一般的になりつつあり [42]，ベンダも含めて信頼性を保証する責務を負う．

次に，信頼性をどのような指標で考えて，評価し，向上させるかについて議論する．

一般的な製品やサービスの信頼性に関しては，MTTR（Mean Time To Repair），MTBF（Mean Time Between Failures）が用いられる．MTTR は，故障してから，修復され使用可能になるまでの時間の平均値であり，「障害修理時間累計÷故障回数」である．値が小さいほど修理に要する時間が短く，保守性が高いことに

なる．MTBF は，故障してから，次の故障が発生するまでの時間の平均値であり，「稼働時間累計÷故障回数」で求める．値が大きいほど故障せずに稼動している時間が長く，可用性が高いことになる．

データベース・システム，ストレージ・システムにおいては，別の信頼性指標；RTO（Recovery Time Objective），RPO(Recovery Point Objective) が，使われることが多い．RTO は，障害が発生してから復旧に要する時間であり，可用性，MTTR と意味が近い．しかし，RTO の意味する復旧はサービスの再開であり，障害発生前とまったく同じ状態（処理性能，応答時間など）に復旧する必要はない．一方，RPO はサービスの再開時に，システムの状態がどの時点のものであるかを示し，保全性に関連する．通常は，最後に予備を取得した時間であり，もし毎日予備をとっていれば，少なくとも 24 時間前の状態に戻ることができ，RPO は 24 時間となる．理想は，障害が顕在化しないことであり，RTO=0，RPO=0 である．

ここで，オンライン・システムとバッチ・システムというシステムの形態に分けて考えると，RTO，RPO は分かりやすい．

オンライン・システムの典型的な形態は，ユーザからの問い合わせに対して，その処理結果を返すものである．信頼性が低いと，ユーザが端末において問い合わせを入力できない，問い合わせを送信しても正しい処理結果が返ってこない，もしくは何も応答がない，といった不具合が発生する．オンライン・システムの場合は，一般的に，利用者の操作には数秒の時間を要することから，その間にシステムが再起動，もしくは予備機への切り替えをすればよいので，RTO の低減が重要となる．

一方，バッチ・システムは，夜間や週末に，その日，その週の売り上げデータをまとめて処理するような形態である．オンライン・システムと違って，数秒以内に応答を返す必要はないので，復旧に要する時間は長くなってもよい．しかし，一般に大量データを処理することから，処理を初めからやり直すということでは，翌週翌朝の営業開始に間に合わない危険がある．つまり，再実行の範囲を小さくすることが要求され，RPO の低減が重要となる．

本論文では，上記に説明した RTO および RPO が，ICT のサービスの信頼性，

サービス事業の継続性を評価するには妥当と考え，可用性・保全性という観点で議論する．

1.3 解決への基本戦略

前節で述べたような対象領域における信頼性向上のための技術として，運用管理の自動化，さらに多重化構成という基本的な解決方針でもって議論を進める．本節では，運用管理の自動化，多重化構成にフォーカスした理由を説明する．

1.3.1 運用管理の自動化

図 2 のように階層に分離したシステムにおいては，階層にまたがる部分，つまりインタフェースの部分において，障害やミスが発生しやすいことが知られている．特にユーザインタフェース，人間が一番のウィークポイントであると言われる．機器やプログラムは，基本的に仕様の通り動作するのに対し，人間は間違いを犯す可能性が潜在的にある．利用者は，一般に操作に必要な専門知識を過度に求められることはない．システムに間違った値を入力しても，検証し，再入力を促すなどの耐操作ミスの設計がなされる．これに対し，運用管理側は，予期しない状況にも対応する必要があるため，あらゆる運用管理操作が可能となっており，間違った操作でも受け付ける仕様となっている．そのため，運用管理の操作ミスが原因となり，重大障害を引き起こすことがたびたびある．

現在の OS のいくつかは，運用管理者（運用管理アカウント）に，制限された運用管理のコマンド実行やファイルアクセスしかできないような設定をすることが可能である．このように運用管理者の操作を制限し，運用管理者の権限を分散させることは，特にセキュリティ面での信頼性向上の効果がある．しかしながら，運用管理の権限が制限されると，システムの全体を把握することが困難となり，何の障害が発生しているかの判明および対処が遅れる可能性が高い．現在の運用管理製品市場では，障害対処の効率性を重視し，システムを統合的に運用管理し，運用管理は強い権限を持つ方向にある [1]．

[1] 例えば，運用管理者は SLA で合意した操作のみが許可され，それ以外の操作をする為には多

最近のシステムは，大規模，複雑化しており，運用管理者がシステム全体の振る舞いを把握することは，ますます困難となりつつある．運用管理者は，多数の管理項目を監視し，異常があれば，適切な順序で対処操作をする必要がある．もし対処操作の順序を間違えると，サブシステム間で矛盾を引き起こし，障害が顕在化したり，障害範囲が拡大したりする可能性がある．そのため，運用管理者の負担は多大である．運用事業者にとっては，常に正しい運用管理マニュアルを整備し，十分なスキルをもつ運用管理者を確保する必要があり，人件費，教育費などの運用管理コストは，ますます上昇しつつある．

このような状況を踏まえて，米IBM社を始めとして，ソリューションベンダは，自律運用管理（Autonomic System Management），ユーティリティ・コンピューティングなどのコンセプトを打ち出して，システム・運用管理の自動化に取り組んでいる．運用管理の自動化は，システムの大規模・複雑化にともなって増大する運用管理コストの低減の他にもさまざまな効果がある．

1. 運用管理コスト，運用管理者育成コストの低減.
2. 人間の操作が減り，信頼性が向上.
3. システムが自動的に対処するので，復旧が迅速.
4. スキルが未熟な運用管理者でも，ある程度の品質で管理でき，運用管理の品質を一定化.

一方，運用管理の自動化のデメリットは以下の通りである．

1. 自動化の技術開発にコストを要する．
2. 想定外の障害では，警報があがらず，障害に気付くのが遅れる．
3. 自動化の処理負荷のため，サーバリソースが余計に必要.
4. 運用管理操作のブラックボックス化.

くの承認を必要とする．このように技術面よりも運用面で対策がなされていることが多い．ただし，その一方，多くの承認を得るために，障害対処が遅れるという弊害もある．

運用管理の自動化には，メリット，デメリットの両方あるが，将来的にはメリットの効果が大きいと考える．自動化の技術開発コストは，一時的なものであり，また技術開発を深めることで，対応可能な障害を網羅でき，将来的には，想定外の障害をなくすことができると考えられる．また，コンピュータの処理能力は，今後も発展することが期待でき，自動化の処理負荷は無視可能なレベルになる可能性がある．最後の運用管理操作のブラックボックス化の課題は，教育，訓練など制度的に解決できる可能性がある．

以上をまとめると，ICT サービスの信頼性の向上のためには，システムのウィークポイントである人ができるだけ介入しないことであり，運用管理の自動化を推し進めることが重要である．

1.3.2 多重化構成

運用管理を自動化するということは，監視情報の収集，システム状態の判断，対処操作の実行という一連の動作を機械的に行うことであるが，そもそも監視情報が存在し，それを外部から取り出せる手段（障害監視手段）が用意されていて，また障害に対応する操作コマンドを外部から実行させる手段（障害対処手段）が用意されていることが前提となる．障害監視手段に関しては，SNMP（Simple Network Management Protocol）や，WBEM（Web-Based Enterprise Management）などの標準的仕様があるが，障害対処手段は，ICT サービスの形態，システム構成，障害の種類等に依存する．そのため，一律的な障害対処手段を用意することは困難である．

しかしながら，障害復旧の基本的な対処は，欠陥箇所を除去し，正常なモノに置き換えることであり，このような基本的な障害対処手段を用意することができれば，多くの障害に対応可能であると考える．

故障，異常を起こした部品を明らかにし，その部品のみを交換することが簡単にできればよいが，実際には，原因部品の究明が困難であったり，その部品のみの交換は困難であったりする可能性が高い．そのため，部品単位でなく，モジュール単位で管理するのが一般的である．原因を突き止めるよりは，サービスを再開させることが重要であるため，詳細な調査することなく，とにかく交換すれば済

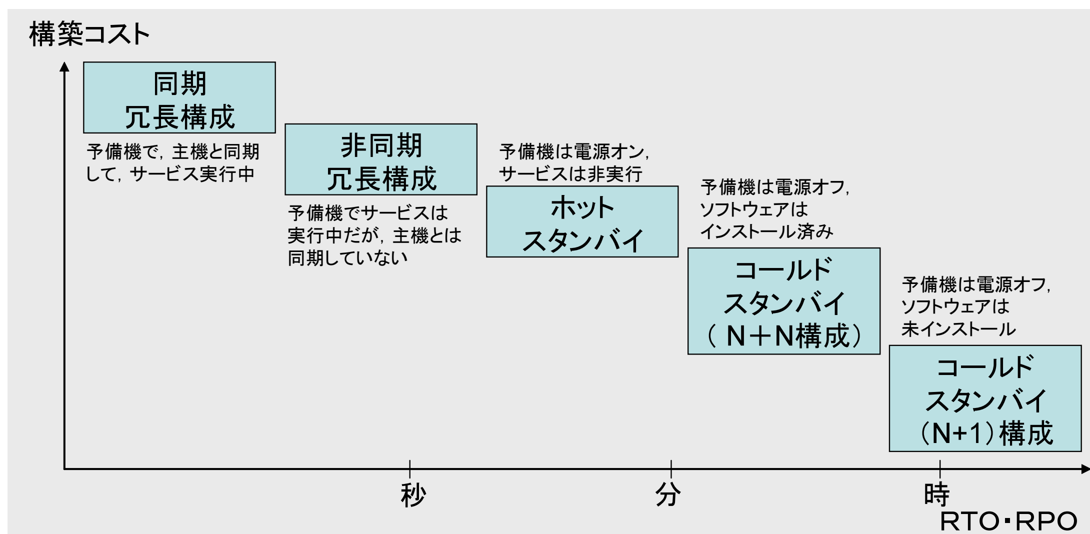


図 3　多重化構成の分類

むように，管理単位は大型化する傾向にある．コンピュータシステムの多くは，PC がコモディティ化したこともあり，PC 単位で管理することが多くなっている．

あらかじめ同等の代替コンピュータを用意しておけば，すぐに対処可能であるが，代替コンピュータを用意しておくためには，コストがかかる．代替品の用意の仕方には，いくつかの方式がある．方式分類を図 3 に示す．

コールドスタンバイ（N+１）構成では，障害発生後に，故障したコンピュータと同等のコンピュータをセットアップする．そのため N 台から構成されるシステムでも，N 台のコンピュータを用意しておく必要がなく，またソフトウェアはインストールされていないため，ライセンス費用が不要であり，コスト的に優位である．ただし，コンピュータのセットアップに手間を要するため，RTO の値は大きい．

コールドスタンバイ（N+N）構成では，システムに N 台のコンピュータがあれば，あらかじめソフトウェアをインストールした N 台のコンピュータを用意しておく．後述する方式に比べ，電源が投入されていないため，エネルギーコスト面で有利であり，またソフトウェアのライセンス的にも，予備用として，追加費

用がかからない場合が多い．コールドスタンバイ（N+１）構成に比べれば，インストールの分だけ RTO は低減される．

ホットスタンバイ構成では，電源が投入されているが，サービスが非実行の状態である．システム上で同一サービスが同時実行されていないので，主機と予備機の間での整合性を管理する必要がない．つまり，冗長構成を管理する特別な機構が必要なく，その分だけコスト面で優位となる．もし，主機が保持しているデータがあれば，そのデータを予備機にリストアする必要があり，その分だけ後述する冗長構成と比べて RTO，RPO は増大する．

非同期冗長構成および同期冗長構成では，冗長構成（予備）側でもサービスを実行し，主機と同じ状態を保持し，主機に障害があれば，いつでも予備機が代替できる．同一サービスを多重に実行し，処理要求，処理応答も多重にあるので，それらを矛盾なく単一サービスと同等に見せるようにする機構が必要である．同期冗長構成では，さらに主機と予備機が同期して動作させるための機構をもつ．

各構成で説明したように，コールドスタンバイ構成では，復旧まで，電源投入，ソフトウェアのインストール，データのリカバリ，サービス再開といった操作手順が必要であるが，同期冗長構成では，そういった操作手順は必要がない．復旧作業に手間がかかると，操作ミスをする可能性が高い．障害発生は，異常事態であり，そのような状態での対応を十分に理解し，訓練をしていないのが一般的であり，管理者はいち早く復旧しようと焦り，操作ミスを重ねるという事例も数多く見られる．

つまり，RTO，RPO が小さいということは，復旧操作ミスのリスクが小さく，復旧の確実性が高いことでもある．サービス事業者の多くは，迅速な復旧だけでなく，復旧の確実性を重視するが，復旧の確実性については，明確な指標を定めることが困難である．そのため，RTO，RPO でもって判断する傾向があり，復旧の確実性という意味でも，RTO，RPO は重要な指標である．しかしながら，RTO，RPO を低減させるほど，構築コストは上昇するので，そのサービスの信頼性要件に適した冗長構成を採用することが重要である．つまり，［サービスが停止，あるいはデータを紛失することによって生じる損失］×［障害の発生確率］と，［冗長構成を構築，運用するコスト］を比較し，見積もることで，好適な冗長構成が

定まる．当然，損失の見積もりでは，楽観的，悲観的な予測があるため，サービス運用者，経営者の戦略的な判断によるところが大きい．

例えば，銀行の勘定系のシステムでは，データを紛失することは一切あってはならないので，多大なコストをかけて信頼性を確保する．一方，商品情報提供サービスで，普段からほとんどアクセスがないような場合には，たとえサーバが故障しても，緊急に修理をする必要はないと判断される場合もある．

以上をまとめると，ICT サービスの信頼性の向上のためには，障害対処手段を用意しておくことが重要であり，コンピュータの多重化構成が基本である．さらに，多様なサービスの信頼性要件に応じて，柔軟に適切な多様化構成をとることが重要である．

1.4 本論文の構成

本章では，企業活動，市民生活が大きく ICT サービスに依存し，今後も安全・安心な豊かな社会を目指し，ICT サービスをますます活用するようになる現状を説明し，ICT サービスの信頼性向上が必要であることを述べた．信頼性の向上とは，いかに障害を顕在化させないか，また障害が顕在化しても，すぐにサービスを再開できるようにするかである．そのための信頼性の指標としては，RTO および RPO があり，できるだけゼロにすることが求められている．RTO，RPO を低減するためには，システムの運用管理を自動化するのが有効であり，運用管理の自動化を推し進めるためには，基本的な障害復旧手段を用意，つまりシステムの多重化構成技術が重要であることを概説した．

以下，2 章では，これまでの障害対策としての多重化構成技術の研究開発を鑑み，本領域における課題を説明する．ここでは，本論文が対象とする障害の範囲を明確化し，システムの多重化に関して，ストレージからアプリケーションまでの各レイヤにおける多重化構成技術の現在までの取り組みと課題を明確化する．

3 章においては，多重化構成技術の１つとして，データベース接続層の拡張によるデータベース複製方式の詳細を説明する．ベンチマークソフトウェアを使った評価にも言及する．

4 章においては，前章で述べたデータベース複製方式の適用場面の１つであるディザスタ・リカバリシステムについて，また同期・非同期複製について詳細に説明する．

本論文で議論するデータベース複製方式の特徴の一つは，信頼性と処理性能とのバランスの柔軟な制御である．5 章では，この特徴を強化する技術として，機械学習を用いたデータベースアクセスのタイミング制御を説明する．

本論文で提案するデータベース複製方式は，アプリケーションとデータベースの間にあって，その間で取り交わされるデータを活用する技術であり，多様な応用が考えられる．6 章では，その応用の１つとして，データベースの置き換え支援について説明する．

7 章では，もう１つの信頼性向上技術として，アプリケーション多重実行方式に言及する．

8 章では，本論文のまとめと今後の展望について説明する．本論文で議論したデータベース複製方式，その中心技術として，データベース接続層の拡張があるが，ここで収集可能なトランザクションの分析を基に，より適応的なトランザクション制御を可能とし，来たるべく安全・安心なユビキタス社会の実現への貢献に関してまとめる．また，今後の研究展望について言及し，結言とする．

2. 多重化構成技術

前章で述べたように，本論文ではICTサービスの信頼性向上の基本的な方針として，運用管理の自動化を目標とし，その障害対処手段の基本手法として，多重化構成技術について議論する．本章では，多重化構成技術についての従来の取り組みついて述べ，解決すべき課題を明確化する．

2.1 障害対処

一般に，障害とは当初の仕様と異なる動作をしている状態である．類似する用語として，故障，欠陥，不具合，バグ，誤り，異常などがあるが，障害が顕在化しているか否か，対象はハードウェアか，ソフトウェアか等で，異なる意味で使われたり，また同じ意味であったりする．本節では，本論文が対象とする障害の意味，範囲を規定し，本論文の障害に対する考え方を説明する．

2.1.1 障害とは

障害については，これまでに様々な研究がなされているが，当麻らは障害発生の事象を図4に示すようにモデル化して，障害を説明している[61]．つまり，Fault（欠陥）→Error（誤り）→Failure（故障）といった順で，障害は発生する．Faultは潜在状態であり，Errorで検知できる状態であり，Failureはサービスに影響が顕在化した状態と考えられる．一般的には，Fault，Error，Failureを区別することなく，すべての状態を「障害」と呼ぶ．ソフトウェアのプログラムミスについては，Fault，Error，Failureの区別なく「バグ」と呼ぶ．また,「不具合」,「異常」は,「障害」と断定できない場合に用いられることが多い．

簡単に，Fault，Error，Failureの各状態を説明する．

Fault システムの構成要素に内在する障害要因であり，ハードウェアの故障やソフトウェアの潜在バグなどを指す．障害が発生した場合にはその根本的な原因となるが，障害が起こるまでは発見は困難である．Fault の存在は，その Fault に起因する Error の発生によって検出される．

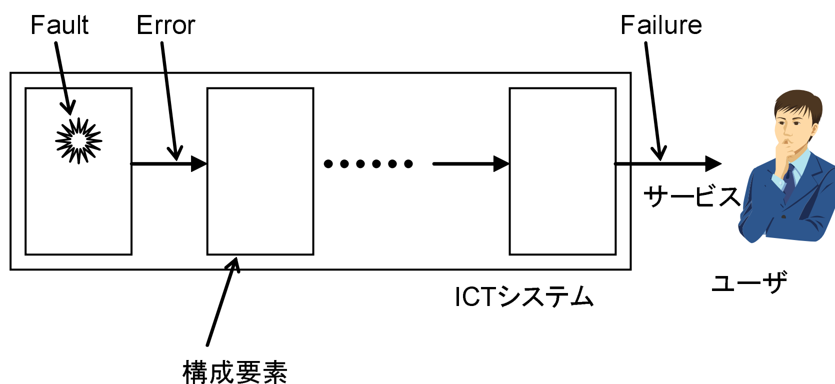


図 4　Fault，Error，Failure（[61] より）

Error Fault によって引き起こされる検出可能な状態を指す．ソフトウェアの出力結果が想定範囲内にない場合などである．Error 状態が波及することによって，最終的に Failure が引き起こされる．記憶装置や通信においては，この段階で検出，訂正する誤り検出訂正の各種符号技術が広く使われている．

Failure 本来の機能が損なわれ，何らかの支障が顕在化している状態を指す．Error によって引き起こされた状態によって，対象物は望ましくない挙動をとる．

一般的にサービスを利用しているユーザが異常に気付き，システム管理者に通報したタイミングが，障害の発生時点となる．障害管理システムを使っている部門であれば，その障害を障害管理システムに登録し，システム管理部門が認識した時点である．障害管理システムでは，一度，障害が認識されると，障害が解消されるまで，トレース（管理）される．また，SLA を結んでいる場合には，SLA に違反する状態を障害と考える．

2.1.2 耐障害とは

前節で述べたような障害モデルを想定すると，Error から Failure に波及する前に，障害対策を行い，サービスへの影響を顕在化させない障害対処が重要である．

具体的には，障害を検知し，例外処理を実行し，障害の波及を抑制する．必要であれば，障害を除去し，処理を再開する．

まず，障害（Error）の検知であるが，基本的に仕様やいつもと異なる動作や処理結果がある状態を検知することである．つまり，正しい動作，処理結果，もしくは通常時の動作，処理結果という比較対象がなければ，障害を検知することはできない．最も簡単なのは，比較対象が運用管理者の頭の中にあり，運用管理者が異常を認識することであるが，当然ながら人間による検知は不確実である，

比較対象の可能性としては，過去の動作，処理結果の履歴か，もしくは複数の処理を同時実行して，その動作，処理結果を比較するかのいずれかである．履歴との比較の例としては，データマイニングを使った外れ値検出の研究などがある[50]．過去の記録が膨大，もしくはシステムが大規模になると，過去の記録と照合という処理のため，多大なコンピュータリソースを必要とし，また高速に障害を検知することは困難となる．複数の処理を同時実行し比較する方式は，システムを多重化構成することに関連し，詳細は次節以降で述べる．

次に検知後の処理であるが，障害の種類やサービスの種類によって，対処の方法は異なる．例えば，長時間稼動システムは，計画的な保守による停止は許容されるが，保守と保守の間で障害が顕在化しないように，たとえ障害が発生した場合でもフェイルオーバーするような対処が要求されるであろう．別のシステムでは，少数の故障は放置してもサービスが継続できるような縮退運転のような対処が求められる可能性がある．

障害を識別し，その障害に応じた対処を行うことが理想的である．しかしながら，障害を識別することは困難である．表 1 に障害の分類の一例を示す [38]．このように障害がどのように発生しているか，多様な観点から調査する必要があるので，障害を識別することは手間と時間がかかる．障害を識別し，障害対処をしていたのでは，サービス再開までに時間がかかりすぎてしまう問題もある．

故に障害を特定することなく，対処を進めることが望ましいと考える．つまり，障害箇所を含むモジュールごと交換する，もしくは切り離して縮退運転することでサービス継続する手法が重要である．

表 1　障害の分類

分類	障害
出力	なし（タイムアウト），あり（間違った値）
活性状態	潜在状態，顕在状態
永続性	一時的，永続
発現	定性的，非定性的
原因	ランダム，一般
意図	過失，故意
頻度	単発，多発
時間（複数障害）	同時，非同時
原因（複数障害）	独立，共通

2.2　耐障害のための基本的な多重化手法

2.2.1　多重化アーキテクチャ

耐障害のために数多くの多重化構成の研究がなされてきたが，そのシステムアーキテクチャとしては，デュアル構成かクラスタシステム構成に集約しつつある．以下に代表的なアーキテクチャであるデュアル構成，タンデム構成，クラスタ構成の 3 つを説明する．

デュアル構成　CPU，メモリ，ビデオカード，ネットワークカードなどのハードウェアコンポーネントを二重化し，それぞれ高速のバスで接続する（図 5）．二つの CPU で同じ処理を行い，二つの内部記憶・外部記憶に同じデータの読み書きを行う．外部インタフェースカードの入出力は，いずれかアクティブになっている側を使う．各コンポーネント間の入出力信号を照合しながら，処理を進めるが，一方が障害をおこした場合には，それを切り離して，処理を続行できるようになっている．RAID[25] は，外部記憶（ハードディスク）に限定した多重化コンポーネントである．

各コンポーネントは，専用のデバイスドライバを必要とするが，OS レイヤよ

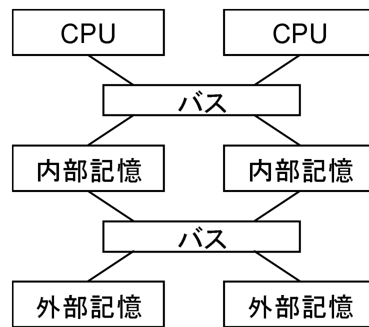


図 5　デュアル構成

り上では，普通のデバイスとして認識され，二重化を意識せずに，通常の単一構成型のコンピュータと同じように使うことができる．どのようなアプリケーションでも，適用可能であるが，普通の単一構成のコンピュータと同様に，OS のバージョンアップなどの作業の際には，コンピュータの停止が必要である．また，二重化されたコンポーネントは冗長な構成であるので，コストパフォーマンスは劣る．

近年，バス速度も向上していることから，デュアルシステムの開発には高度な同期技術が必要となっている．現在は，耐障害性とコストパフォーマンスで優位な CPU のマルチプロセッサ構成，ハイパースレッディング機能，マルチコアアーキテクチャなどの方が一般的である．

タンデム構成，デュプレックス構成　デュアル構成が，同等のコンポーネントであるのに対して，異なる（通常は能力の劣る）コンポーネントを用いた構成をタンデム構成・デュプレックス構成と呼ぶ．通常，タンデムは直列構成，デュプレックスは並列構成を指す．デュアル構成に比べ，ハードウェア的には，コストパフォーマンスに優れるが，このシステム専用のプログラム開発を必要とするなどソフトウェア開発の負担が大きく，また障害発生時の切り替えも難しいので，最近はほとんど採用されることはない．

クラスタ構成　クラスタは，複数のコンピュータ（ノード）を接続して，処理（ジョブ）を分散して実行する（図 6）．クラスタは，性能向上と耐障害の両面で有効で

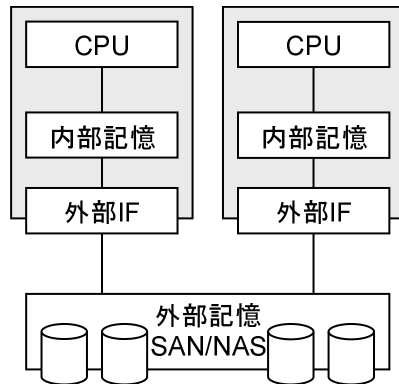


図 6 クラスタ構成

ある．性能向上として負荷分散機能を備え，耐障害としてフェイルオーバー機能を備える．フェイルオーバーにおいて，待機系側が何も処理せずに，いつでもオンライン処理に切り替えられる状態で待機していることをホットスタンバイという．各ノードは，１つ１つが独立しているので，１つを停止させて，OS やアプリケーションのバージョンアップなどを容易に行うことができる．クラスタ構成は，デュアル構成に比べ柔軟性が高いといえる．一方では，各ノードが IP アドレスなど異なるシステム構成情報となるため，アプリケーションによっては，切り替え時に特別な処理を必要とする場合がある．クラスタシステムでは，ノード間の通信，システム構成情報の更新などのため，デュアル構成のように一瞬で切り替えるということは困難である．

これらの多重化構成の特徴を表 2 にまとめた．このような多重化構成では，どのように構成変更の設定をし，どの時点で構成変更を実行するかということが問題となる．

上述したような多重構成において，障害検知後に構成の切り替えを一瞬で行い，RTO=0，RPO=0 を実現することが理想である．しかし，無停止で切り替えるハードウェアおよびソフトウェアの機構は高価であり，きめ細かくチェックポイントを実施することは性能に影響がある．サービスの要件によって，耐障害の対象とする障害の種類，処理パフォーマンス，復旧機能は異なり，耐障害の技術，

表 2　デュアル構成とクラスタ構成

<table>
<tr><th></th><th>デュアル構成</th><th>クラスタ構成</th></tr>
<tr><td>対処レイヤ</td><td>ハードウェア（＋OS）</td><td>ミドルウェア＋AP</td></tr>
<tr><td>対処方法</td><td>障害部位の切り離し</td><td>代替部位への移行</td></tr>
<tr><td>HW 障害への対応</td><td>故障した HW を切り離し無停止で対応可能</td><td rowspan="2">予備系へのフェイルオーバーで対応，アプリケーションの移動など数分を要する場合あり</td></tr>
<tr><td>SW 障害への対応</td><td>基本的に対応できない．再起動で対処できることもある</td></tr>
<tr><td>操作ミスへの対応</td><td colspan="2">対応できない</td></tr>
<tr><td>対象アプリケーション</td><td>任意</td><td>フェイルオーバー機能の追加が必要</td></tr>
<tr><td>事前設定</td><td>不要</td><td>フェイルオーバーのスクリプト</td></tr>
<tr><td>保守</td><td>構成全体を停止させる必要</td><td>構成の一部を切り離して保守可能</td></tr>
<tr><td>コスト</td><td>専用ハードウェアで高価</td><td>汎用 PC を利用可能</td></tr>
</table>

構成は使い分けがなされている．

2.2.2 システム/ソフトウェアの多重化

ICT サービスは多様であり，信頼性の要件もさまざまである．高い信頼性を実現するには費用がかかるので，信頼性の要件に適応したサービスシステムの構築が求められる．当然，低コストでありながら，高い信頼性を得ることができるような信頼性技術が重要であるが，運用管理者の人件費，教育費などを含めたトータルコストを算出することは，容易でない．さらに，サービス要件は，サービス運用中にも変わる可能性がある．顧客数の増減，何かしらのイベントでバースト的にアクセス数が急増するなどの環境の変化は，インターネットサービスではしばしばある．そのため，システムを一度構築したら，それが最終形ということはなく，環境の変化に応じて，柔軟にシステム構成を変更できる柔軟性，スケーラビリティが重視されるようになってきている．

故に，以下においては柔軟性の高いクラスタ構成を想定し，さらにソフトウェアレベルでの信頼性向上の技術について論じる．ハードウェアに比べ，ソフトウェアは，その一部のモジュールを停止，起動させることも容易であり，またパッチをあてるなど，プログラムの変更も容易であり，柔軟性は高い．

ソフトウェアレベルでの従来の信頼性向上の技術としては，リカバリ・ブロック [26]，N バージョン [3] の手法が著名である．いずれもソフトウェアレベルで複数のバージョンを用意し，システムの信頼性を高める手法である．以下に簡単に説明する．

リカバリ・ブロック　本手法は，あらかじめ複数のバージョンを用意しておき，もし Error が発生したら，別のバージョンで処理を実行する手法である．基本的な構成を図 7 に示す．通常は，主バージョンで処理を進めるが，もし，障害があれば，まずチェックポイント管理を使って，当該処理を主バージョン実行前に戻る．そして，第 1 代替バージョンを実行する．出力切り替えで，第 1 代替バージョンの処理結果を出力する．もし，第 1 代替バージョンでも障害があれば，第 2 バージョンを実行し，それでも障害があれば，第 3 バージョンを実行し，を繰り返す．

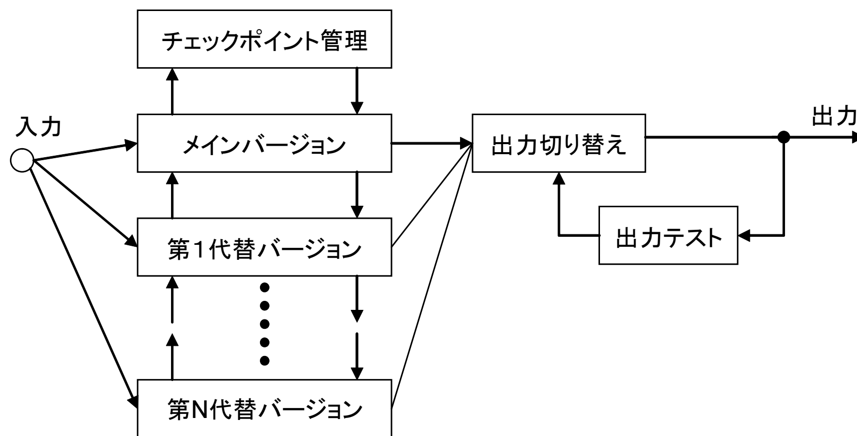


図 7 リカバリ・ブロック

N 個のバージョンの実行は，並行実行でもよいし，順番に実行するようにしてもよい．

Java の例外処理機構が，リカバリ・ブロックの実現事例としてよく知られる．

N バージョン 本手法は，異なるソフトウェアベンダに，同一の機能要件のソフトウェアを開発させ，それらのソフトウェアを同時に実行し，出力結果を比較する手法である．適切な出力結果（通常は多数決）を選択することで，システムの信頼性を向上させる技法である（図 8）．

ここで問題となるのは，出力計算の方式である．各バージョンからの出力を投票し，多数決などにより出力を計算する．他の出力計算の方式としては，最も投票の多かった範囲の代表値，平均値，各バージョンに重みを付けた平均値などがある．Croll らのようにニューラルネットなどの AI 技術を使って出力するような手法もある [11]．また，複数のバージョンの中から，障害を発生中のバージョンを検知する手法としては，ビザンチン問題 [21] がよく知られる．それぞれのバージョンが異なった投票をした場合，もしくはしなかった場合に，本当に故障しているのはどのバージョンか，もしくは入力分配器なのかを特定する．

上記で述べた，リカバリ・ブロック，N-バージョンといった手法は，ソフトウェアレベルでの耐障害性を備えるため，異なるバージョンを用意し，また高度な入

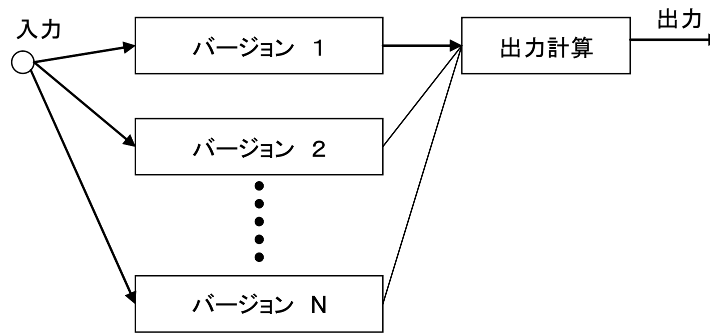


図 8 N バージョン

出力の切り替えが必要であり，実装コストが多大である．実際には，これらの手法の本格的な適用は，航空，原子力といった多くの人命に関わるようなミッションクリティカルなシステムに限定されている．リカバリ・ブロック，N バージョンのいずれの手法も，ソフトウェアに限らず，その考え方は，さまざまな場面で適用できる汎用的なものである．次に，このような多重化管理処理をどのレイヤで行うかの観点で，レイヤごとに多重化技術を議論する．以下の節では，ストレージ，データベース，アプリケーションなどの各レイヤにおける具体的な多重化事例について議論し，現状の耐障害，高信頼化技術を概観する．

2.3 ストレージの多重化

ICT サービスを実現するシステムには，さまざまな構成要素からなるが，特に重要なのは，データである．処理プロセス，あるいは処理を実行する主体であるプロセッサは，処理途中で停止，故障したとしても，処理をやり直せばよい．それに対して，データを消失した場合，元に戻すことができない．予備やデータの生成元から，再度データを再構築するのは，時間と手間がかかり，サービス再開までの RTO の値は悪くなる．

そのため，データを保護する取組みはたいへん多い．本節では，データの保存先であるストレージを多重化することで，データ保護し，信頼性向上を実現する

手法を説明する．

2.3.1 ストレージアクセス

前節で述べたように，モジュールそのものを多重化することだけでなく，その入出力管理も重要である．ストレージの管理は，通常 OS のファイルシステムが中心となっており，ファイルシステムとストレージとの間での入出力を理解しておく必要がある．以下に，例として Linux を取り上げ，ファイルシステムを概観する．

Linux のファイルシステムは，主に 2 層から構成されている．上位層は，VFS（仮想ファイルシステム）と呼ばれ，アプリケーションから，ファイルシステムの差異を意識せずにさまざまな形式のファイルシステムにアクセスすることを可能とするインタフェースである．一方，下位層は，各種のファイルシステムで，表 3 のような主なファイルシステムがカーネルに組み込み可能である．

また，ファイルシステム層の上位および下位のインタフェースで，キャッシュ機構が実装される．上位インタフェース（VFS のカーネルインタフェース）では，i-node および dentry のキャッシュが行われる．下位インタフェース（各種ファイルシステムのデバイスインタフェース）では，ページ単位のキャッシュが行われる．

Linux のファイルシステムの構成を図 9 に示す．Linux のファイルシステムの詳細な解説については，[2][31][45] などに掲載されている．

図 9 にあるように，ストレージを多重化する場合，ファイルシステムより上位（システムコール），ファイルシステム内部，ファイルシステムより下位（ドライバ），ストレージ外部（ネットワーク），ストレージ内部でその多重化の入出力を管理する可能性がある．

システムコールレベルにおける多重化管理では，下位にキャッシュ機構があるために，本当にストレージに書き込んだかどうかの確認が困難であり，他アプリでシステムコールがフックされていた場合に動作が不安定になる可能性がある．システムコールそのものが多様であり，同じ書き込み機能でもアプリケーションによって，異なるシステムコールを使っている場合がある．このように，システムコールレベルでは，信頼性，汎用性に問題があり，このレイヤにおける多重化

表 3 Linux のファイルシステム

ファイルシステム名	特徴
Ext2	基本的なファイルシステム
Ext3	Ext2 をベースにジャーナル機能を追加
Reiserfs	Ext3 に比べ，高速，ジャーナル機能強化
JFS	（IBM 社製）ジャーナルファイルシステム
VxFS	（VERITAS 社製）予備機能強化
FAT/VFAT	MS-DOS
Iso9660	CDROM
UDF	DVD
NFS	ネットワーク
shm	RAM ディスク

管理はほとんどなされていない．

ファイルシステム内部における多重化管理，つまり予備や複製を管理する例としては，Coda[33] が良く知られる．しかしながら，一般的なファイルシステムと違い，特殊用途向けであり，広く利用されているとはいえない．コンピュータシステムとしては，性能を重視する場合も多く，ファイルシステムレベルでは，最小限の信頼性（ジャーナル，ログ）で十分な場合が多い．ファイルシステムを変更することは，ファイルシステム間で互換性がなくデータの移動が必要なこともあり，リスクが高い．何よりも，将来でもアクセス可能であるという信頼性の観点もあり，最も普及しているファイルシステムを採用することが多い．

多重化管理に関連して Spiralog[18] と ReiserFS[22] などで使われているファイルシステムログ技術がある．これらのシステムでは，ファイルアクセスログが保護され，システム障害のためにファイルが壊れた場合に，そのファイルの復旧に利用される．後述するアプリケーションやデータベースの状態を復旧するために，ファイルだけでなく，処理状態も含め，統合的に管理，制御するが，これらのファイルシステム技術はその要素技術として重要である．

ファイルシステム下位（ドライバ）における多重化管理では，デバイスドライ

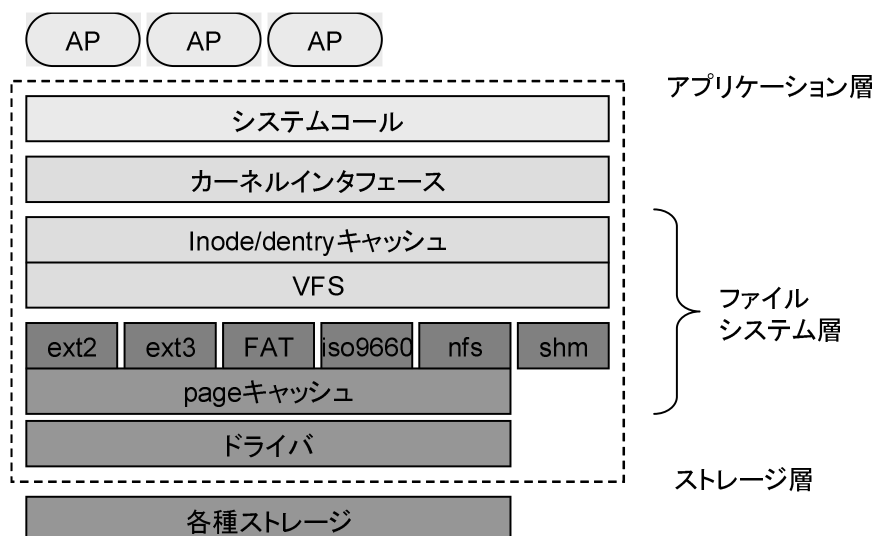


図 9 Linux のファイルシステム構成

バを置き換える手法であり，2.3.2 において詳細を説明する．

ストレージ外部（ネットワーク）における多重化管理では，ネットワーク経路上の中継装置を置き，そこで何らかの制御・管理処理を実行する手法であり，2.3.3 において詳細を説明する．

ストレージ内部では，ストレージ製品依存の多重化管理であり，ハードウェア的な信頼性向上のアプローチである．そのため，本論文の議論の対象外とする．

2.3.2 ドライバレイヤでの多重化

このレイヤでの多重化管理に関しては，RAID（Redundant Arrays of Inexpensive Disks）[25] がよく知られている．RAID は，複数のディスクを用いることで，一部のディスクが故障してもデータ損失を防ぎ，また処理性能を向上させる技術である．RAID を実現するには，ハードウェア方式とソフトウェア方式とがあり，いずれも複数のディスクをあたかも 1 つのディスクのように見せるような制御機構を有する．特に，耐障害に関連するのは，RAID1 に分類されるミラーリング機能である．しかし，エラーデータも含めて，複製，上書きされるなど，耐障害の

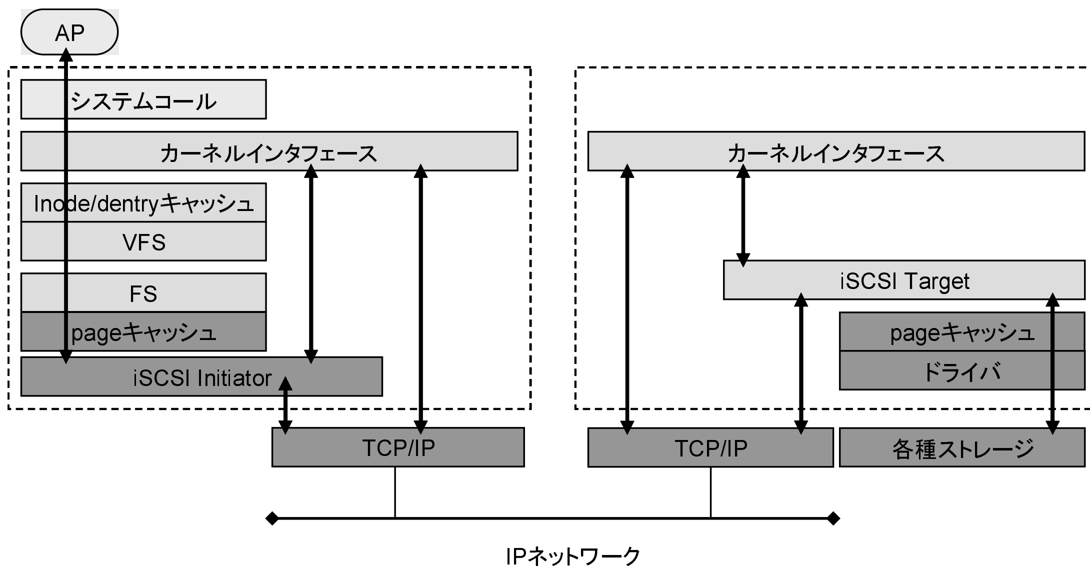


図 10 iSCSI システムの構成

機能は十分とはいえない．

RAID が１つのコンピュータシステムに適用される技術に対し，ネットワーク上のシステムに拡張された RADD と呼ばれる機構もある [37]．RADD は，コンピュータ間でデータを保護する基本的な耐障害システムといえるが，LAN のみを対象としており，広域ネットワーク上での動作性能は検証されていない．

また，最近 iSCSI[9] を利用したデータ複製の研究が活発である．iSCSI とは，SCSI コマンドを IP パケット化することで，ネットワーク上のディスクをローカルな SCSI ディスクと透過的に扱うことを可能とする規格である．その一般的な構成を図 10 に示す．SCSI の名前が付いているが，IP ネットワーク上を流れるストレージ領域提供側と利用側のネゴシエーション情報が，SCSI コマンド体系に準拠しているからで，SCSI ディスク，SCSI デバイスとは直接関係はなく，また各ファイルシステムの実装と独立している．ストレージ領域を提供する側を Target，ストレージ領域を利用する側を Initiator と呼ぶ．

Initiator は，Target に対して，マウントの要求を行う場合，Initiator-Target 間でネットワーク接続を行い，認証を行い，接続ソケットのファイルディスクリプ

タを登録する．データ通信を行う場合には，セッションごとにスレッドを生成し，登録されている接続ソケットを用いて，それぞれ通信ソケットを生成する．データ通信は，ページ単位で行われるので，例えば１つのファイルを変更（生成）すると，inode，dentry，データ領域の少なくとも３つのページが更新される．Targetは，送られてきたページデータを page キャッシュ機構に対して渡すだけである．図 10 に示すように，iSCSI の両側のインタフェースにおいて，page キャッシュ機構があるので，アプリケーションおよびストレージにおけるネットワークの影響は極力抑えられている．

インターネットを利用することで，コスト削減を進めたいなどの市場ニーズから，iSCSI は広く使われ始めている．しかし，小サイズのデータを主サイトと予備サイトとで高頻度にやり取りするため，すべてのサーバが集積され，高性能ネットワークに接続されているようなデータセンタでは有効かもしれないが，遠距離間では実用的な性能を期待することはできない．

2.3.3 ネットワークレイヤでの多重化

ネットワークレイヤでの多重化管理では，NFS サーバ／クライアント間の通信データを監視し，予備サーバに転送し，複製を行う手法などがある [55]．この手法では，通信データの監視サーバおよび予備サーバのネットワークと既存システムとを切り離すことができ，システム性能に影響を与えることなく複製処理を実現できる．しかしながら，予備サーバの処理状態を既存システム側にフィードバックすることが困難であり，整合性を保持することは難しい．さらに，パケットロスへの対応も困難である．

また，P2P ネットワークを使ったストレージの多重化も提案されている．Ocean-Store[32] は Tapestry[4] を用いて，また PAST[29] は Pastry[30] を使って，複製（多重化）を管理する技術が知られる．これらは，ファイル単位での複製管理，およびバージョン管理を行うものである．そのため，継続して更新されるようなデータ処理を含むシステムへの適用は想定されていない．

このような P2P 型のシステムの特徴の１つは，信頼性の低いサイトを多数集めながらも，信頼性を確保していることである．構成要素のサイトの多くが停止し

た場合でも，システム全体として，正常データにアクセス可能とする．また，アクセス元から，最も近いサイトのデータにアクセスを誘導するなど，負荷分散を含めた高速なアクセスを実現可能とする．つまり，アップストリームよりは，ダウンストリーム型のサービスへの適用が想定されている．

システムが多数の複製を管理するので，広域かつ大規模となると，名前管理，検索が困難である．また，各サイトの信頼性が低い可能性から，セキュリティを要するようなサービスには，不向きである．信頼性が見積もれないことは，ビジネスへの適用を致命的なものとしている．信頼性の低いサイトを集めるより，ある程度信頼性を確保できる予備サイトを一つか二つ用意する方が，信頼性，性能，費用を見積もることができ，サービス運用者にとっては都合がよい，

2.4 データベース・サーバの多重化

前節では，ストレージの多重化，複製について説明したが，データベース・サーバの単位で多重化し，サービス全体の信頼性を向上する手法もよく知られる．データベース・サーバの製品には，複数のストレージを制御し，複製を管理する機能を有するものもあるが，ここでは，データベース・サーバのプロセスが複数実行される図 11 のようなクラスタ構成を対象とする．

データベース・サーバは ACID 性 [17]，つまり Atomic（原子性），Consistent（一貫性）， Isolated（分離性），Durable（存続性）を実現するように構成されたシステムである． データベースへのデータベース処理要求（トランザクション）は，ストレージレイヤよりは抽象化されたコマンドであり，一般に通信データサイズは小さい．そのため，通信オーバーヘッドの影響を受けにくく，ネットワーク環境に対するロバスト性は高いといえる．このような特長があるため，データベースレイヤで多重化を実現するデータ複製技術に関する取組みは数多く，データベース・サーバをシステムの構成要素に含む場合には，データベース・サーバで多重化を管理する場合も多い．

またトランザクションは，バイナリ形式とテキスト形式があるが，最近は処理性能が向上し，処理性能の点では差異が小さいため，可読性の点で有利なテキスト形式が利用される．例えば，デバッグや障害時のログをトレースするなどが容

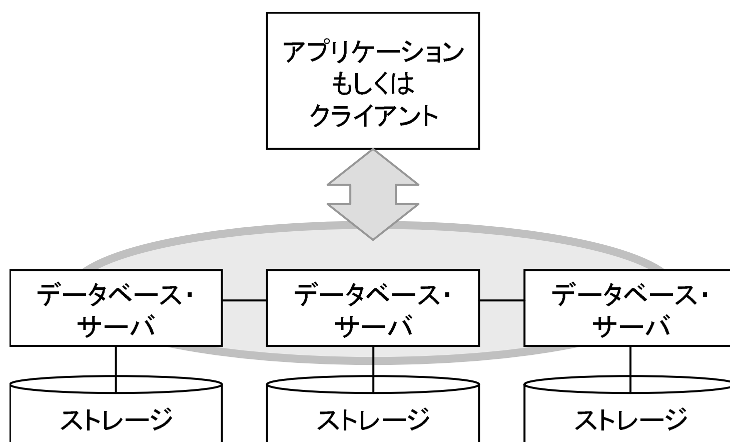


図 11　データベース・サーバの多重化

易となる．一般的には，トランザクションの記述には，SQL が使われる．可読性があることから，データ処理，トランザクション処理に応じてのシステムの振る舞いを制御し，柔軟性の高いシステムを構築することを可能とする．

既存のデータベースレイヤでの多重化技術は，データベース・サーバ内部の多重化と外部の多重化に分けられる [14]．

いくつかのデータベース製品においては，データベース内部型の多重化機能を提供しており，ベンダ独自のプロトコルで，トランザクションを複製し，予備データベースに通信することで，多重化機能を実現している．しかし，多重化機能はハイエンドのエンタープライズバージョンに限定されていた．

最近，多重化機能はいくつかのオープンソースのデータベース製品にも実現されてきているが，まだ機能的に十分であるとは言えない．例えば，MySQL Cluster[28] と Postgres-R[19] は多重化機能を追加するために，プログラムソースコードを修正する必要がある．PGCluster[59] は修正なしで多重化機能を追加することが可能だが，オリジナルの PostgreSQL データベースエンジンを変更する必要がある．

その他に負荷の問題もある．データベース・サーバは，分散構成では整合性を確保するオーバーヘッドが大きいため，単一サーバで構成されることが多い．そのため，データベース・サーバには高性能のサーバマシンを用いるが，サーバマ

シンの追加による処理性能増強ができないので，想定以上の処理でデータベース・サーバがボトルネックとなることも多い．従って，データベース・サーバの負荷が問題となるシステムの場合，データベース・サーバ内部に多重化機能を追加することは，不適切である．

データベース・サーバ外部での多重化方法としては，Pgpool[58]，C-JDBC[7] 等の技術がよく知られている．Pgpool は，PostgreSQL へのアクセスを複製するプロキシサーバをシステムに追加し，また C-JDBC においても，C-JDBC コントローラと呼ばれるモジュールをシステムに直列追加する．このような追加モジュールは，アプリケーション・サーバとデータベース・サーバのネットワーク上にあっても，またいずれかのサーバ内にあってもよい．ただし，データベース・サーバ外部での多重化を実現する際，このような追加モジュールが，単一障害点とならないように，かつ性能ボトルネックとならないようなシステム構成とすることが必要である．

また，データベースレイヤにおける多重化では，障害後の冗長体制への復旧に際し，データベースを再構築するためのログを保存する大量のストレージを要し，またログを再投入処理するので復旧に長時間を要するというような課題がある．そのため，データベースを一時停止し，スナップショットを取得，それを復元して冗長体制を復旧する方式が用いられることも多い．

以上の議論をまとめると，表 4 のようになる．サービスの信頼性要件に応じて，適した多重化方式を選択するべきであり，さらに，単一障害点を新たに設けない，アプリケーションのコード修正が不要などの機能も考慮する必要がある．

2.5 アプリケーションの多重化

一般的なインターネットサービスでは，障害があると，そのセッションを破棄，ログインをしなおし，入力を始めからやり直すことが多い．サーバは，負荷分散構成になっており，あるサーバが故障したとしても，そのアクセス状態が失われるだけで，ストレージ，もしくはデータベース・サーバに保存されたデータは保護される．

表 4 多重化管理位置の検討

多重化管理位置	ストレージ	DB（内部）	DB（外部）
導入の容易さ	○	×	×
原子性保証	△	○	○
サーバ負荷	○	×	△
通信帯域	×	○	○
冗長体制復旧	○	△	△
柔軟性	×	△	○

それに対し，たとえ障害があっても，処理状態を保持した状態で処理を継続したいという要求がある．例えば，複数の予約を連続して，予約状況を確認しながら，作業を進めたいような場合である．このように，状態を継続するように，システムの切り替えを行うことができる仕組みを「ステートフルなフェイルオーバー」と呼ぶ．一方，状態を保持しないものは,「ステートレスなフェイルオーバー」である．いくつかのアプリケーション・サーバにおいては，ステートフルなフェイルオーバー機能を備える．

BEA 社の WebLogic では，アプリケーションの処理実体の JavaBean をメモリ（内部記憶）レベルで多重化する．つまり，ステートフルなフェイルオーバーを実現するには，アプリケーションを特定の Java クラスで実装する必要がある．また，メモリレベルでの多重化であるので，遠隔での多重化は性能的には困難で，ローカルに接続された環境が前提となる．

データベースにおけるトランザクション制御と同様に，アプリケーションレベルでの ACID 性を保証するような仕組み，つまり，メッセージの送達保証，重複防止，順序保証を Web サービスで実現するミドルウェアが提供され始めている．対象となるプロトコルは SOAP で，WS-Reliability[35]，WS-RM（ReliableMessaging）[36] といった仕様が公開されている．このようなプロトコル，およびミドルウェアを採用することで， Web サービスにおいて，ステートフルなフェイルオーバーが可能な環境が整いつつある．

Web サービスに依存しないステートフルなフェイルオーバーを実現する技術と

しては，ソケットライブラリを置き換え [1]，もしくは TCP のパケットをラッピングするなどして [12]，ネットワークアドレスを変換し，仮想的に同一のネットワーク環境を複数用意し，そこで同一のアプリケーションを動作させる研究もある．これらは，ほとんどのアプリケーションを変更することなく，そのまま適用できる点で有利である．一方，デメリットとしては，アプリケーションレベルのログ，障害監視は，逆変換，もしくはパケットのラッピングを解除する必要があり，手間がかかることがある．また，データベースやストレージを共有するようなアプリケーションへの適用には注意が必要であり，この詳細については，7 章で言及する．

2.6 多重化構成技術のまとめ

本章では，多重化構成技術についての従来の取り組みついて述べた．障害は，Error が顕在化しないよう，また顕在化してもできるだけ迅速にサービスを復旧しなければならない．そのために，システムはクラスタ構成を採用し，一部の構成要素が障害となっても，他の構成要素で継続してサービスを提供する，もしくは障害部分を切り離して，別の構成要素に高速にフェイルオーバーしてサービスを再開することが重要である．具体的な多重化構成技術への取り組みとして，ストレージの多重化，データベースの多重化，アプリケーションの多重化を説明した．ストレージの多重化では，ドライバレベルとネットワークレベルの多重化について詳細に述べた．

近年のサービスの多様化，動的な構成変更にも対応できるような柔軟性の高い高信頼化技術が求められており，データベースの多重化技術の可能性が高い．しかしながら，性能，単一障害点などの課題もあり，また処理性能と信頼性のバランスの制御手法はまだ確立されていない．

さらに，データの保護だけでなく，処理プロセスの保護，つまりステートフルなフェイルオーバーに関する研究も盛んである．現状は，従来アプリケーションとの互換性，運用管理など他ミドルウェアとの連携といった点で，技術的にはまだ発展途上である．

本論文の 3 章以降では，多重化構成技術の１つであるデータベース接続層にお

ける多重化管理を中心に，ICT サービスの信頼性向上技術を提案し，その詳細を説明する．3 章では，基本構成として，図 12 に示すようなアプリケーション１：データベース N の構成を説明し，7 章では，その逆で，アプリケーション N：データベース１の構成を説明する．4，5，6 章では，3 章の応用構成，応用技術を説明する．

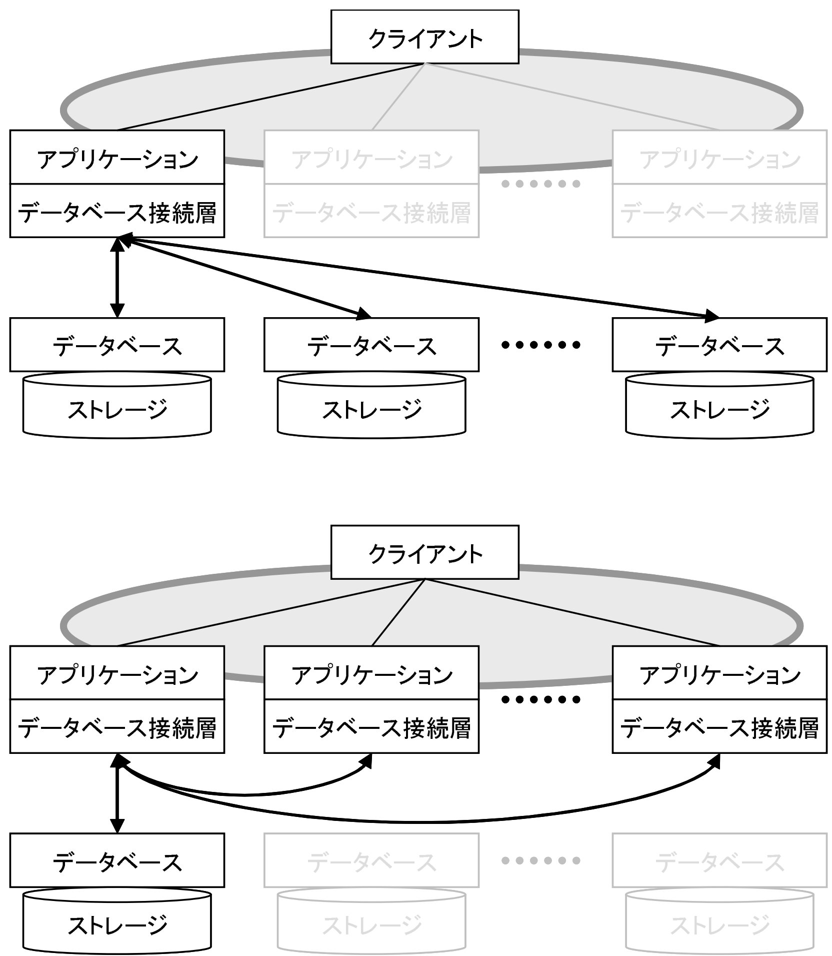


図 12　（上）アプリケーション 1：データベース N 構成（3 章）と (下）アプリケーション N：データベース 1 構成（7 章）

3. データベース接続層の拡張によるデータベース複製方式

前章で見てきたように，さまざまな多重化構成技術があり，それぞれにメリット，デメリットを有する．従来は，あるICTシステムを構築しようとすれば，そのシステムの提供するサービスに必要とされる信頼性，投入可能な資源，障害が発生した場合の損失などを考慮して，適切な多重化構成を採用し，高信頼化を図ればよかった．しかしながら，サービスの移り変わりは早く，利用者や業務の量が急に増減し，システムに求められる要件もダイナミックに変化している．多くのシステムベンダは，システムの仮想化に取り組み，こういったダイナミックな変化に迅速に対応可能なサービス基盤の提供を始めている．多重化に関しても，サービスおよびシステムの変化に追従する必要があるが，これまでの多重化管理技術では，柔軟性については議論されておらず，また十分な柔軟性があるとはいえない．実際，多くの多重化技術の障害対処では，フェイルオーバーは自動的に行われるものの，冗長体制への復旧は，サービス負荷の低い時間帯にサービス停止をスケジューリングし，そのサービス停止の間に冗長構成に戻す手順となっている．これでは，システム構成を安易に変更することは困難である．

このことより，解決すべき課題は，柔軟な多重化管理が実現できていないことであると考える．具体的には，現在主流のストレージのミラーリングでは，サービスとデータの対応付けが難しく，サービスに応じた多重化制御は困難である．また，ディザスタリカバリのように，データを遠隔地に保存する場合に，信頼性を確保しようとすれば，通信遅延による処理性能の低下は避けられない．ミッションクリティカルなサービスにおいては，構築コストによらずに信頼性を重視するため，サービスに応じた多重化や通信遅延の影響を最小化するように，高い費用をかけてプロプライエタリにシステムを構築する．しかし，一般的なICTシステムにおいては，高い費用をかけて信頼性向上を行うことはできない．汎用かつ柔軟な多重化管理の基盤が提供されれば，ICTシステム全体の信頼性向上に有用である．

以下，考案した汎用かつ柔軟な多重化管理技術について説明し，サービスレベ

ルの多重化制御，通信遅延のシステム性能への影響を最小化する制御をいかにして実現しているかを説明する．

3.1 データベース複製

3.1.1 多重化管理位置の定性的検討

ICT システムは，5 ページ図 2 に示したように，一般に，サービスを要求するクライアント，サービスを提供する Web サーバおよびアプリケーション（AP）サーバ，アプリケーションの処理に必要なデータの管理を行うデータベース（DB）サーバ，実際にデータを格納するストレージから構成される．ここで，どの位置で多重化管理を行うべきかを検討する．各レイヤの機能をより詳細に説明する図を図 13 に示す．

アプリケーションでデータベースに対する処理要求が発行されるが，この段階の処理要求は，トランザクション処理，つまり確定的により処理される．これより上位においては，たとえば，HTTP の処理要求は，どの Web サーバで，どの順番で処理されるかは不確定である．多重化管理する場合は，処理順序が変わると，複製処理結果が異なる可能性がある．そのため，処理要求は確定的である必要がある．

また，サービス状況に応じた柔軟な多重化管理を行うためには，どのようなデータが予備系に転送されているかどうかを把握する必要がある．例えば，トランザクションの途中での多重化構成の変更では，主系と予備系との処理結果は異なる可能性があるため，トランザクションが完了したかどうかを確認する必要がある．下位（ストレージ側）においては，データがバイナリであったり，デバイスの制御コマンドであったりするなど，転送データの意味を把握することは困難である．

処理要求は，上位（クライアント側）においては抽象的であり，下位においては具体的である．そのため，上位の方が一般的にデータ量，データ通信頻度は小さくなる．遠隔やモバイルのように，通信環境が良好でない場合，データ量，通信頻度が少ない方が優位である．

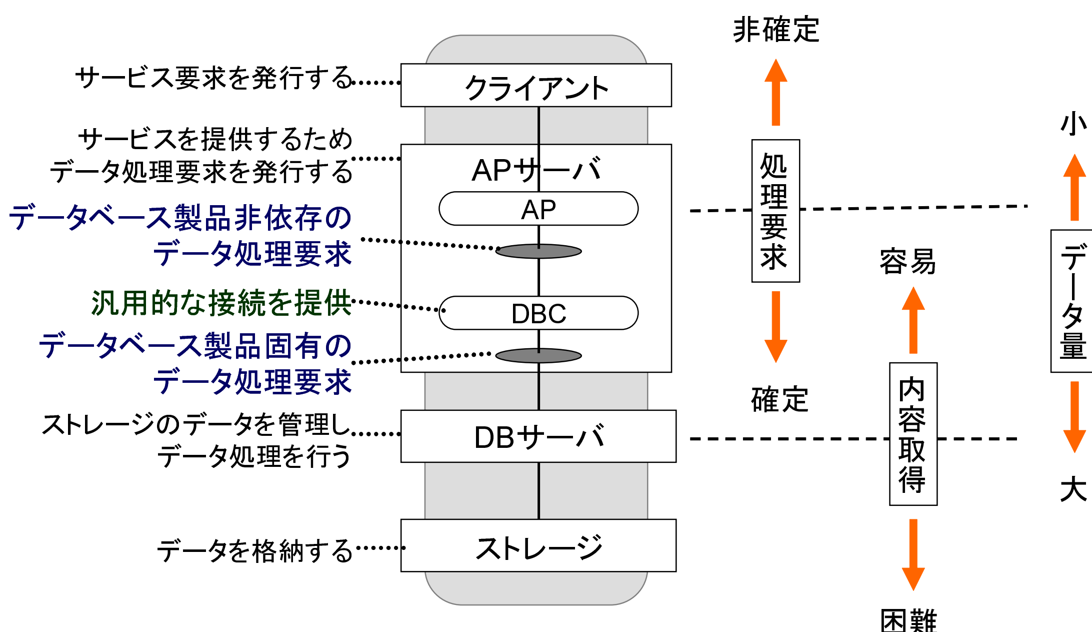


図 13　多重化管理位置の検討

以上の要件を考慮すると，データベース接続層，もしくはデータベース・サーバにおける多重化管理が，管理位置として候補となる．

データベース接続層における多重化　近年，データベース・サーバと連携した処理を行うアプリケーションにおいて，アプリケーションからデータベース・サーバに直接接続するのではなく，汎用的なデータベース・サーバ接続機能を提供するデータベース接続層を介して接続することが一般的である．データベース接続層としては，JDBC，ODBC，ADO.NET などが知られる．データベース接続層は，アプリケーションから発行されるデータベース製品に依存しない処理要求を受け，データベース製品依存の処理要求に変換して，データベースに送る．データベース側は製品ベンダから提供されるドライバを通常用い，ドライバを取り換えることで，各種のデータベースに対応することが可能である．

そのため，アプリケーション側で多重化管理しようとすれば，JAVA や Windows のような開発環境に依存することになる．一方，データベース側で多重化管理し

ようとすれば，データベースに依存した実装となる．

課題としては，アプリケーション・サーバはスケールアウト，つまり負荷分散構成をとる可能性があり，多重化管理も分散処理となり，分散した多重化管理の整合性を管理する必要がある．さらに，既存方式では，多重化管理をカスタマイズするような機能として，フェイルオーバーするまでのタイムアウト時間の設定程度しかサポートされておらず，柔軟性は十分とはいえない．また，障害発生時（誤検知を含む）にフェイルオーバーした後，冗長構成に復帰するにはサービス停止が必要であり，この点でも柔軟性に欠ける．

データベース・サーバにおける多重化　データベース・サーバにおいて多重化管理しようとすれば，データベース接続層からの処理要求は製品依存のプロトコルであり，製品依存の実装となる．

データベース・サーバにおける多重化の問題点は，データベース・サーバの負荷が高まることである．最近では，データベース・サーバそのものがクラスタ構成を採用し，負荷分散することが可能となってきたが，分散構成で処理性能を向上させることはいまだ困難な作業であり，スケールアウトすることは稀である．スケールアウトをしない，もしくはスケールアウトしても性能向上が困難なデータベース・サーバがボトルネックとなる場合も多く，データベース・サーバの処理が増加することは望ましくないと考える．

他の問題点としては，データベース接続層における多重化の場合と同様に，カスタマイズ機能が不十分であること，サービス無停止で構成変更ができないことである．

3.1.2 多重化管理位置の定量的検討

本論文における信頼性を測る指標はRTO，RPOであることは前述した．多くのフェイルオーバー方式はRPO=0でサービスを復旧するものであることから，本節ではRTO，特に可用性を対象に議論を進める．

検討方法は，(i) ストレージ，(ii) ストレージとデータベース間，(iii) データベース，(iv) アプリケーションとデータベース間の各位置における複製に対し，(1) 主

表 5　多重化管理位置と障害復旧手順

<table>
<tr><th>多重化管理位置</th><th>故障箇所</th><th>処理中の障害発生</th><th>待機中の障害発生</th></tr>
<tr><td rowspan="5">(i)ストレージ</td><td>(1)主ストレージ</td><td>タイムアウトで故障部を切断
ストレージの切り替え
FSCK</td><td>タイムアウトで故障部を切断
ストレージの切り替え</td></tr>
<tr><td>(2)予備ストレージ</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(3)主データベース</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(4)予備データベー</td><td colspan="2">N/A</td></tr>
<tr><td>(5)アプリケーションサーバ</td><td>タイムアウトで故障部を切断
ロールバック，デッドロック解除</td><td>なし</td></tr>
<tr><td rowspan="5">(ii)ストレージ～DB間</td><td>(1)主ストレージ</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(2)予備ストレージ</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(3)主データベース</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(4)予備データベー</td><td colspan="2">N/A</td></tr>
<tr><td>(5)アプリケーションサーバ</td><td>タイムアウトで故障部を切断
ロールバック，デッドロック解除</td><td>なし</td></tr>
<tr><td rowspan="5">(iii)データベース</td><td>(1)主ストレージ</td><td>タイムアウトで故障部を切断
DBの切り替え
ロールフォワード</td><td>タイムアウトで故障部を切断
DBの切り替え</td></tr>
<tr><td>(2)予備ストレージ</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(3)主データベース</td><td>タイムアウトで故障部を切断
DBの切り替え
ロールフォワード</td><td>タイムアウトで故障部を切断
DBの切り替え</td></tr>
<tr><td>(4)予備データベー</td><td colspan="2">タイムアウトで故障部を切断</td></tr>
<tr><td>(5)アプリケーションサーバ</td><td>タイムアウトで故障部を切断
ロールバック，デッドロック解除</td><td>なし</td></tr>
<tr><td rowspan="5">(iv)AP～DB間</td><td>(1)主ストレージ</td><td colspan="2">タイムアウトで故障系を切断</td></tr>
<tr><td>(2)予備ストレージ</td><td colspan="2">タイムアウトで故障系を切断</td></tr>
<tr><td>(3)主データベース</td><td colspan="2">タイムアウトで故障系を切断</td></tr>
<tr><td>(4)予備データベー</td><td colspan="2">タイムアウトで故障系を切断</td></tr>
<tr><td>(5)アプリケーションサーバ</td><td>タイムアウトで故障部を切断
ロールバック，デッドロック解除</td><td>なし</td></tr>
</table>

ストレージの停止，(2) 予備ストレージの停止，(3) 主 DB サーバの停止，(4) 予備 DB サーバの停止，(5) アプリケーション・サーバの故障が，それぞれ (a) 処理中に発生した場合，(b) 待機中（処理中以外）に発生した場合での復旧手順と復旧に要する時間を検討する．まず，表 5 に復旧手順を示す．

次に，復旧に要する時間を算出するためには，次のようなパラメータを導入する．単位時間を T とし，T の時間内にストレージの停止障害が発生する確率 Ps，同じくデータベースの停止障害が発生する確率 Pd，同じくアプリケーションの停止障害が発生する確率 Pa，ストレージの処理中率 Us，データベースの処理中

率 Us，アプリケーションの処理中率 Ua，ストレージの停止障害と判断されるタイムアウト時間 Ts，データベースの停止障害と判断されるタイムアウト時間 Td，アプリケーションの停止障害と判断されるタイムアウト時間 Ta，ストレージの fsck に要する時間 Tfsck，データベースのブートに要する時間 Tboot，ロールバックに要する時間 Trb，ロールフォワードに要する時間 Trf，とする．なおタイムアウト時間には，IP アドレスや通信先の設定変更の処理を含む．表 5 に対応する可用性を表 6 に示す．主系と予備系とが非同期の際，復旧処理に依らずにサービスを継続することが可能な場合があり，表中では (Async: 100 %) で示す．

それぞれの障害が発生する確率，処理中率は，各場合で異なるため簡単には比較することができない．ストレージの処理中率は，ストレージが高性能であれば低下し，またサービスによっても大きく異なる．また，多重化管理を含むシステムでは，その部分に障害が含まれる確率が高くなるだけでなく，処理中率も高くなる．当然ハードウェアおよびソフトウェアのモジュール構成が異なれば，これらのパラメータは大きく異なる．

ここで，比較を簡単にするため，障害が発生する確率，タイムアウト時間は同じと仮定すると，各多重化管理位置の可用性の分母の右項は次のような差となる．この項ができるだけ小さい方が可用性は高くなる．

ストレージ	$Ps \times Us \times Tfsck + Pd \times Tboot$
ストレージ〜DB 間	$Pd \times Tboot$
データベース	$(Ps \times Us + Pd \times Ud) \times Trf + Pd \times Td$
AP〜DB 間	$Pd \times Td$

ここから分かることは，データベースを瞬間的にブートできるのであれば，ストレージ〜DB 間 での多重化管理（複製）が有利であり，またデータベースの応答タイムアウト時間を短く設定できるのであれば，アプリケーション〜データベース間での多重化管理が有利であることである．データベース処理のタイムアウト時間に関しては，サービスに依存するが，通常のインターネットサービスでは，ネットワーク通信を含めて 3 秒程度以内で応答し，また後述する本提案環境でのデータベース性能が毎秒 10 トランザクション程度であったことを考えると，2 秒程度で十分と考えられる．一方，ブート時間は，サービスの規模にもよるが，ホットスタンバイ型のデータベースとしても，テーブルの初期化等で，数 10 秒

表 6 多重化管理位置と可用性

多重化管理位置	故障箇所	処理中の障害発生	待機中の障害発生
ストレージ	主ストレージ	$\frac{T}{T+Ps\times Us\times(Ts+Tfsck)}$	$\frac{T}{T+Ps\times(1-Us)\times Ts}$
	予備ストレージ	$\frac{T}{T+Ps\times Ts}$ （Async: 100 %）	
	主データベース	$\frac{T}{T+Pd\times(Td+Tboot)}$	
	予備データベース	100 %	
	AP サーバ	$\frac{T}{T+Pa\times Ua\times(Ta+Trb)}$	100 %
ストレージ～DB 間	主ストレージ	$\frac{T}{T+Ps\times Ts}$	
	予備ストレージ	$\frac{T}{T+Ps\times Ts}$ （Async: 100 %）	
	主データベース	$\frac{T}{T+Pd\times(Td+Tboot)}$	
	予備データベース	100 %	
	AP サーバ	$\frac{T}{T+Pa\times Ua\times(Ta+Trb)}$	100 %
DB	主ストレージ	$\frac{T}{T+Ps\times Us\times(Ts+Trf)}$	$\frac{T}{T+Ps\times(1-Us)\times Ts}$
	予備ストレージ	$\frac{T}{T+Ps\times Ts}$ （Async: 100 %）	
	主データベース	$\frac{T}{T+Pd\times Ud\times(Td+Trf)}$	$\frac{T}{T+Pd\times(1-Ud)\times Td}$
	予備データベース	$\frac{T}{T+Pd\times Td}$ （Async: 100 %）	
	AP サーバ	$\frac{T}{T+Pa\times Ua\times(Ta+Trb)}$	100 %
AP～DB 間	主ストレージ	$\frac{T}{T+Ps\times Td}$ （Async: 100 %）	
	予備ストレージ	$\frac{T}{T+Ps\times Td}$ （Async: 100 %）	
	主データベース	$\frac{T}{T+Pd\times Td}$ （Async: 100 %）	
	予備データベース	$\frac{T}{T+Pd\times Td}$ （Async: 100 %）	
	AP サーバ	$\frac{T}{T+Pa\times Ua\times(Ta+Trb)}$	100 %

程度要すると考えられる．小規模なシステムであれば，より高速なブートが可能であろう．

よって，一般的にはアプリケーション〜データベース間での多重化管理，例えばデータベース接続層での多重化管理が RTO の観点からは優位である．これは多重化管理が上位レイヤになれば，それだけ多重化度が上がるため，信頼性も向上するのは当然である．一方，小規模なシステムであれば，つまり代替データベースをすぐに起動できれば，データベース〜ストレージ間の多重化管理が優位である．

3.1.3 データベース接続層における多重化の特徴

論理レベルでの多重化の問題：検証手法　システムを構築する場合，多重化の機能だけでなく，他の付随する機能も踏まえて検討する必要がある．例えば，システムやコンポーネントを多重化した際のリアルタイム検証がある．論理レベルの多重化では，複製データが正確に複製できているかどうかの検証は困難であり，これまで検討してきたデータベース多重化の方式にも共通する問題である．

単純なデータ複製であれば，既存の各種の誤り検出，誤り訂正技術を適用できる可能性がある．しかしながら，このような物理的データを対象とした誤り検出，誤り訂正技術では，論理的なデータの複製の正しさを検証することはできない．そこで，いくつかの解決手段の可能性を検討した．

まず，データ操作処理の後に，自動的にデータ照会処理を行うことが考えられる（図 14）．データ操作（update など）に対するデータベース応答には，操作処理のデータは含まれないが，自動的にデータ照会処理を行い，操作処理したデータを得る．各データベースからの応答が異なれば，データベース接続層において複製に失敗したこととなる．

しかしながら，この手法では，データベース側のキャッシュされたデータを照会している可能性が高く，本当に保存されたデータではない可能性が高い．

また，別の方式としては，通常データベース・サーバでは，処理ログを保存するが，その処理ログを利用することが考えられる（図 15）．データベース接続層で管理するアクセスログは，データベース非依存の処理要求であるが，データベース・サーバが管理する処理ログは，データベース製品依存の処理要求である．こ

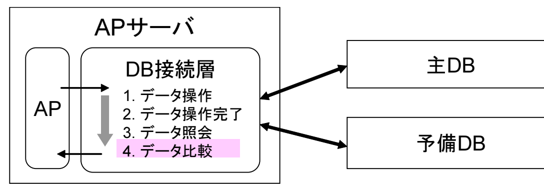


図 14　データ照会による複製検証

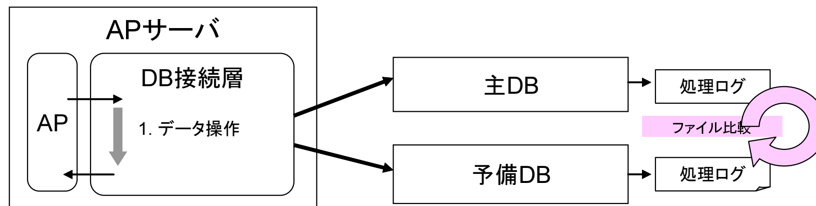


図 15　データベース処理ログによる複製検証

の処理ログを比較することで，複数のデータベースの複製状態を確認できると考えられる．

しかしながら，比較対象は，ログであり，データそのものではないので，信頼性の点で問題がある．

上述したように，論理レベルでの複製に対する検証手法の方式開発は発展途上である．従来研究においても，この点に関して言及しているものはほとんどなく，今後の課題であると考える．リアルタイムで複製検証するようなサービスは，ハイエンドのミッションクリティカルサービスであり，通常は，オフラインでデータ照合すれば十分であることから，本論文においては，このようなリアルタイム検証を考慮せずに議論を進める．

多重化管理の要件　上記の検討を踏まえて，データベース複製による信頼性向上には，データベース接続層における多重化管理が適していると考える．図 16 にデータベース接続層による多重化管理の基本構成を示す．アプリケーションが発行したデータ処理要求を主データベース用に変換して発行するだけではなく，予備データベース用にも変換して，予備データベースにも発行する．このようして，主データベースと予備データベースの両方で同じデータ処理を行わせることで，

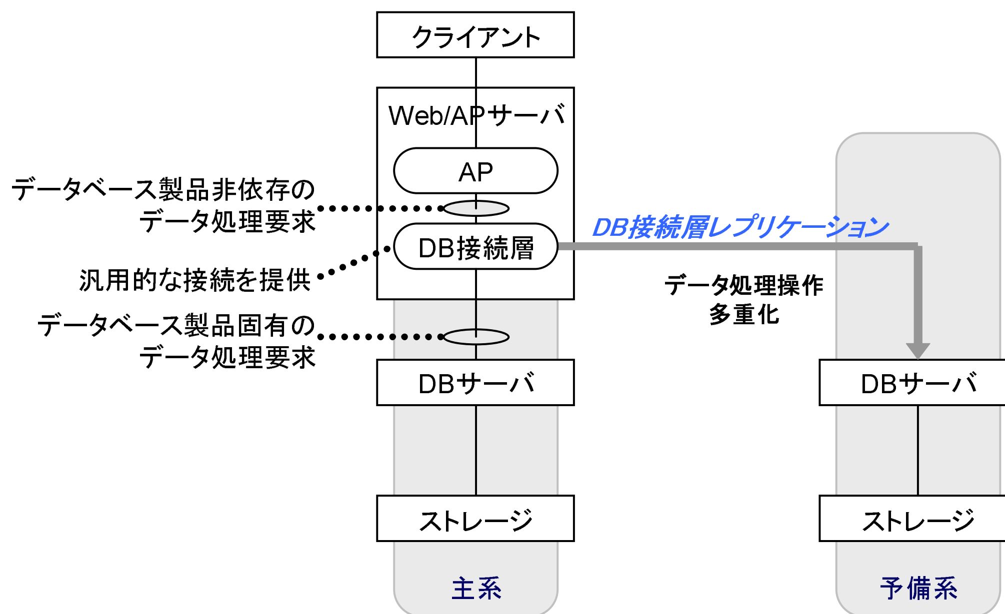


図 16　提案するデータベース接続層におけるデータベース複製

データ複製を実現する．

データベース・サーバで行った処理の結果である応答は，主／予備両方のデータベース・サーバからの応答をデータベース接続層が一旦受けて，主データベースの応答のみをアプリケーションに返す．アプリケーションにとっては，通常どおりデータベース接続層に対してデータ処理要求を発行し，そしてその応答が正常に（＝１つだけ）返ってくるようにみえるので，アプリケーション，およびクライアントからは，この多重化処理を意識することはない．

そして，データベース接続層における多重化管理をベースとして，前述した下記の課題に取り組む．

課題１　冗長構成への復帰にはサービス停止が必要

課題２　負荷分散構成時の整合性管理による性能低下

課題３　複製制御のカスタマイズ機能が不足

課題１を解決する技術について以下に説明する．課題２については，3.5 節において，課題３については，次章において議論する．

3.2 無停止フェイルバック

サービス無停止で，多重化管理を行うために，主系および予備系の両方が動作している構成を前提とする．この場合のフェイルオーバーとは，障害発生時に，冗長構成から，障害発生モジュールを切り離し，縮退運転でサービスを継続することである．また，フェイルバックとは，縮退運転状態から，主系および予備系の両方が動作している冗長構成に復旧することである．

このように冗長モジュールを接続したり，切り離したりすることができれば，容易に多重化構成の変更が可能となり，柔軟な多重化管理が実現できる．

フェイルオーバーに関しては，サービスを停止させない耐障害機構として実現されてきている．それに対して，フェイルバックは，任意のタイミングで実行すればよいもので，通常はアクセス負荷の少ない都合のよい時間に計画的にサービスを停止させて構成変更を行っている．そのため，汎用的な無停止フェイルバックに対しては，取り組みが少なく，技術的に確立されているとはいえない．

本来，次の障害がいつ発生するかは分からず，できるだけ早い時期にフェイルバックすることが望ましい．無停止フェイルバックが実現できれば，できる限り早いタイミングで冗長体制に復旧することが容易となる．

また，フェイルバックが容易であると，少しでも障害の兆候があれば，早めにフェイルオーバーすることが可能である．安全が確認できたのであれば，すぐにフェイルバックすることができるためである．例えば，早めにフェイルオーバーするとは，接続のタイムアウト時間を短縮することである．通常は，間違ってフェイルオーバーしないように，長めにタイムアウト時間を設定するが，障害検知時間を遅らせることとなる．タイムアウト時間を短縮することで，信頼性は向上する．

フェイルオーバーの手法は既知の事項であるので，ここではフェイルオーバーについては，簡単に説明し，その後フェイルバックの詳細について説明する．

3.2.1 フェイルオーバー

本提案システムのフェイルオーバーとは，拡張したデータベース接続層内で，接続中のデータベース・サーバとの応答を監視し，もし一方のデータベース・サーバの応答がない，もしくは別のデータベース・サーバと異なる応答があった場合に，そのデータベース・サーバとの接続を切断し，それ以降は，残りのデータベース・サーバとのみ通信をして，処理を継続する．接続を切断するだけであるので，その処理は短時間で終了する．

ただし，データベース・サーバが二つの場合，異なる応答があった場合に，いずれのデータベース・サーバの応答が正しいかどうかは判断できない．このため，この処理関しては，ロールバックし，処理前の状態に戻り，アプリケーションに対しては，エラーを返す．エラーのハンドリングに関しては，アプリケーション側に任される．

通常のシステムでは，別途管理サーバがあるか，データベース・サーバが相互に監視する形態であり，フェイルオーバーのトリガーは外部にあるが，本提案はデータベース接続層に閉じてフェイルオーバー処理がなされるので，処理が簡単で，高速かつ確実である．[2]

3.2.2 短時間停止における無停止フェイルバック

データベース接続層において，データベース処理要求ログおよびデータベース処理応答ログを保存する．このログを使って，無停止フェイルバックを実現する．

何かしらの障害が発生した場合には，フェイルオーバーし，縮退運転にはいる（図 17 の (1)）．縮退運転中，データベース処理要求・応答ログは，データベース接続層のバッファに蓄積される（図 17 の (2)）．もし，短時間停止であれば，つまりバッファが溢れる前に冗長データベースが復旧すれば，その処理要求ログを

[2] 他の要因で引き起こされるフェイルオーバーについては，別途用意される運用管理ソフトウェアに依存することとなる．例えば，他の要因とは，ネットワーク経路上のハブの故障などである．アプリケーション・サーバもデータベース・サーバも正常に動作を継続し，故障したハブを迂回するようにネットワーク設定を変更するのであるが，そのような設定変更は，多重化管理機構だけでは対処できない．他のミドルウェア管理も含めた統合的な運用管理が必要である．

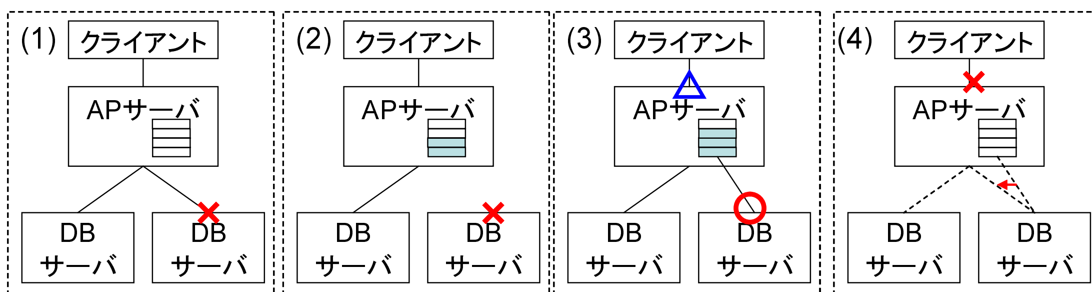


図 17 短時間停止における無停止フェイルバック

使って，一時停止したデータベース・サーバに処理要求を投入する（図 17 の (3)）．すべてのバッファにある処理要求を投入し終われば，サービスを一時停止し，冗長体制に復旧する（図 17 の (4)）．クライアントからの処理要求頻度が高く，バッファにある処理要求数が減少しないようであれば，クライアントからの処理要求を制限してもよい．

3.2.3 長時間停止における無停止フェイルバック

短時間停止の場合と同様に，何かしらの障害が発生した場合には，フェイルオーバーし，縮退運転にはいる（図 18 の (1)）．縮退運転中，データベース処理要求・応答ログは，データベース接続層のバッファに蓄積される（図 18 の (2)）．もし，長時間停止であれば，つまりバッファが溢れていれば，バッファの処理要求ログを使うことはできない．まずは現在稼働中のデータベースの予備を取得する（図 18 の (3)）．次に縮退運転状態に戻し（図 18 の (4)），停止したデータベースに先に予備したデータをリストアする（図 18 の (5)）．この間は，データベース処理要求・応答ログは，データベース接続層のバッファに蓄積される．リストア終了後，蓄積された処理要求を一時停止したデータベース・サーバに処理要求を投入する（図 18 の (6)）．すべてのバッファにある処理要求を投入し終われば，サービスを一時停止し，冗長体制に復旧する（図 18 の (7)）．クライアントからの処理要求頻度が高く，バッファにある処理要求数が減少しないようであれば，クライアントからの処理要求を制限してもよい．

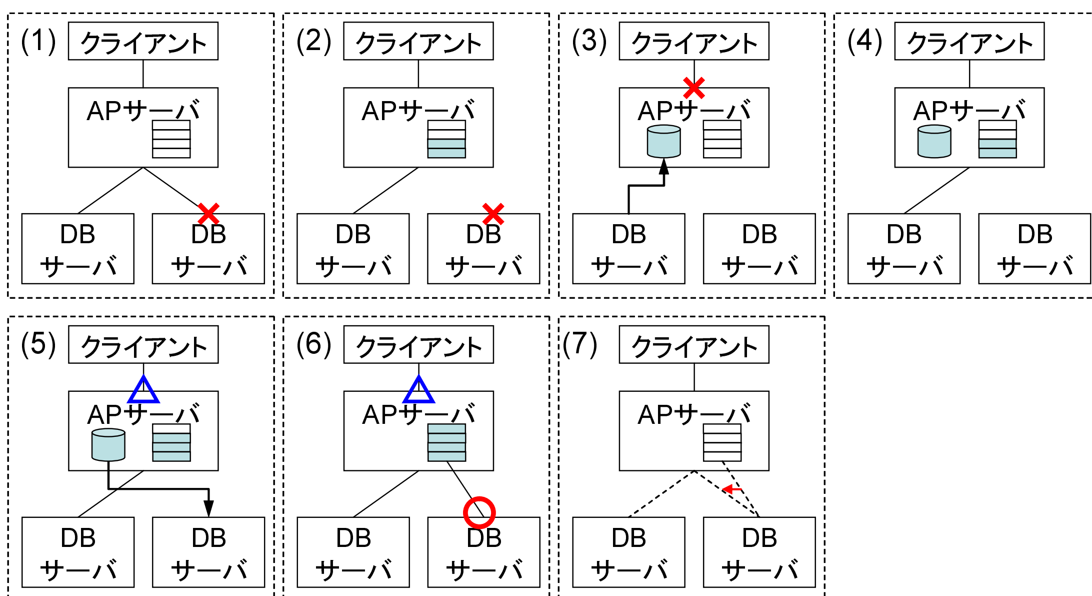


図 18　長時間停止における無停止フェイルバック

3.3　基本試作

本章の以降の節では，データベース接続層でのデータ複製の詳細な方式を説明し，上記のメリット，デメリットを評価することで，提案方式のICTシステムへの適用可能性を議論する．J2EEプラットフォームにおけるデータベース接続層のミドルウェアであるJDBC[3] [27]に，データベース接続層拡張によるデータ複製方式を実装した．本節では，この実装について述べる．

多くのJ2EEアプリケーションは直接，もしくはBeanなど上位のライブラリから間接的に，JDBCを介してデータベース・サーバ接続を行っている．JDBCは，JDBCドライバとドライバマネージャの二つのコンポーネントから構成される．JDBCを用いた場合における，アプリケーションからデータベース・サーバへのデータ処理要求の流れを，図19に示す．

JDBCドライバは，各データベース・サーバ固有の接続を提供することでデータベース・サーバ毎の差異を吸収するコンポーネントであり，主にデータベー

[3] Java DataBase Connectivity が由来とされる

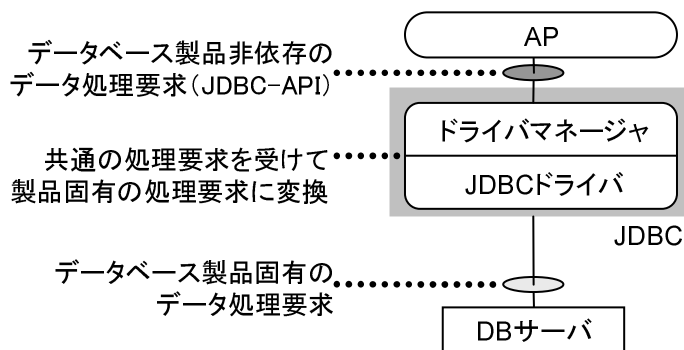


図 19 JDBC の構成

ス・サーバベンダより提供される．ドライバマネージャは，アプリケーションと JDBC ドライバとを接続するための，データベース・サーバに依存しない標準インタフェース（JDBC-API）を提供するコンポーネントであり，JDK コア API として java.sql パッケージに実装されている．アプリケーションは JDBC-API を用いることで，データベース製品に依存することなくデータベース・サーバと接続できる．

このような JDBC の構成を踏まえ，共通のデータ処理要求，すなわちドライバマネージャ部におけるデータ処理要求を多重化することで，データベースの複製を実現する．具体的には JDBC ドライバマネージャにラッパをかぶせ，アプリケーションから JDBC-API を介して発行されるデータ処理要求を複製する．このラッパを JDBC ゲートウェイと呼ぶ．JDBC ゲートウェイを用いた際の，アプリケーションからデータベース・サーバへのデータ処理要求の流れを，図 20 に示す．

このようにデータ処理要求を多重化することで，主データベース，予備データベース両方のデータベース・サーバに同じ処理を行わせ，その結果，二つのデータベースの状態を同一とし，もし，主データベースに障害発生しても，予備データベースを使って，サービスを継続させることが可能となる．

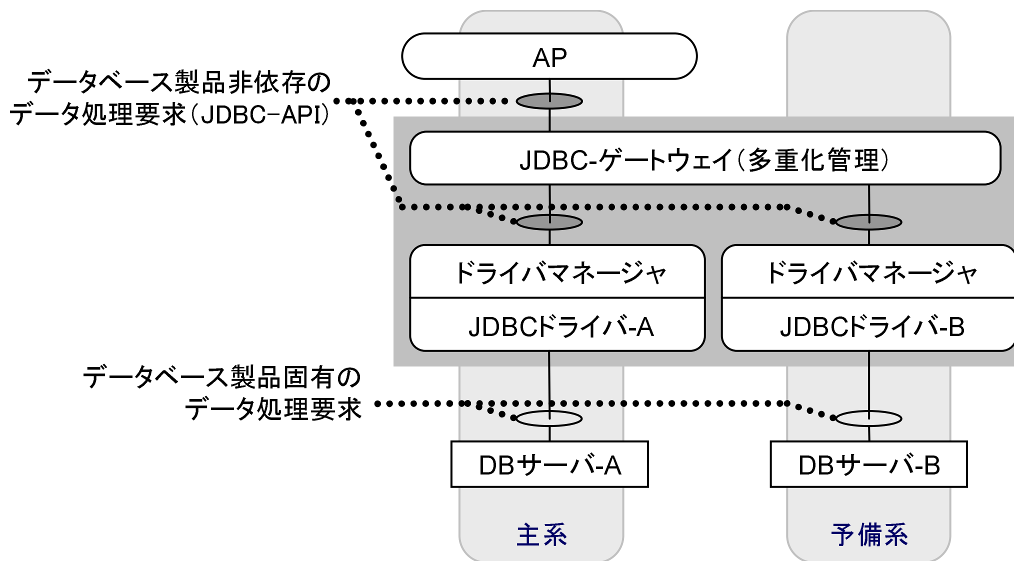


図 20 JDBC ゲートウェイ

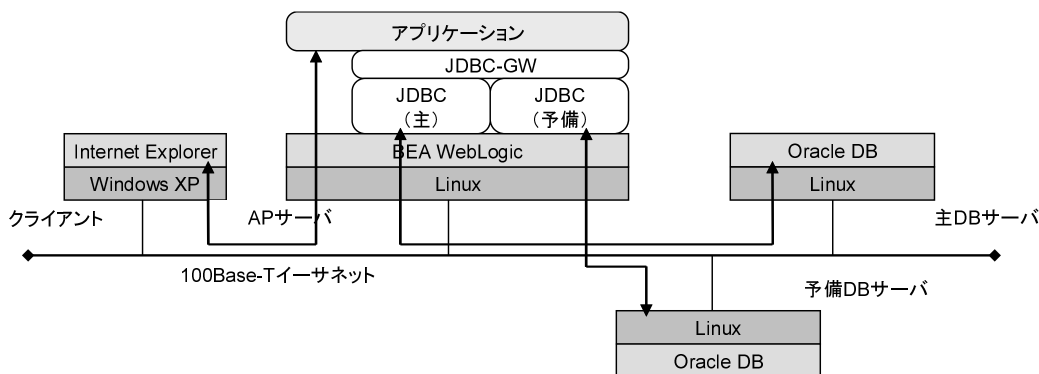


図 21　基本動作確認実験環境

3.4　基本動作

本節では実装したデータベース接続層の拡張によるデータベース複製方式の動作について述べる．本動作例の実行環境を図 21 に示す．

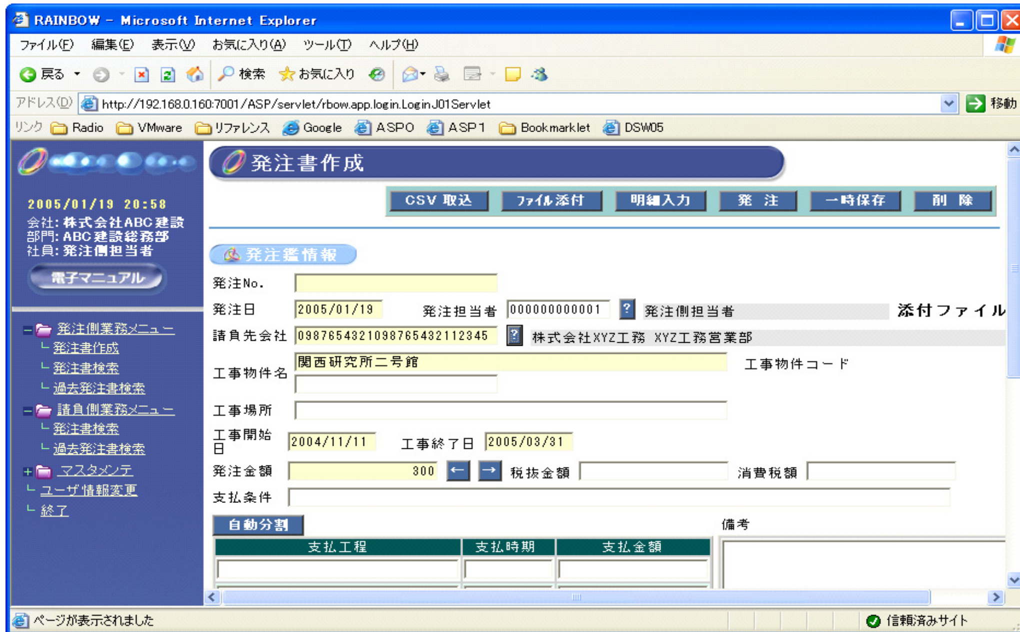
まず，アプリケーション・サーバ，主データベース，予備データベース，そしてクライアントの４台のコンピュータを 100Base-T イーサネットで接続した．そして，アプリケーション・サーバには， BEA 社 WebLogic Server 8.1，Java 実行環境として Sun JDK 1.41，アプリケーションとして NEC 住宅業向け工事発注 ASP サービス [62] を使用した．二つのデータベース・サーバには，それぞれデータベース・サーバとして Oracle9i Database Standard Edition を使用した．今回用いたアプリケーションは，いわゆる Web アプリケーションであるため，クライアントでは，Web ブラウザとして Microsoft Internet Explorer 6.0 を使用し，人手で操作を行った．

3.4.1 正常時の動作：データ複製

正常時にどのようにデータが複製されるのかを述べる．正常時の動作を図 22 に示す． 図 22 の上側に，クライアント操作の画面例を示す． クライアントで Web ブラウザを介して操作を行うと，操作に応じてサービス要求が発行される．次に，サービス要求を受けたアプリケーションが，JDBC を介して主データベースに対しデータ処理要求を発行しようとする．その際，JDBC ゲートウェイがデータ処理要求を多重化することで，同じ内容のデータ処理要求を主データベースと予備データベースの両方に発行する．そして主データベースと予備データベースは，同じデータ処理要求を受け取る．

二つのデータベース・サーバのネットワーク I/O を監視した結果を 図 23 に示す．なお，データベースが正しく複製されているかどうかは，一連の処理の後に，登録されたデータ数を調べ，データ数が同じであることを確認し，また登録されたデータの内容は目視で確認した．複製の検証を自動的に行い，もしエラーがあればただちに障害通知を行うことが期待されるが，3.1.3 節で述べたように，このような複製検証技術はいまだ発展途上であるため，今回は採用しなかった．

図 23 から分かるように，二つのデータベース・サーバにおいて，ほぼ同じパターンでネットワーク I/O が発生しており，つまり，アプリケーション・サーバからの処理要求が複製されて，二つのデータベースにと通信していることがわかる．その結果，両方のデータベース・サーバが同じデータ処理を行い，データが複製される．



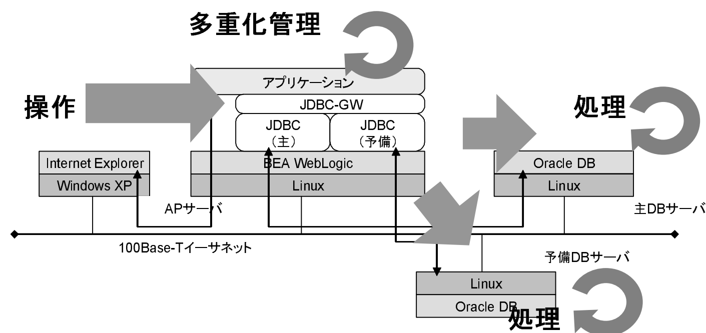
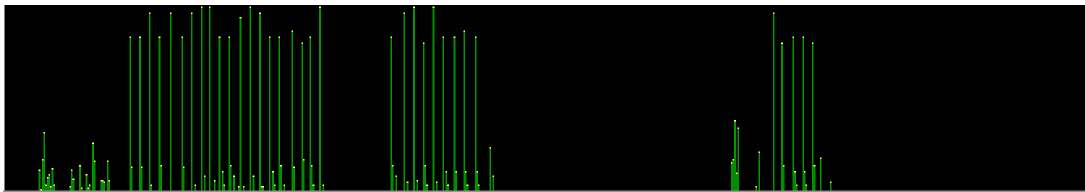


図 22 正常時のデータベース複製動作



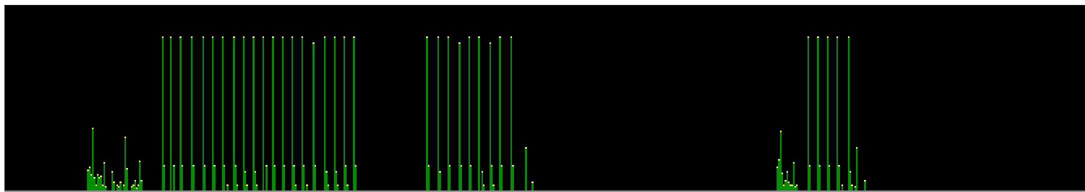


図 23 データベース処理要求アクセスの様子

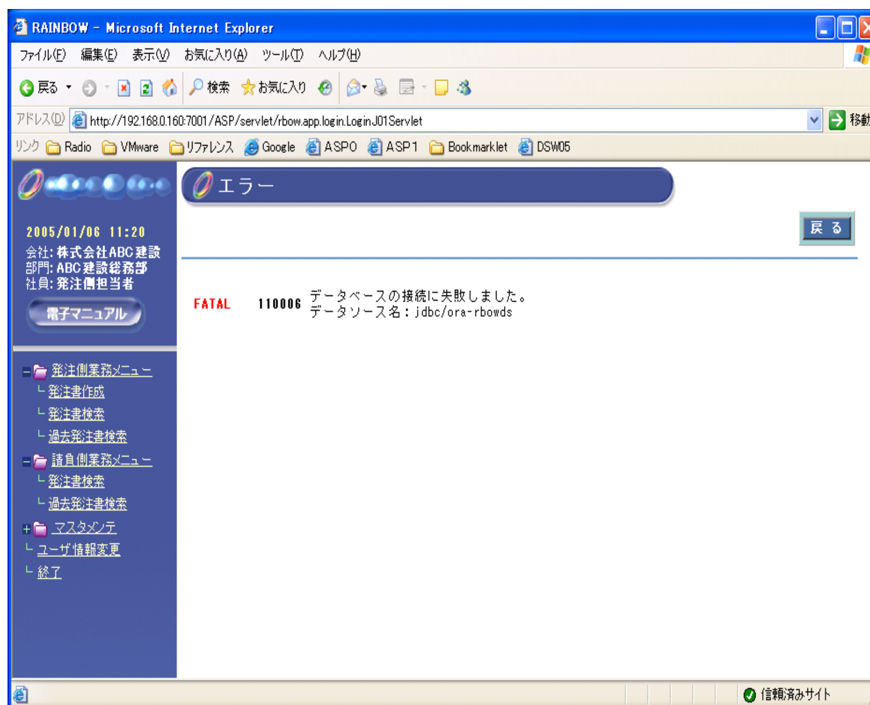


図 24 データベース接続エラー画面

3.4.2 障害時の動作：サービスの継続

次に，障害時のどのようにデータが保護され，またサービスを継続できるのかを述べる．

サービス提供中に，主データベースのプロセス異常終了 という障害が発生したと仮定する．提案機構を適用していないシステムでは，図 24 の画面に示すように，データベース接続エラーとなって，処理を継続できない，また，データベースを再起動しても，図 25 の画面に示すように，操作入力したはずのデータは失われてしまう．

一方，提案方式では，障害によって主データベースのデータが破壊されても，予備データベースには正常なデータが複製されて，保存されている．この予備側のデータを用いて，サービスの継続等が可能となる．障害時におけるサービス継続の動作を図 26 に示す．

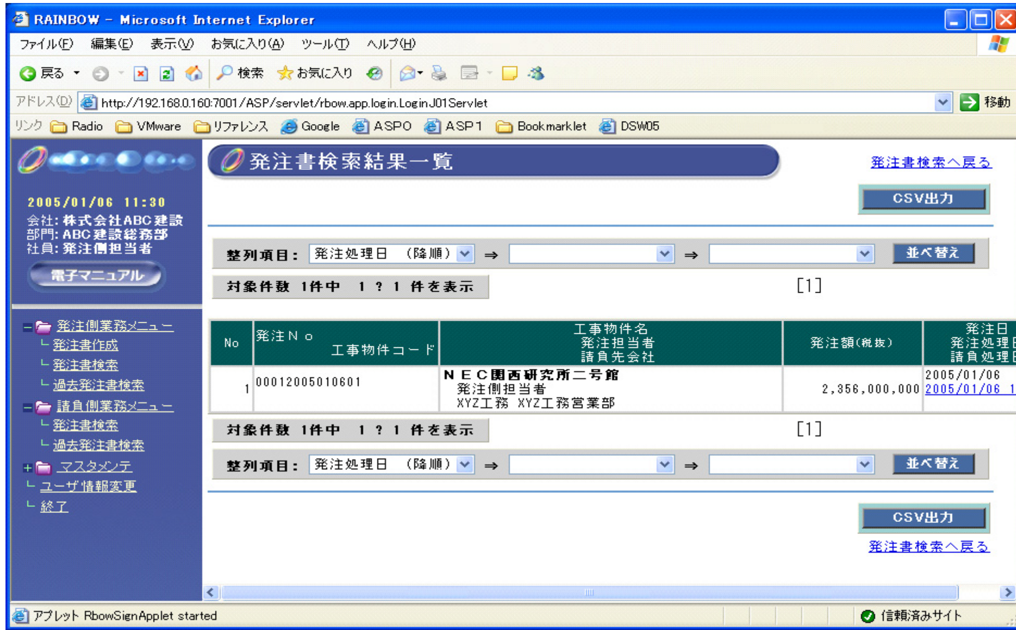
図 25 該当データなし画面

障害が発生していても，クライアントがサービス要求を発行すると，それを受けたアプリケーションは，正常どおりデータベース・サーバに対しデータ処理要求を発行する．そのデータ処理要求を JDBC ゲートウェイが多重化し，主データベースと予備データベースの両方に発行しようとする．しかし，主データベースは，障害のため応答が無いので，アクセスは失敗する．すると JDBC ゲートウェイは，以降，主データベースへはデータ処理要求を発行せず（切り離し），予備データベースに対してのみ，データ処理要求を発行するようにする．すなわちシステムは，予備データベースのみによる縮退運転状態となる．

図 26 の上側に，検索結果画面例を示す．データ複製機能により，予備データベースには主データベースと同様のデータが格納されているため，予備データベースは正常なデータ処理を行い，応答を返すことができている．

予備データベースからの応答は，JDBC ゲートウェイを経由して，アプリケー

ションに伝えられる．JDBC ゲートウェイを介しているため，アプリケーションからは，その応答が主データベースのものか，予備データベースのものかは意識されない．つまり，アプリケーションにとっては，主データベースの障害にかかわらず，データベース・サーバにデータ処理要求を発行したら，普通に正常な応答が返ってくる．すなわち，主データベースが停止したということは隠蔽され，無停止でサービスが継続される．



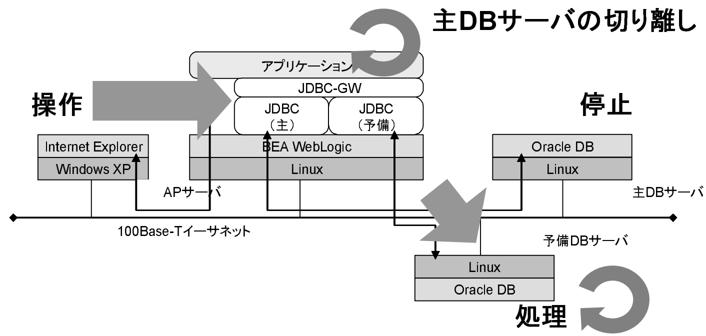


図 26　障害時のサービス継続動作

3.5 負荷分散構成

3.5.1 順序問題

前述のように，本提案手法では，データベース接続層を拡張し，データベース処理要求を多重化し，主データベース，予備データベースに同じデータ処理を行わせることでデータベース複製を実現する．このとき，二つのデータベースが同じ状態であるためには，データベース処理要求が同じ順序で投入される必要がある．たった1つのアプリケーション・サーバから構成されるシステムでは，たとえ複数のスレッドで動作していたとしても，データベース接続層がデータベース処理要求をシリアライズ管理するので，順序管理は不必要であった．

しかし，アプリケーション・サーバは，負荷分散のために，複数で構成されることが多い．この場合，複数のアプリケーション・サーバから，連続的に処理要求が発行されると，順序管理が必要となる．例えば，図27に示すように，複数のアプリケーション：AP-i，AP-ii があり，それぞれが発行するデータベース処理要求が拡張されたデータベース接続層 JDBC-GW-i，JDBC-GW-ii を介して主データベースと予備データベースに多重化されて，転送されるとする．ここで AP-i は Txi-1，Txi-2 の順で，AP-ii は Txii-1，Txii-2 の順でそれぞれデータベース処理要求を発行し，主データベースへの到達順序が Txi-1，Txii-1，Txi-2，Txii-2 だった場合，二つのデータベースが同じ処理を行うには，予備データベースにおいても，同じ順序でデータベース処理要求が到達しなくてはならない．

同一アプリケーション・サーバから発行される処理要求に関しては，主データベース，あるいは予備データベースと確立した同じコネクション内で発行されるデータベース処理要求のため，両方の DB での到達順序が同じであることは保証される．

しかし，異なるアプリケーション・サーバから発行される処理要求に関しては，異なるコネクションにおけるデータベース処理要求のため独立しており，その到達順序は保証されない．ネットワーク遅延等の影響によっては，図28に示すように，Txii-1 と Txi-2 の順序が入れ替わり，主データベースと予備データベースで到達順序が異なる場合がある．

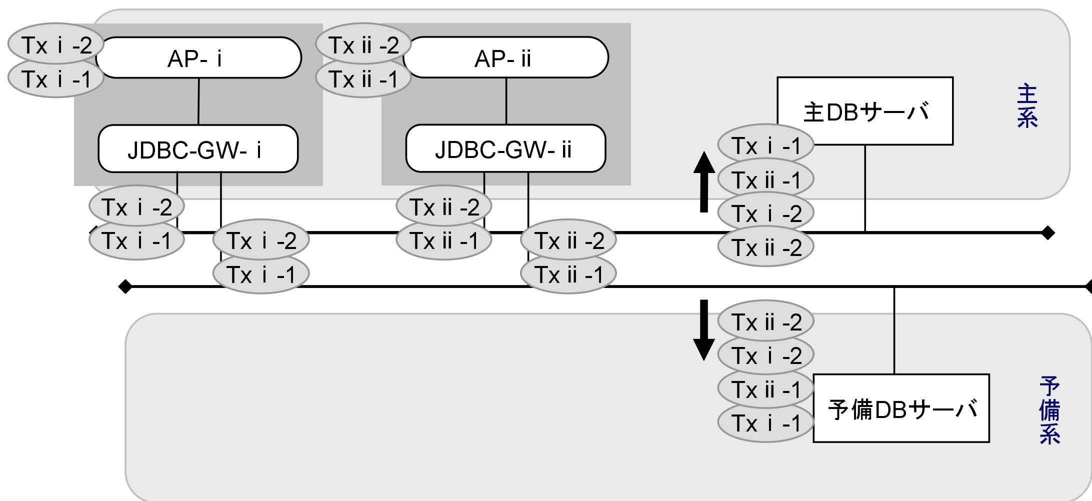


図 27　データベース処理要求の順序問題（正常）

到達順序，すなわち処理要求のデータベースへの投入順序が異なれば，データ処理の結果も異なり，二つのデータベースの状態が異なる可能性がある.

そこで，それぞれ独立したアプリケーションの拡張されたデータベース接続層間で，システム全体のデータベース処理要求の到達順序を保証するデータベース処理要求の順序制御が必要となる.

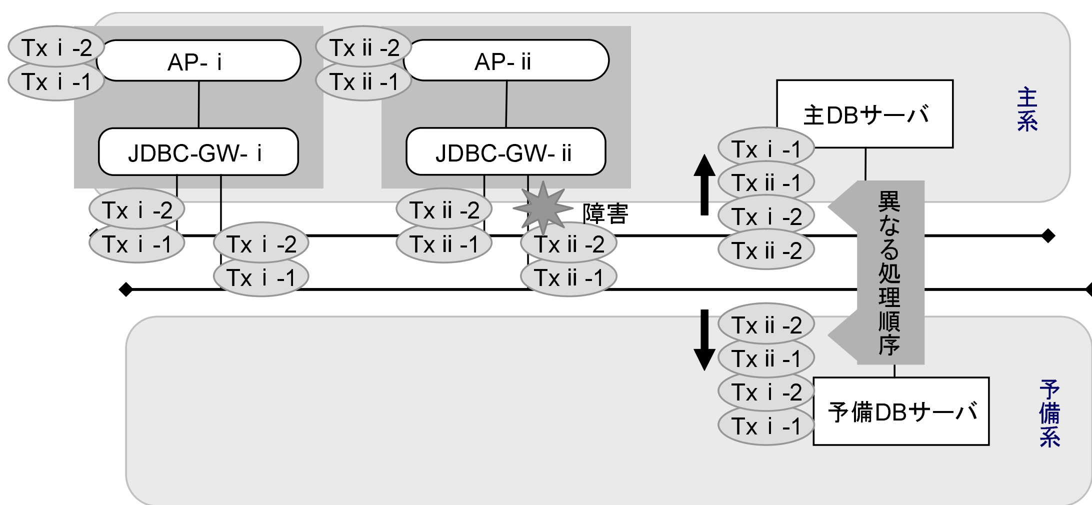


図 28　データベース処理要求の順序問題（異常）

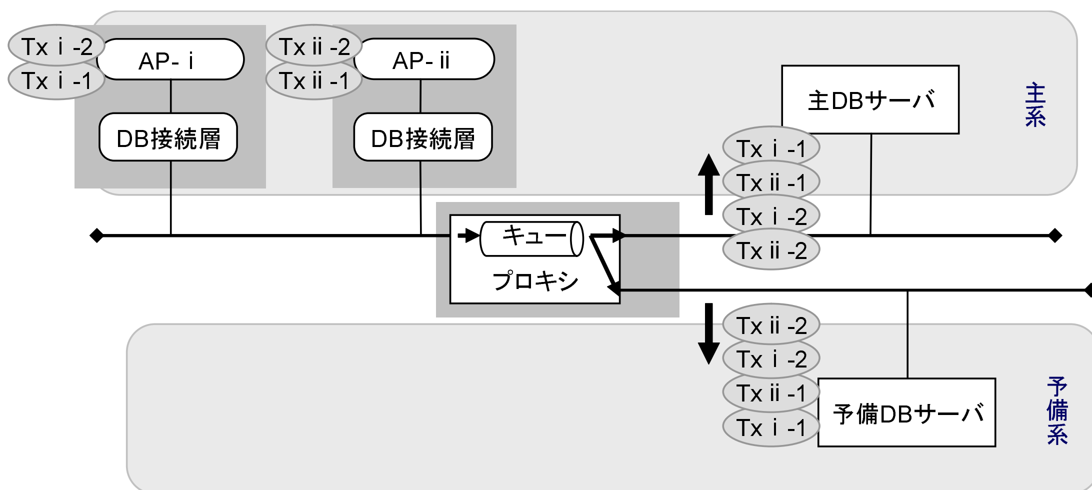


図 29 プロキシ型順序制御

3.5.2 従来方式

このような分散モジュール間で順序制御を実現するには，図 29 に示すように，pgtool[58] 等のように全てのデータベース処理要求が特定のプロキシ部を通過するようにしてプロキシ部でデータベース処理要求の順序制御を行うプロキシ方式，図 30 に示すように，全順序マルチキャスト [19] のように各分散モジュールが特定のシーケンスサーバに対し順序を問い合わせることで順序制御を行うシーケンスサーバ方式などがある．

プロキシ方式は，プロキシ部が単一障害点 (Single Point of Failure) となる．プロキシ部に障害が発生すると，順序制御のためにキューイングしているデータベース処理要求も失われてしまう．一方，シーケンスサーバ方式は，シーケンスサーバで障害が発生しても，データベース処理要求は失われないため，耐障害性の面で優れる．ただし，順序制御対象となるメッセージに送信元で順序番号を添付し，受信先で順序番号に基づき順序制御を行うため，通信プロトコル自体を拡張する必要がある．

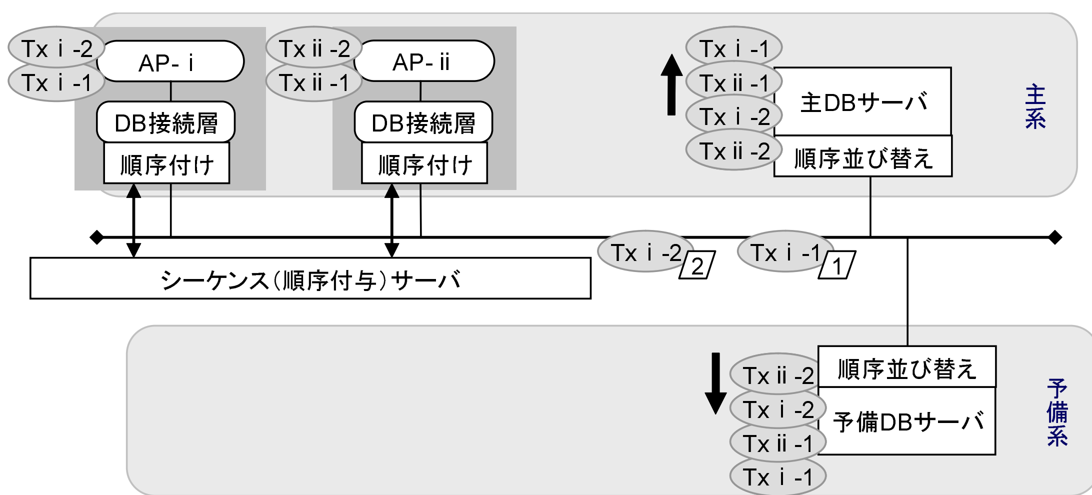


図 30 マルチキャスト型順序制御

3.5.3 提案方式：セマフォサーバ

本提案では，汎用性の観点から，送信元であるデータベース接続 API と受信先であるデータベース間のプロトコルは変更すべきでないと考え，図 31 に示すように，順序番号をキューイングしてデータベース処理要求の順序制御を行う手法を考案した．順序番号キューイングを行うサーバをセマフォサーバと呼び，以下のように動作する．

1. 各拡張データベース接続層は，データベース処理要求発行の際，セマフォサーバに対して発行許可を申請．
2. セマフォサーバは，申請された発行許可をキューイングし，キュー先頭の申請に対し，発行を許可．
3. 発行許可を与えられた拡張データベース接続層は，主データベース，予備データベースの 両方にデータベース処理要求を発行し，両方からの応答が返ってきたら，発行完了の旨をセマフォサーバに通知する．
4. 完了通知を受けたセマフォサーバは，その申請を削除し，次の申請に対し，同様の処理を繰り返す．

このようにすることで，ICT システム全体におけるデータベース処理要求の到達順序を保証する．

また，主データベースもしくは予備データベースに障害が発生した場合，各データベース接続層は，故障したデータベース・サーバを切り離してサービス継続する．フェイルバック（冗長構成への復旧）は，3.2 節で述べたように，アクセスログを一時的に保持するバッファをセマフォサーバに設け，同様の手法で冗長構成に復帰する[4]．

あるアプリケーション・サーバで，いずれかのデータベース接続に障害が発生した場合は，処理中のアクセスが commit のような順序管理されたアクセスであるか否かでフェイルオーバーの方法が異なる．順序管理されていないアクセスで停止した場合，障害を発生したアプリケーション・サーバのデータベース接続層

[4] ただし，縮退運転中のセマフォサーバは，単一障害点となる問題がある．

は，アプリケーションに対してはエラーを返し，当該トランザクションは破棄する．そして，障害を発生したアプリケーション・サーバを系から切り離す．もし，10 台のアプリケーション・サーバで運用しているのであれば，残りの 9 台のアプリケーション・サーバは通常通り動作し，サービスを継続する．

もし，処理中のアクセスが順序管理されている場合，当該トランザクションを破棄，もしくはロールバックするといった処理は，他のアプリケーション・サーバの処理状態で困難となる可能性がある．そのため，この場合には，片方のデータベースを用いて縮退運転状態でサービスを継続する．上記の異常のあったアプリケーション・サーバで，もし切り離しに失敗した場合には，いずれ順序管理されるアクセスがあるために，縮退運転状態となる．

縮退運転状態になった後，セマフォサーバにその情報は通知される．そして，他のアプリケーション・サーバも，順序管理すべきアクセスにあった時点でセマフォサーバにアクセスし，エラー情報を受信し，縮退運転となる．

縮退運転中のセマフォサーバによる順序制御は無意味となるが，システム上の問題を生じることはない．

アプリケーション・サーバに障害があったとしても，セマフォサーバは，発行要求に対する応答処理であり，障害が発生したアプリからの発行要求がないだけであり，システム上の問題を生じることはない．

もし，セマフォサーバに障害が発生した場合には，セマフォサーバのフェイルオーバーが必要である．システム上に複数のセマフォサーバが存在してはいけないので，問題のあるセマフォサーバを完全に停止させる必要がある．そして，各アプリケーション・サーバは，許可待ち状態であるが，その許可待ちはタイムアウトとなり，再度セマフォサーバに処理要求を行うことになる．セマフォサーバが停止中は，アプリケーションの処理も停止するが，主データベースと予備データベースは同じ状態が保たれる．セマフォサーバが再起動すれば，当初のアプリケーションの発行要求順とは異なる可能性はあるが，整合性が保持された状態で，サービスは継続できる．また，セマフォサーバは，発行許可を与えるだけなので，状態保存も不要であり，フェイルオーバーも容易である（ステートレスなフェイルオーバーである）．

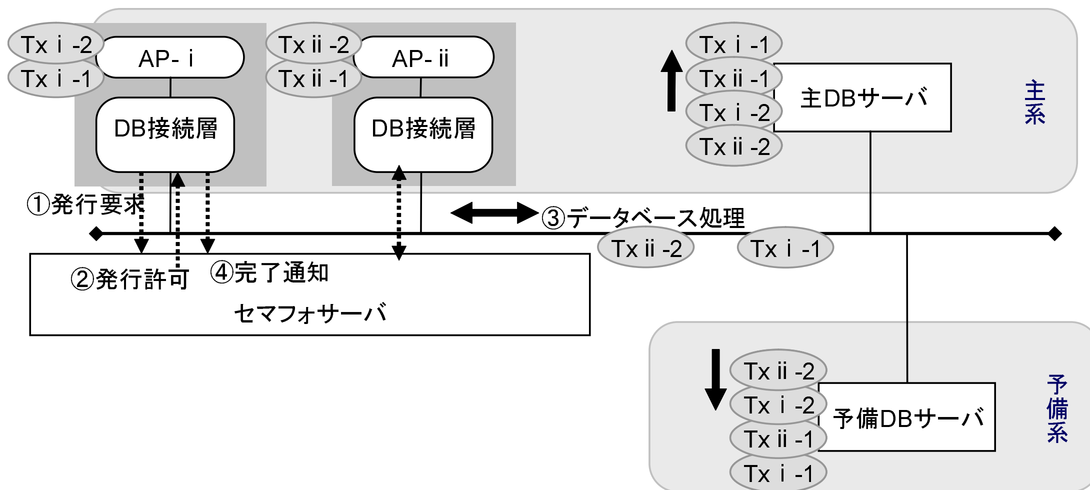


図 31 セマフォサーバ

一方，セマフォサーバを用いることで，性能が著しく低下することが予想される．すべての処理をシーケンシャルに実行していたのでは，データベースの性能を十分に引き出すことは不可能である．そこで，全てのメソッドを順序制御対象とするのではなく，順序が入れ替わることでデータ不整合が発生する可能性があるデータベース処理要求のみを順序制御対象とする．JDBC の場合は，次の 7 種類の処理が対象となる．

- DriverManager.getConnection
- Connection.commit
- Connection.rollback
- Statement.executeQuery
- Statement.executeUpdate
- PreparedStatement.executeQuery
- PreparedStatement.executeUpdate

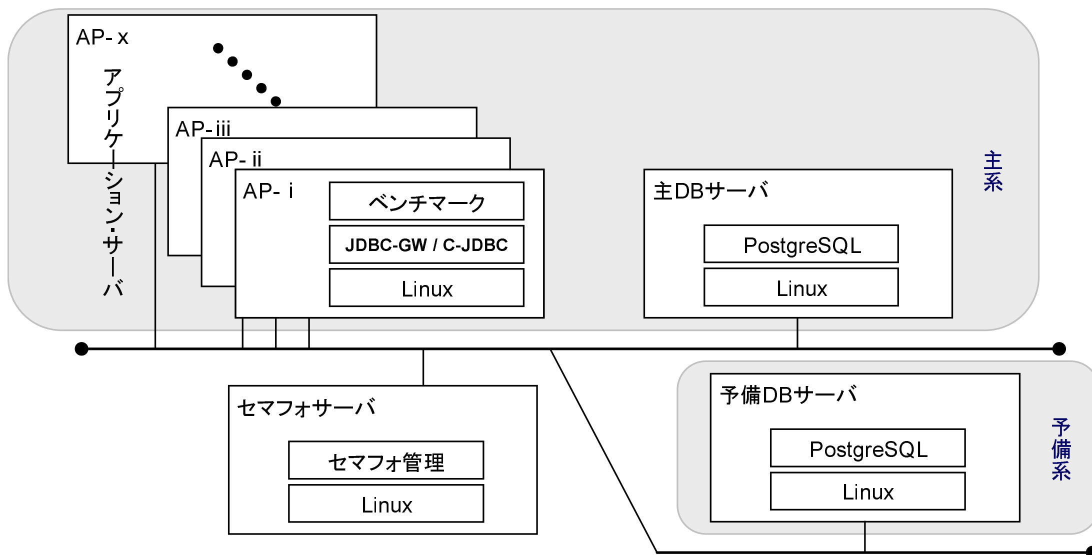


図 32 負荷分散構成評価実験構成

3.6 負荷分散構成における性能評価

データベース処理要求の順序制御を行う機構の評価のために，発行許可の申請を行う機能を JDBC ゲートウェイ に組み込み，また，発行許可を与えるセマフォサーバを，各 JDBC ゲートウェイ から独立したモジュールとして，ICT システムのネットワーク内に配置した実験システムを構築した．評価実験を行ったシステムの構成を図 32 に示す．

10 台のアプリケーション・サーバ：AP-i〜AP-x，主データベース，予備データベース，セマフォサーバをそれぞれギガビットイーサで接続した．データベース・サーバにはそれぞれ PostgreSQL 7.4 をインストールし，AP-i〜AP-x には，Java アプリケーション として TPC-C[41] をベースにした OLTP ベンチマークツール [15] をインストールし，データベースアクセスを行うようにした．

データベース複製機構がない場合と，従来のデータベース複製機構として，C-JDBC を適用した場合と，本論文で提案する手法 JDBC ゲートウェイを適用した場合とで動作および性能の比較を行った．

障害発生時のシステムの振る舞いは，下の通りであった．

表 7 負荷分散構成での性能評価実験結果

同時接続数		1	10	20	30	40	50	60	70	80	90	100
Case1	性能(Tx/秒)	17.67	15.64	14.24	14.12	13.93	13.65	13.93	13.65	13.97	13.84	13.86
(未適用)	性能比(%)	100	100	100	100	100	100	100	100	100	100	100
Case2	性能(Tx/秒)	14.6	13.37	12.61	13.09	12.95	13.21	13.16	13.08	13.04	12.64	12.82
(C-JDBC)	性能比(%)	82.63	85.49	88.55	92.71	92.96	96.78	94.47	95.82	93.34	91.33	92.5
Case3	性能(Tx/秒)	16.05	13.43	13.61	13.59	13.69	13.41	13.9	13.83	14.25	14.44	14.28
(JDBC-GW)	性能比(%)	90.83	85.87	95.58	96.25	98.28	98.24	99.78	101.3	102	104.3	103

未適用時 ベンチマークは停止，ベンチマークを再起動しても継続不可

C-JDBC ベンチマークは停止．ベンチマークを再起動すれば実行可能

JDBC-GW ベンチマークは継続

C-JDBC では，必要最小限のデータベース処理要求のみが転送されているものと思われる．一部のデータベース処理要求は，データベースの更新は実行しないが，データベース内部の状態管理に何らかの関与をしており，これらのデータベース処理要求が予備データベースに送信されていないと，この内部処理状態まで含めた二つのデータベースの状態を同じには保つことができていないと思われる．そのため，C-JDBC では，予備データベースだけでの処理の継続はできないが，アプリケーションを再起動すれば，継続可能状態からデータベース処理を始めるので，正常処理となる．

未適用時，C-JDBC，JDBC-GW の各ベンチマーク結果を図 33 に示す．Case1 は，データベース複製機構がない場合，Case2 は，C-JDBC を使ってデータベース複製を行った場合，Case3 は，本論文提案した JDBC ゲートウェイを使ってデータベース複製を行った場合である．

Case1 では，同時接続数が増えるにつれ，処理性能は低下している．各接続に対し，データの到着の有無を調べるなど，接続の管理処理が増えるためである．

Case2 の場合では，複製処理のオーバーヘッドのために，処理性能は Case1 に比べ低下する．同時接続数が小さい間は，劣化率は 83 %から 96.7 %に改善している．これは，データ接続数が大きくなると，未適用時の処理性能が低下し，一処理要求あたりの処理時間は長くなり，相対的に複製処理のオーバーヘッドが小

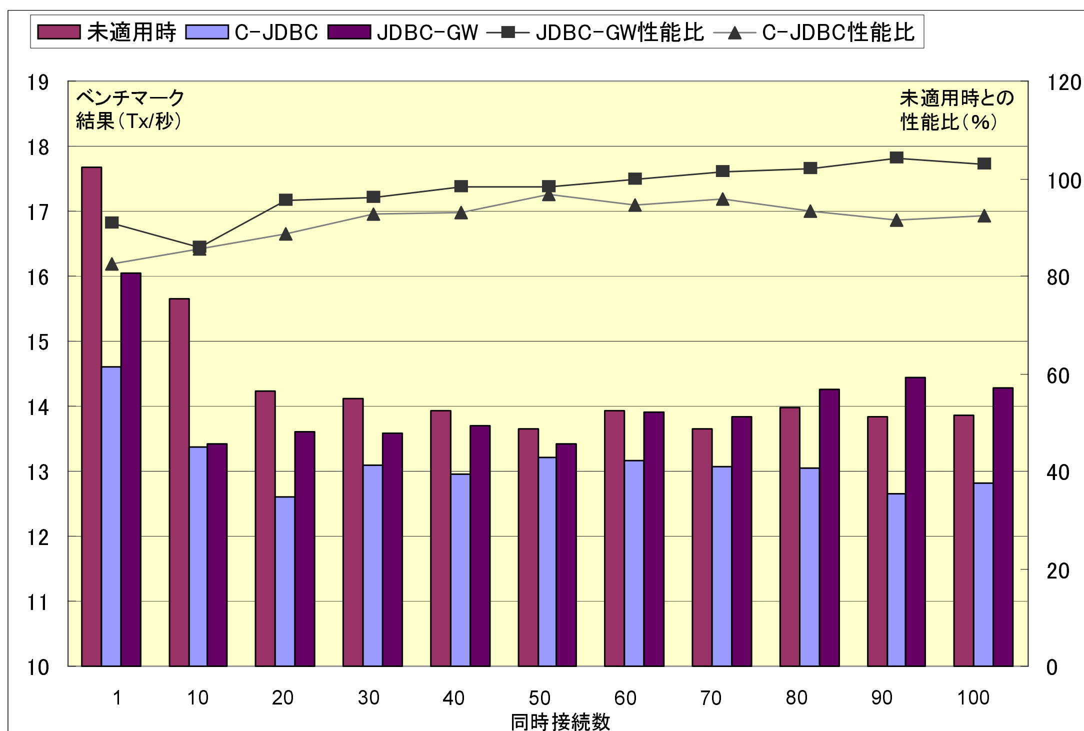


図 33　負荷分散構成評価実験結果

さくなっているためである．しかし，同時接続数がさらに増加すると劣化率は悪化する．C-JDBC の場合，複製処理が集中管理のため，その集中管理部がボトルネックとなっていると思われる．

Case3 の場合では，同時接続数が小さい間は，Case2 と同様である．しかし，同時接続が大きくなっても，複製処理が分散管理のため，ボトルネックとならない．順序制御は集中管理となるが，その処理はシンプルであり，ボトルネックとなるほどの処理量ではない．Case3 の同時接続数が 70 以上では，Case1 の複製処理を行っていない場合より，処理性能が上回るという事象が見られる．Case1 は，データベース処理要求はデータベース・サーバ側に溜まり，データベース・サーバ側でキュー管理をする必要があるのに対し，Case3 では，アプリケーション・サーバ側で処理要求が抑制され，データベース処理要求がデータベース・サーバ側で

溜まることは少なくなる．そのためデータベース・サーバ側でのキュー管理処理負荷が軽減したものと思われる．

このように，提案手法では，データベース接続層のライブラリを交換することで，データベース複製機能を追加することが可能である．データベース複製のオーバーヘッドによるシステムの性能劣化はあるが，クライアント数が大きい場合には性能劣化はほとんどなく，無視できる可能性がある．本提案手法は，負荷の大きいデータベース・サーバで複製処理を行うのではなく，負荷分散が可能なアプリケーション・サーバで処理を行っているためである．本提案手法は，スケーラビリティの面で有利であると考える．

また，信頼性（可用性）については，表 6 に示したように，提案方式は AP〜DB 間での多重化管理であるので，計算式は $\frac{T}{T+Ps\times Td\times 2+Pd\times Td\times 2+Pa\times Ua\times Ta\times Trb}$ となる．性能向上により，Ua（アプリケーションの処理中率）は明らかに低下し，また Ta（アプリケーションのタイムアウト時間）を低減できる可能性がある．ロールバック不要，各タイムアウト時間１秒，各障害の発生率は 0.01 %，処理量を毎秒 10 トランザクションと仮定して，可用性を算出する．処理中率を同時接続数１の実験結果から求めると，従来方式では，14.6 トランザクション/秒であるので処理中率は 0.685，提案方式では，16.1 トランザクション/秒であるので，処理中率は，0.621 となる．上式に各値を代入して，可用性を求めると，従来方式では 99.95317 %，提案方式では 99.95379 %となる．その差は，0.00062 %である．可用性に関しては，ほぼ従来方式と同等であることが確認できた．

4. ディザスタリカバリ機構への適用

ディザスタリカバリとは，自然災害等により，ICT サービスを提供するサイトの１つが機能停止となった場合でも，別のサイトでサービスを継続するソリューションである．地震，台風等の影響が同時に及ばないように，各サイトは遠隔に配置される．本章では，ディザスタリカバリへの適用を踏まえたデータベース複製について議論する．

4.1 遠隔地への同期複製

近年，ICT システムがビジネスに不可欠となり，災害，破壊活動などのためにシステムが停止しても，別のサイトにおいて，同じサービスを継続させることが求められている．金融やキャリア向けのシステムにおいては，高いコストをかけて，専用の耐障害システムが構築されていることが多い．しかし，多くのシステムでは，テープなどの別メディアに予備を作成し，災害・障害発生時には，予備からデータを復元するのが一般的であり，通常の運用管理および障害復旧は，時間と労力がかかる作業である．そこで，通常時の運用サイト（主サイト）が停止した場合に，別サイト（予備サイト）において同一サービスを迅速に引き継ぐ復旧方式がいくつか提案されている．特に，主サイトと予備サイトとの間でデータを同期して更新するデータ複製を行い，障害時には複製データを用いてサービスを継続する方式が注目されている．

既存の同期複製技術は大きく Primary Copy と Update Anywhere の２つに分けられる [43]．一般に Update Anywhere は高信頼であり，Primary Copy は高性能とされる．Primary Copy 方式の中には，完全同期の方式もあれば，一部の予備系のみ同期方式で，その他の予備系は非同期方式であるような方式もある．

遠隔地にデータを同期複製する場合，問題となるのは通信の遅延があることである．遠隔地へのデータの書き込みを確認していたのでは，システムの処理性能は大きく低下する．処理性能の低下を防ぐには，遠隔地へのデータの更新を非同期とすればよいが，それでは主サイトと予備サイトで異なる状態となり信頼性は低下する．

信頼性は，RTO および RPO を指標とし，できる限りそれらを最小化することが目標である．しかし，処理性能と信頼性とは，通常トレードオフの関係にあり，信頼性を良くすれば，性能は悪化する．耐障害機構としては，多様なサービスやアプリケーションの信頼性の要件は確保しながら，性能を最大化することが重要である．

そこで，システムの処理性能と信頼性とのバランスを制御可能とし，高い柔軟性を持ちながら，性能を最大化するために，Primary Copy と Update Anywhere の２つの方式のハイブリッド方式を提案する．つまり，信頼性の高い Update Anywhere 方式を基にしながら，すべての処理を１つ１つ同期的に処理することはせず，処理状況（処理の意味）に応じて，複製方式を切り替えるというものである．

提案方式では，各処理要求の意味に応じて複製方式を制御し，複製するデータサイズそのものを削減することで，ネットワーク遅延とパケット損失等の影響を小さくし，遠隔ネットワークにおけるデータ複製を可能とする．さらに，複製するデータサイズが小さくなれば，ネットワークだけでなく，CPU 性能，メモリ等の要件が低くなり，ハードウェアのコストを抑えることも可能である．

以下，本章は次のような構成で議論を進める．4.2 では，既存のデータ複製方式を鑑み，遠隔地への複製の課題と解決方針を述べる．4.3 で提案するシステムの構成と動作を説明し，4.4 でディザスタリカバリへの適用を説明する．4.5 では，性能評価のための実験および結果を示す．最後 4.6 で本章のまとめを述べる．

4.2 遠隔地への複製の課題と解決方針

停止した主サイトのサービスを予備サイトで継続するためには，予備サイト側で主サイトと同じデータを保持する必要がある．そして，いかにサービスを停止することなく継続してサービスできるかは，主サイトと予備サイトがどのくらい正確に同期できているかに依存する．

既存の遠隔地への複製では，ストレージレイヤにおいて複製処理をする製品を利用することが多い．装置内部にデータ複製機能を有し，広帯域の専用線で接続された異なる場所のストレージ装置との間で同期複製を行うが，広域ネットワークのスループットは，ストレージへのアクセス I/O スループットより低いため，

複製処理は大きな性能劣化を引き起こす可能性がある．上位レイヤでは１つのコマンドであっても，下位レイヤでは，OS やストレージの管理・制御のために，高頻度に大量のデータをやり取りしていることもあり，遠距離通信のオーバーヘッドによる性能劣化の影響を受けやすい．

また，ストレージレイヤの複製では，復旧処理の信頼性にも問題がある．予備サイトで保持されているデータは，再開可能な状態で保存される必要があるが，その保証が困難なことである．例えば，データの更新途中で，主サイトの障害が発生した場合，その更新途中のデータは，最後まで更新されるか，更新前に戻るかのいずれかの状態となることが少なくとも必要である．このような原子性の保証と呼ばれるデータ管理は，通常 OS 層より上位で実現される．

一方，前章までに述べてきた提案手法では，データベース接続層を拡張し，データベース処理要求を多重化し，両方の系で，同一のデータ処理を行う． 従来のデータベース・サーバの複製と異なり，データベース・サーバへの負荷も少なく，データベース接続ライブラリを交換するだけで，アプリケーション・サーバを冗長構成に対応させることができるので導入も容易である．

さらに，ディザスタリカバリ機能に対応した遠隔地への複製のためには，遠距離通信のオーバーヘッドによる性能低下をできるだけ抑えることが重要となる．データベース接続層では，SQL レベルのデータベース処理要求がモニタリング可能であり，処理要求に応じて複製するか否かを制御し，サービスの信頼性要件に応じた必要最小限の処理要求のみを同期処理とすれば，遠距離通信による性能低下を抑えることができると考えられる．

本章の以降の節では，このような解決方針をベースに柔軟な複製機能を実現する機構を説明する．

4.3 ディザスタリカバリ向けデータベース複製システム

4.3.1 システム構成概要

データベース接続ライブラリとは，アプリケーション・サーバのデータベース接続層において，データベース・サーバとの接続手続を仮想化することで，データ

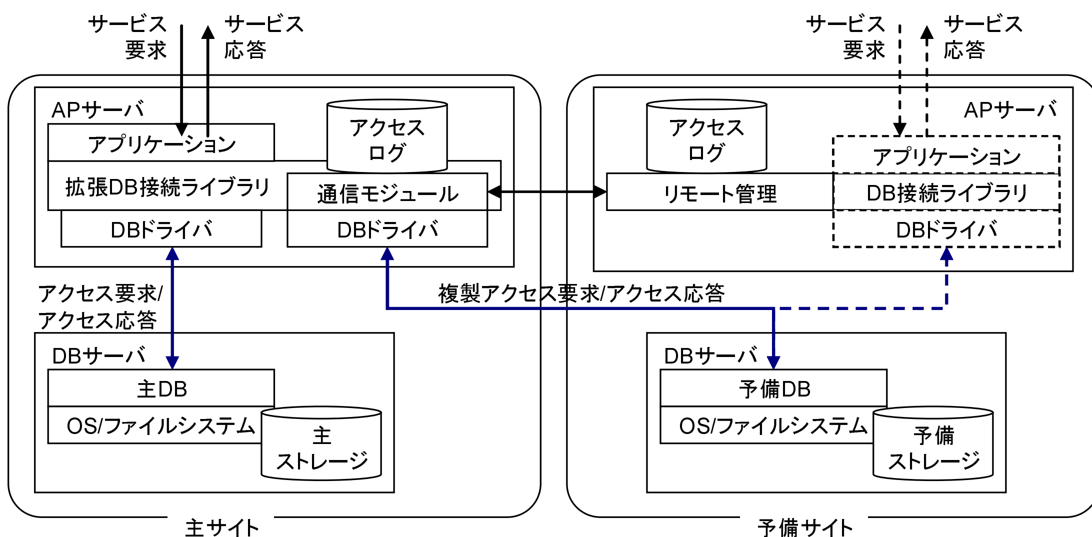


図 34 提案システム構成の概要

ベース・サーバの種類に依らない統一的なAPIを提供する．JDBC[27]，ODBC[13]，ADO.NET[34] などがデータベース接続ライブラリとしてよく知られている．本提案では，データベース接続ライブラリを拡張し，アプリケーションのデータベース処理要求を複数のデータベース・サーバに送信し，またそれらからの応答を受け取り，1 つの応答のみをアプリケーションに返すようにシステムを構成する．

図 34 に提案するシステム構成の概要を示す．システムは，主サイトと予備サイトから成り，両サイトは，アプリケーション・サーバとデータベース・サーバを含む．主サイトのアプリケーション・サーバは，サービス要求の受信とサービス応答の返信を行うアプリケーション，サイト間通信モジュールを備えた拡張データベース接続ライブラリ，ローカルのデータベース・サーバと接続するデータベースドライバ，遠隔の予備サイトのデータベース・サーバと接続するデータベースドライバから成る．サイト間通信モジュールは，アクセスログ（データベース処理要求・応答のログ）を予備サイトに送信することと，データベースドライバを介して，予備サイトのデータベース・サーバとデータベース処理要求・応答を通信することを担う．なお，アクセスログの保存は，シーケンシャルに追記する処

理であり，通常のデータベース処理と比べ，高速に処理が可能である [47]．本論文では，アクセスログの保存機構は，別途提供されるものとし，議論の対象としない．

予備サイトのアプリケーション・サーバには，災害時に有効となるアプリケーション，予備サイト内のデータベース接続ライブラリおよびデータベースドライバがスタンバイされている．リモート管理は，通常時は主サイトからアクセスログを受取り，保存する．障害発生時には，復旧処理を担う．主サイトと予備サイトのデータベース・サーバは，データベース，OS（ファイルシステム），ストレージから構成される．

4.3.2 データベース複製処理

主サイトのアプリケーション・サーバ内の拡張データベース接続ライブラリは，複数のデータベース接続を行い，接続の１つは，データベースドライバを介して，主データベースに接続し，別の接続は通信モジュールにリンクする．通信モジュールは，データベースドライバを介して，予備データベースに接続する．両サイトのデータベース・サーバが論理的に同一の状態を保持するように，主データベースに送信するデータベース処理要求を，同時に予備データベースにも送信する．

一般に，データベース・システムを利用する場合，connect（データベースへの接続），execute（データ処理の実行），commit（データ処理の確定）， close（データベースとの接続断）の順にデータベース処理要求が進む．各データベース処理要求は，複数のデータベース処理要求を含むが，処理要求の種類によっては，予備データベース側で同時に処理する必要はない場合もある．一連のデータベース処理要求，時間，処理数等をアクセス状況と呼び，そのアクセス状況に応じて，異なる複製方式で処理を実行する制御機構を設ける．本論文では，通常処理，スキップ処理，順序確定処理，同期処理の４種類の複製方式を用意した．４種類の複製方式のアクセス手順を図 35 に示す．

通常処理では，データベース接続ライブラリは，アプリケーションからのデータベース処理要求を受け，主データベースと予備データベースの両方にアクセスする． 主データベースからアクセス応答を受け取った後に，アプリケーションに

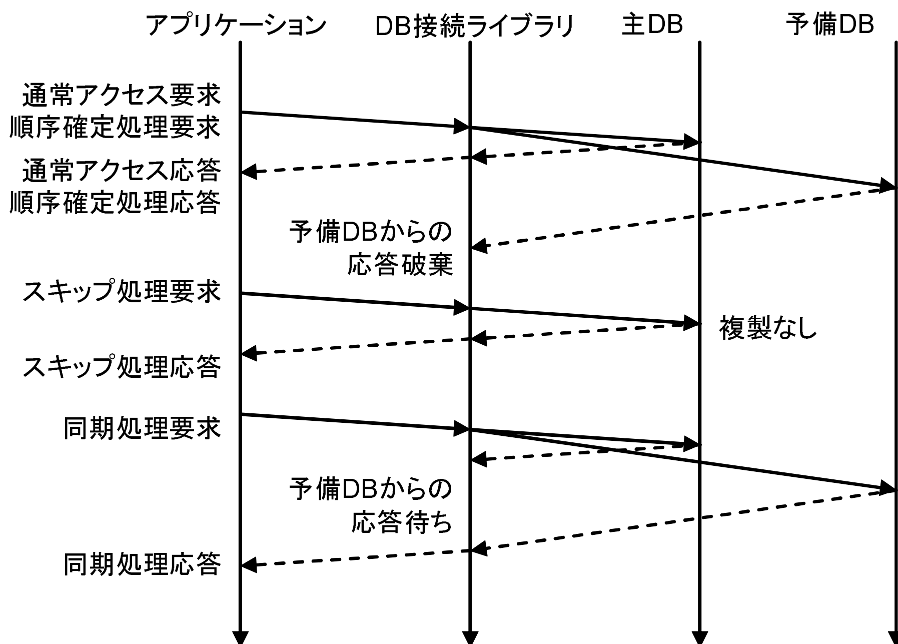


図 35 複製処理のアクセス手順

返す．予備データベースからのアクセス応答も受け取るが，そのアクセス応答は破棄される．

スキップ処理では，データベース接続ライブラリは，アプリケーションからのデータベース処理要求を受け，主データベースのみにアクセスし，予備データベースにはアクセスしない．

順序確定処理では，基本的に通常処理と同じであるが，処理順序を確定して処理を進める点で異なる．3.5 節で述べたように，主データベースと予備データベースとで，データベース処理要求の順序が異なる可能性があり，両データベースで処理順序が同じになることを保証する場合に適用される．順序確定処理では，前述したように，セマフォサーバと呼ばれる順序管理サーバと通信し，処理要求発行許可を待って，データベース処理要求を進める．

同期処理では，主 DB と予備 DB の両方に，同時にアクセスするようにアクセスを制御する．さらに両方のデータベースから，アクセス応答を受取り，そのア

クセス応答をアプリケーションに返す．

4.3.3 データベースアクセス状況

本提案方式では，順序確定処理や同期処理が必要なシステム状態およびイベントログをアクセス状況としてあらかじめ登録しておき，実際にアクセスログなどを監視して得られたシステム状態及びイベントログと，登録されたアクセス状況とを照合して合致した場合に，順序確定処理，あるいは同期処理を行う．アクセス状況には，データベースへのアクセスの種類，操作テーブルの種類，セッションのユーザ ID，グループ ID，アクセス元（IP アドレス），日時などがある．

順序確定処理，同期処理を高頻度で実行すれば，システムの信頼性は向上するものの，順序確定処理は並列的に処理できないため，また同期処理は遠隔地の予備 DB の更新確認を待つために，システム処理性能を劣化させる．登録するアクセス状況を変更することで，順序確定処理，同期処理の頻度を管理し，システム処理性能と信頼性とのバランスを柔軟に制御する．システムによっては，通常時の処理性能を重視し，順序確定処理，同期処理を厳密に行わず，RTO，RPO の信頼性を犠牲にするという設定も可能である．

データベースアクセス状況は多様であり，また各アクセスの意味の理解は困難であることから，アプリケーションプログラムの詳細を知らない運用者が，アクセス状況と複製処理の関連付けを直接記述することは困難な作業である．実際の運用場面においては，システム構築ベンダがルールセット，もしくはアプリケーション種類別のテンプレートを提供するものとする．なお，次章において，機械学習を用いて，アクセス状況と複製処理の対応付けを自動化する手法について，詳細に述べる．

4.3.4 複製処理時の障害

本節では，ディスク障害など日常発生する障害に対する処理について述べる．具体的には，いずれかのデータベースの応答がなくなる場合と，いずれかのデータベースがエラーを返す場合について議論する．ここでは，本提案で独自拡張を

行っている複製制御機構に関連する障害についてのみ言及し，より大規模な障害への対応については4.4節で述べる．

いずれかのデータベース・サーバで障害が発生し，アクセス応答がない場合，データベース接続ライブラリにおいて，そのデータベース・サーバとの通信はタイムアウトし，片方のデータベース・サーバのみで処理を継続する．障害発生中は，そのデータベース・サーバへのデータベース処理要求をセマフォサーバに保存しておく．アプリケーション・サーバが１つの場合には，データベース処理要求の順番を考慮する必要がないので，データベース処理要求はアプリケーション・サーバ内にローカルに保存すればよい．データベース・サーバが復旧した際には，そのデータベース処理要求を使って，障害発生中のデータベース処理を再現する．すべての保存したデータベース処理要求を処理した時点で，サービスを一時停止し，通常の複製処理構成に戻す．

障害発生中のデータベース処理を再現する間も，片方のデータベース・サーバでサービスを継続し，データベース処理要求を保存するが，再現処理が遅いと通常の複製処理構成に復帰できない．このような場合には，アプリケーション・サーバ側でサービスを制限するなどの対処を行う必要がある．

障害が長時間になると，データベース処理要求を保存するサイズが大きくなり，ディスク容量が不足，また障害発生中のデータベース処理の再現に長時間を要するなどの問題がある．その場合には，チェックポイント，スナップショット等のデータの保存・復元技術を利用する．

いずれかのデータベース・サーバで障害が発生し，エラーを含むアクセス応答があった場合，２つのアクセス応答だけでは，どちらが正しいアクセス応答かは分からない．本提案の同期処理では，どちらかのアクセス応答を使って処理を進めることはせず，アプリケーションにエラーを返し，両データベースはロールバックする．同期処理以外では，異なる応答があっても主データベースの応答のみで処理は進む．異なる応答があった場合に処理を進めては不都合となるデータベース処理要求には，同期処理を行う必要がある．

上述した方式は，正しい応答が含まれていてもエラーとするため，処理性能は劣化する．異なる応答の中から，確からしい応答を使って処理を進めるには，三

つ以上のデータベース・サーバを用意し，多数決で決定するなどの方式がある．

4.4 基本動作

本提案システムでは，自然災害による地域的な停電，ネットワークの停止など，主サイト全体が停止する場合，予備サイトにおいて，主サイトで提供していたサービス環境を引き継ぎ，予備データベースを使って，同一サービスを提供する．本節では，その手順を詳細に述べる．

4.4.1 フェイルオーバー

災害が発生し，アプリケーション・サーバが停止した場合，クライアントとのセッションは失われ，予備データベースにあった未確定のデータベース処理要求は破棄される．アプリケーション・サーバの製品には，クライアントの処理状態を保存し，その状態を予備サイトのアプリケーション・サーバに引き継ぎ，データベース・サーバ側も，データベース処理要求処理を継続する機能をもつものもある．しかし，クライアントの処理状態を引き継がない方がシステムの実装は容易であり，オーバーヘッドも少ないことから，アプリケーション・サーバの状態保存はいまだ利用は少ない．

処理状態を引き継がない場合でも，確定処理の途中で災害が発生した場合には，クライアントと予備データベースとで操作（処理）の確定，未確定の状態が異なる可能性がある．このような不整合の回避については後述する．

予備サイトで，サービスを継続するフェイルオーバーは，通常システム管理者の判断により開始する．まず，これまでの主サイトのサービスをネットワークから切り離されていることを確認し，予備サイトで，サービスしていたURL，IPアドレスを引き継ぎ，アプリケーション・サーバを起動する．さらに，アプリケーション・サーバを予備データベースに接続し，サービスを開始する．

フェイルオーバー後に，主サイトが復旧した場合，もしくは新たな予備サイトを設定する場合，まずはデータベースを同期させる．サービスを一時停止し，スナップショットを取得する．もしサービスを長時間停止させたくなければ，アク

表 8 主データベース，予備データベース，クライアントの処理状態

	主データベース	予備データベース	クライアント
ケース 1	処理済	処理済	処理済
ケース 2	処理済	未処理	処理済
ケース 3	処理済	未処理	未処理
ケース 4	処理済	処理済	未処理
ケース 5	未処理	処理済	未処理
ケース 6	未処理	未処理	未処理

セスログを保存しながら，サービスを再開する．スナップショットを新サイトに転送し，スナップショットを復元する．そして，スナップショット取得後のアクセスログを使って，データベースを同期させる．もし，サービス中であれば，データベースの同期完了後に，いったんサービスを停止する．最後に，適切なサイトでアプリケーション・サーバを起動し，複数のデータベース・サーバに接続し，サービスを再開する．

4.4.2 不整合回避

アプリケーションからのデータベース処理要求がデータベース接続ライブラリを介して，各データベースで処理されている間に災害が発生した場合，主データベース，予備データベース，クライアントは，表 8 に示すような処理状態と成り得る．

ここで，ケース２，ケース４，ケース５の場合，予備データベースとクライアントとの処理状態に不整合が発生しているため，フェイルオーバー後に問題となる可能性がある．

ケース４，５の場合，不整合を解消するには，本来であれば，予備データベースにおいてロールバック作業を行い，当該データベース処理要求を未処理の状態に戻す必要がある．しかしながら，一般的なデータベースアプリケーションでは，commit など後戻りできない処理に関して，その応答が返らなくとも，処理が正しくできていれば論理的に問題とはならない．クライアント側で，応答がないか

らといって，複数クリック（処理要求）するようなことがあっても，アプリケーション側で同一処理を行わないような実装を行うためである．通常のアプリケーションにおいては，ロールバック処理は不要である．

ケース２は，主データベースとクライアントでは処理済となり，予備データベースでは未処理という不整合である．顧客側が予約したにも関わらず，事業者側がその予約を紛失するようなケースである．

基本的には，予備データベースにおいて未処理となっているデータベース処理要求を処理すればよいが，主サイトが消失した場合，どのデータベース処理要求がアプリケーションで処理済か未処理かを判別することは困難である．ただし，予備サイトに保存されたアクセスログと予備データベースのアクセスログから，予備データベースにおいて，どのデータベース処理要求以後のデータベース処理要求が未処理かを抽出することは可能である．

よって，新しいアプリケーションプログラムを予備サイトで起動し，新しいセッションを開始し，未処理のデータベース処理要求をすべて処理すればよい．アプリケーションが未処理かもしれないデータベース処理要求まで予備データベースで処理する可能性はあるが，これはケース４，５の状態に相当し，前述したようにこの状態は問題とはならない．

このようにアクセスログを使って，予備データベースの処理を進める場合に，問題は３つある．１つは，予備サイト側のアクセスログが通信遅延のために，最新状態とはなっていない場合があることである．もし，予備サイト側に保存されていないデータベース処理要求の中に，commit 等の確定処理が含まれていなければ，アプリケーションにおいても，当該トランザクションは破棄されるだけなので，問題とならない．もし確定処理が含まれている場合，そのトランザクションは失われ，復旧が困難となる．これに対し，本システムでは，確定処理のデータベース処理要求に対しては，同期処理を指定することとする．これにより，常に予備サイト，クライアントの順で確定されるので，上記のような問題は生じない．

二番目の問題は，アクセス応答が異なる，つまりアクセスログ中のアクセス応答内容が，実際の予備データベースのアクセス応答内容が異なる場合があることである．両サイトが同期しているならば，当該トランザクションの破棄，ロール

バックを行い，データベース処理要求前の状態に戻し，アプリケーションにエラーを返せばよい．しかし，両サイトが非同期であれば，アプリケーションに確定処理を返した後に予備データベースが異なる応答となる可能性がある．そうなると別途手動で処理するか，予備データベースを再構築することで修正するなどの復旧作業を行うが，いずれも手間のかかる作業である．よって，復旧作業が困難なデータベース処理要求に対しては，同期処理を指定する．

三番目の問題は，データベース処理要求が異なる，つまりアクセスログ中のデータベース処理要求内容が，実際の予備データベースのデータベース処理要求内容と異なる場合があることである．例えば，データベースの内容が異なるような状況で，データベース内のデータを参照した結果を使ってデータベース処理要求を行うような場合である．このような事態を回避するためには，少なくともデータベースが別途更新されることがないよう処理の順序を管理する必要がある．よって，データベースの状態が更新される可能性があるクエリ実行や更新型のデータベース処理要求は少なくとも順序確定処理を指定する．

4.5 遠隔複製評価実験

4.5.1 遠隔複製機構の実装

提案方式を Java のデータベース接続ライブラリを拡張し，前章で述べた JDBC ゲートウェイを用いて，ディザスタリカバリ向け耐障害システムに適用して性能評価実験を行った．実験では，アクセス状況として，SQL 処理を指定できるようにした．JDBC ゲートウェイは，データベース処理要求とアクセス状況（SQL）とを照合し，四つの複製方式，通常処理，順序確定処理，同期処理，スキップ処理に分類して，処理を進める．具体的には，executeQuery と executeUpdate を順序確定処理とし，commit を同期処理とし，その他の処理を通常アクセス処理とした．

4.5.2 遠隔複製実験システム

図36に実験システムの構成を示す．サーバOSはLinux(FedraCore5)，データベースにはPostgreSQL 8.0，及びTPC-C[41]に基づくIBM社OLTPベンチマークツールキット[15]を用いた．インターネットでの利用を想定し，遅延およびパケットロスをエミュレートするためにNistNET[6]を用いた．実験では，NistNETのパラメータを変動させ，さまざまなネットワーク環境下での性能検証を行い，以下の処理構成の性能比較を行った．

1. データ複製なしの構成,
2. 従来のデータベース内部でのデータ複製を想定し，PGCluster[5] [39]によるデータ複製を行う構成 ,
3. 従来のデータベース外部でのデータ複製を想定し，pgpool[58]によるデータ複製を行う構成,
4. 提案方式JDBCゲートウェイによるデータ複製を行う構成.

Pgpoolのシステム構成では，プロキシサーバをアプリケーション・サーバ内で動作させ，ベンチマークプログラムはローカルなソケット通信を介して接続した．

[5] PostgreSQL 7.4.7ベース

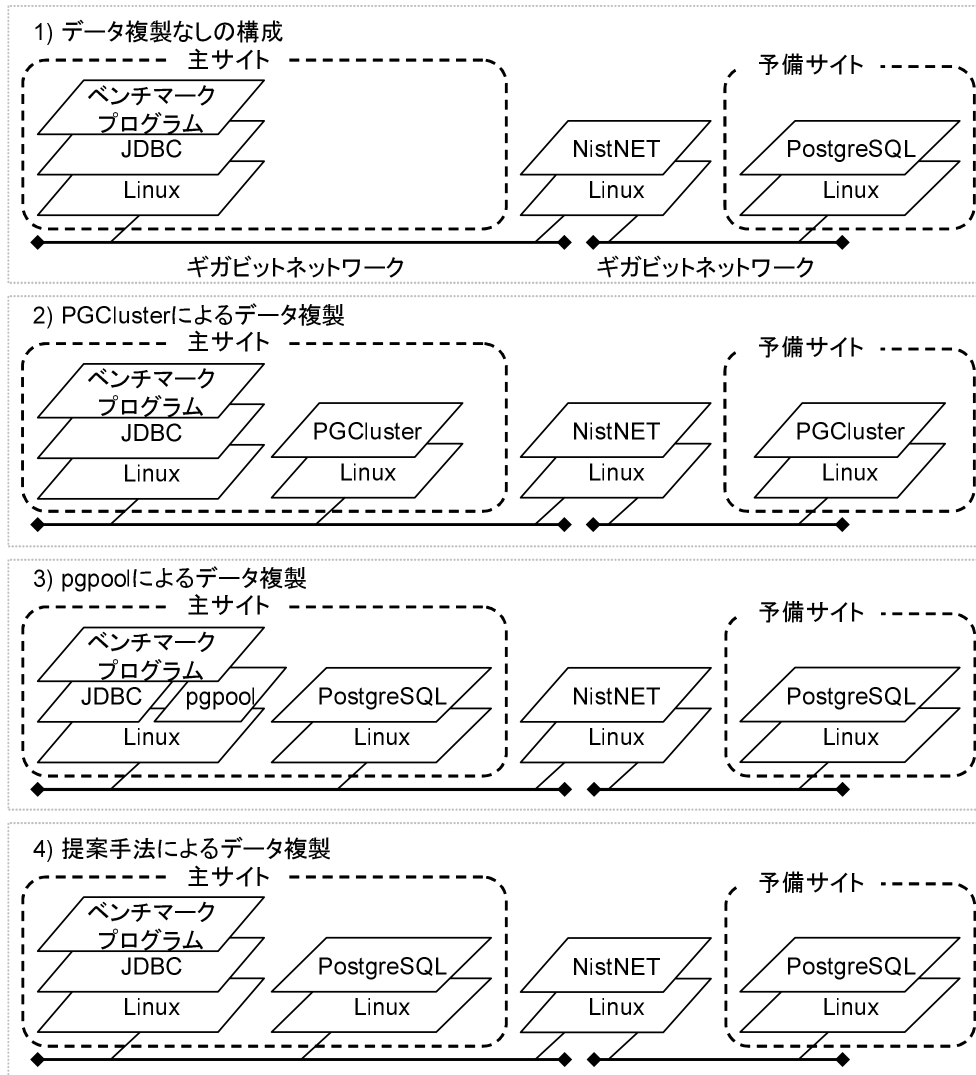


図 36 遠隔複製実験システム

表 9　遠隔複製実験結果

複製方式	遅延(ms)	パケットロス(%)	Tx/秒	性能劣化率	複製方式	遅延(ms)	パケットロス(%)	Tx/秒	性能劣化率
データ複製なし	0	0	11.42	0.0%	pgpool	0	0	5.07	55.6%
		0.1	10.51	8.0%			0.1	4.29	62.4%
		1	4.78	58.1%			1	1.67	85.4%
	1	0	8.66	24.2%		1	0	4.01	64.9%
		0.1	8.08	29.2%			0.1	3.68	67.8%
		1	4.31	62.3%			1	1.3	88.6%
	10	0	2.85	75.0%		10	0	1.33	88.4%
		0.1	2.76	75.8%			0.1	1.35	88.2%
		1	2.16	81.1%			1	0.86	92.5%
PGCluster	0	0	4.1	64.1%	提案方式	0	0	7.7	32.6%
		0.1	3.64	68.1%			0.1	6.9	39.6%
		1	2.5	78.1%			1	5	56.2%
	1	0	3.62	68.3%		1	0	6.9	39.6%
		0.1	3.48	69.5%			0.1	6.6	42.2%
		1	2.04	82.1%			1	4.5	60.6%
	10	0	1.8	84.2%		10	0	6.5	43.1%
		0.1	1.72	84.9%			0.1	6.2	45.7%
		1	1.45	87.3%			1	5.3	53.6%

4.5.3 実験結果

実験結果を表 9，図 37 に示す．実験では，NistNET を用いて，RTT のネットワーク遅延を 0 ミリ秒，1 ミリ秒，10 ミリ秒に設定し，またパケットロス率を 0 %，0.1 %，及び 1 %に設定した．グラフの縦軸は，データベースのスループットを毎秒のトランザクション数 (tps) で示す．データ複製なしのローカルなデータベースアクセスでは，11.4tps である．データ複製なしの遠隔データベースアクセスでは，ネットワーク遅延とパケットロスの大きい影響を受け，処理性能が低下している．トランザクションの中で，多くのファイルシステムアクセスがあり，それらがボトルネックとなっていると思われる．

PGCluster の場合，データ複製時の性能は，データ複製なしのローカルなデータベースアクセスの場合と比較して，64.1 %から 87.3 %下落した．pgpool の場合は，55.6 %から 92.5 %下落した．これに対して，提案方式の JDBC ゲートウェイの場合，性能の劣化は 32.6 %から 60.6 %にとどまっており，特に，ネットワーク遅延・パケットロスに対する許容性が高いことが分かる．データベースアクセ

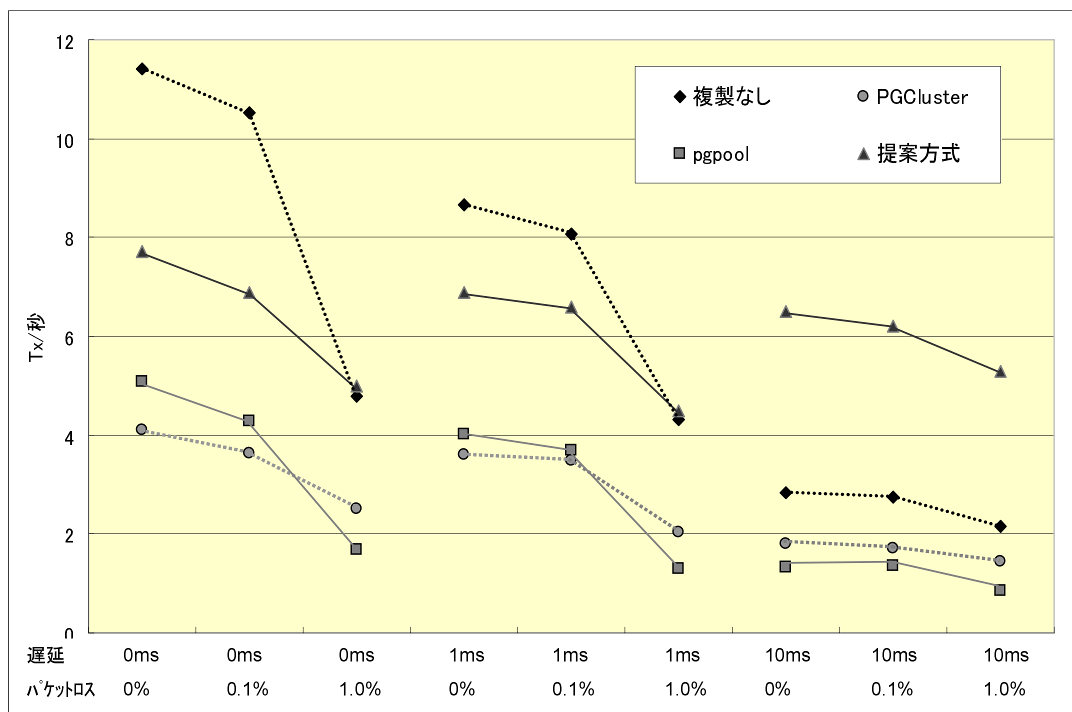


図 37　遠隔複製実験結果

スに着目すると，提案方式では，ローカルなデータベースアクセスのみ，非同期遠隔アクセス，同期遠隔アクセスがあるが，ネットワーク遅延・パケットロスが大きくなると，同期遠隔アクセスの影響のみが目立つ．

また，信頼性（可用性）については，前章と同じ計算式 T/(T + Rs*Td*2 + Rd*Td*2 + Ra*Ua*(Ta*Trb)) を利用して計算する．性能向上により，Ua（アプリケーションの処理中率）は明らかに低下し，また Ta（アプリケーションのタイムアウト時間）を低減できる可能性があるロールバック不要，各タイムアウト時間１秒，各障害の発生率は１%，処理量を毎秒１トランザクションと仮定して，可用性を算出する．処理中率を遅延 10 ミリ秒，パケットロス 0 の実験結果から求めると，従来方式では，1.33 トランザクション/秒であるので処理中率は 0.752，提案方式では，6.5 トランザクション/秒であるので，処理中率は，0.154 となる．上式に各値を代入して，可用性を求めると，従来方式では 99.95250 %，提案方式

では 99.95848 %となる．その差は，0.00598 %である．可用性に関しては，負荷分散構成での実験と同様，ほぼ従来方式と同等であることが確認できた．

4.6 ディザスタリカバリ機構への適用のまとめ

本章では，ディザスタリカバリ機構への適用を想定した柔軟な障害復旧機構を議論した．データベース接続層における汎用的なデータベース接続ライブラリを拡張し，あらかじめ登録されたデータベースアクセス状況と，実状況とを照合し，同期複製，非同期複製方式を制御することで，性能と信頼性とのバランスを実現する．アクセス状況の登録を変更することで，状況と処理：通常アクセス（非同期処理），同期処理，順序確定処理，スキップ処理との対応付けを変更し，IT サービスの災害時復旧要件 RTO，RPO に応じての設定を行うことができる．

実験では，TPC-C ベンチマーク（一般的な卸売り会社のトランザクション処理）を用いて，従来手法と性能比較を行った．データベースの同期更新頻度を調整することで，提案手法はネットワーク遅延とパケットロスの両方に従来手法と比べ許容性があることが確認できた．東京〜大阪間でのディザスタリカバリ目的で同期複製を行おうとすれば，およそ遅延は 10 ミリ秒であり，その環境では，既存手法の 3.6 倍 (提案手法:6.5tps，PGCluster:1.8tps) 優れている．中小規模でのコマースサイトの更新性能要件は5〜10tps といわれており，本実験では限定的ながら，提案方式が実用可能レベルであることを確認できた．

5. SVMを用いたデータベース処理要求の複製処理制御

5.1 データベース処理要求の複製処理制御

前章においては，災害発生時の業務サービスを継続することを目的に，主サイトと予備サイトのデータベースを同期させる方式を提案した．信頼性と性能は，トレードオフの関係にあり，サービスの信頼性の要件を確保しつつ，性能を最大化するような柔軟な制御が重要である．ここでの柔軟な制御とは，システムの状態およびサービス要件に応じ，サイト間のデータ複製方式の同期，非同期を切り替えることである．データベースアクセスの種類に応じたルールを設定し，そのルールに応じた同期・非同期の切り替えを行い，信頼性と性能の制御を実現する．しかし，最適な制御を行うためには，データベースアクセスの種類だけでなく，多様なシステムの状態を考慮する必要があり，人手によるルールの記述は困難である．

このような運用管理におけるルール記述の困難さに起因する問題は，データベース・システム特有のものではない．現在，システム運用管理においては，ルールに従って，システムの状態を監視，その監視結果にもとづいて何らかの管理操作を実行するという自律運用管理が，一般的になりつつある．これは，ITシステムが，複雑化，巨大化し，人手による管理は限界にきているためである．このようにルール化することで，システムを熟知していないシステム管理者でも，一定の品質の運用管理が期待でき，低コストに，安定したシステムの運用管理が実現できる．しかし，その運用管理のルールの記述には，先に述べたデータベース・システムの運用管理の場合と同様の問題を抱えている．

運用管理のルールは，システムベンダが，システムの特性を理解し，検証を繰り返し，これまでの経験やノウハウをもって作成するものであり，簡単に記述できるものではない．基本的なルールは，あるイベントが発生したら，そのイベントに関連付けられた対処コマンドを実行するというものであるが，複雑なシステムでは，単一のイベントではなく，複数のイベントを考慮する必要があり，また一定時間内の発生頻度といった時系列をも考慮する必要がある．

このような運用管理の自動化の問題に対して，本章では，これまで開発をして

きたデータベース複製機構の同期制御を対象に議論を進める．具体的には，イベントの時系列パターンを学習し，そのパターンに応じて，コマンドを実行する機構について述べる．

以下，5.2 節で運用管理の自動化に向けての課題を明確化し，5.3 節でその課題解決のための方式提案を行う．5.4 節で方式検証のための実験および結果を議論する．最後に 5.5 節で本章のまとめを述べる．

5.2 運用管理の自動化に向けての課題

5.2.1 ルールベース運用管理

システムの運用管理の基本は，管理者がイベントを監視し，問題を発見し，その対処方法を調査し，対処を実行することである．正しい対処をするために，問題の原因を分析する場合もあるが，必須ではない．つまり，イベントと対処の対応付けが完全にできれば，運用管理を自動化することができる．しかしながら，システムが大規模複雑化し，多種大量のイベント発生のために，システムの状態把握は困難となり，またシステムを構成する多様なミドルウェアには多くの設定パラメータがあり，ミドルウェアを跨ってパラメータを整合させて更新する必要がある場合も多いため，対処作業の導出は困難である．

いくつかの運用管理製品においては，障害対処の事例をナレッジとして蓄積し，ある事象に対して，その原因および対処方法を検索することで，問題解決を支援するエキスパートシステム機能を備える．しかしながら，上述したようなシステムの状態把握を支援しておらず，またすべての障害をカバーするように事例を設定することはほぼ不可能であることから，十分機能しているとはいえない．

一方，最近の製品においては，自律運用管理と呼ばれる技術が導入されつつある [46][52][63]．自律運用管理においては，問題解決の導出だけでなく，監視・判断・対処という一連の処理を繰り返し，まったく人手を介さずに，運用管理を実現することが目標である.「監視」においては，大量に発生しているイベント（ログ）を収集し,「判断」においては，監視データの中から，鍵となるパターンを迅速に発見し，そのパターンに対するいくつかの対処策の中から，適切な対処方法

を導出し,「対処」において，前段で導出した対処方法を実行，制御する．ここで，イベントのパターンと対処策のセットは，ルール，あるいはポリシーと呼ばれ，IF-THEN 型で記述されることが多い．IF 項には，発生したイベント，システムのパターンを記述し，THEN 項に対処実行するコマンドを記述する．あるイベントが発生したら，各ルールの IF 項を照合し，もし合致するルールがあれば，THEN 項に記述されているコマンドを自動的に実行する．大規模複雑なシステムにおいては，IF 項は多様化し，THEN 項は複雑となり，正しいルールの設定は非常に困難である．

5.2.2 運用管理における学習技術の利用

前述したようにルールベースの運用管理では，システムが大規模複雑になると，ルールの記述が困難となる．そのため，システムの目標状態のみを設定するようなルールの簡単化やルールを自動的に設定する試みがなされている．

Bigus らは，ABLE と呼ばれる自律エージェントのプラットフォームを開発し，そのプラットフォームにおいて，対象システムのモニタリング，ゴール状態との比較，および調整を繰り返すことで，Web サーバのパラメータを調整する自律運用管理機能の実現方式を示した [5]．

このような性能チューニングのために学習が適用される試みは，パラメータ数の多少の違いはあるが，従来からある [51][53]．しかしながら，パラメータの調整などの簡単な制御は可能だが，まったく新たなコマンドを自動的に設定することや，システムの制御系が未知である場合は，この方式の適用は困難である．すべてのコマンドを試行錯誤して，正しく動作するコマンド探索することも考えられるが，復旧困難な障害を引き起こす危険がある．また，調整した結果が，すぐに効果を表すとは限らず，後になって，別の箇所で不具合を引き起こす可能性もある．システムの特性・振る舞いを十分理解していなければ，自律機能の実現は困難である．

また，ABLE においては，システム管理者は，モニタリング結果とゴール状態の比較結果から，対処方法を導出するポリシーを設定する必要がある．言い換えると，従来型のルールベースの運用管理におけるログと対処コマンドという低レ

ベルのルール記述と比べれば，抽象度が高く，簡単化されてはいるが，イベント，対処方法の対応を記述したルールが必要であり，そのルールを設定することは，やはり困難な作業である．

原田らは，システムにおけるファイルアクセスの権限を規定するポリシーを適切に設定することは困難であることから，プロセス生成の仕組みを利用し，まずプロセスの実行状況に応じたデータベース処理要求履歴を抽出し，それを基に不要なアクセス許可を含まないポリシーを半自動的に生成する手法を示している[48]．

しかしながら，管理者は，どのようなイベント（ここではファイルアクセス）が生じるかをあらかじめ分かっている必要があり，またそのファイルアクセスに対するアクションとしては，許可か不許可かのいずれかしかない．そのため，何らかの対応する処理（コマンド）を設定することには対応していない．コマンドを設定できたとしても，どのコマンドを設定すればよいかという支援は考慮されていない．

メールのアンチスパム機能においては，多種多様なスパムに対応するために，ルールベースでは限界があることから，ベイズ推論等の学習・推論技術が利用されている．スパムメールの特徴が，接続先，字句，プロトコル等に表れており，接続解析（送信先，送信元），字句解析（URL，メッセージ内容），プロトコル解析（ヘッダ，ドメイン）等の分析を行うことで，スパムのイベントパターンを発見し，スパムであるか否かを推定する．

また，山西らは，マイニング技術をベースとした外れ値検出，変化点検出，異常行動検出といった機能をログに適用し，正常時のログの時系列との差異から，未知のウィルスや攻撃の検出を行ってきた[50]．しかし，正常時の変化の範囲を，マイニングエンジンのパラメータで調整する必要があり，あらかじめシステムの特性を分析する必要がある．

以上，運用管理領域においては，特定機能の実現のために自律エージェント，ベイズ推論，マイニングなどの学習・推論技術が利用されてきている．学習技術を適用するには，学習に適したモデル化が必要であり．どのようなシステム，機能にも学習技術が適用できるわけではない．前節で述べたように，ルールベース

の自律運用管理は，イベントパターンをいかに発見し，そして，いかに適切な対処コマンドを対応付けるかがポイントである．これらの処理は，パターン認識，相関分析と考えられ，学習技術を適用できる可能性がある．本章の以下の節においては，本研究が対象としている，データベース複製機構における運用管理の自動化を実現する手法について議論する．具体的には，あるイベントパターンに対して，管理者が行ってきた，もしくは行うべきであった処理（コマンド）を学習し，イベントと対処とを対応付けるものである．

5.3 学習型運用管理方式のデータベース複製機構への適用

5.3.1 学習型運用管理

本節では，データベース複製機構の同期制御に機械学習を用いて自動化する手法について述べる．基本的なコンセプトは，図 38 に示すように，イベント監視を行い，イベントログおよび操作コマンドのデータを収集し，あるコマンドに関して，そのコマンドの最近に発生したイベント列に対して正解，その他は不正解として，機械学習を行い，学習モデルを作成する．そして，通常の運用管理では，その学習モデルを用い，イベント履歴を入力として，各操作コマンドに対して，正解，不正解を推定する．正解と推定された操作コマンドを管理者に提示し，管理支援を行う．

5.3.2 複製処理のタイミング制御

サービスを運用する場合，サービス事業者と顧客の間では，SLA（Service Level Agreement）を結び，Service Level Objective（SLO）を設定する [44]．SLO は，応答時間，稼働率等のシステムが満たさなければならない最小限の要件である．転送タイミング制御に関する最小限の要件とは，指定された時間以内に転送することである．本稿では，最大転送間隔が SLO として指定されるものとし，最大転送間隔で指定された時間内に少なくとも一度は予備サイトへの転送は行われる

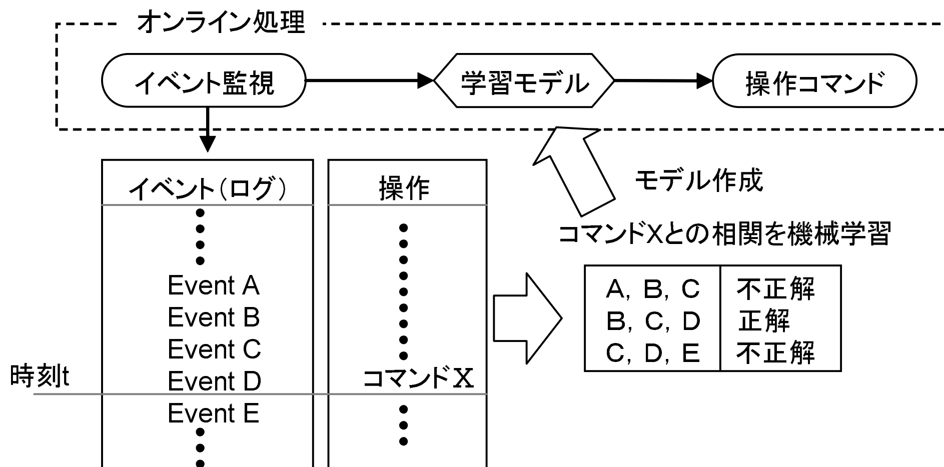


図 38 学習型運用管理方式

ものとする[6].

上記の条件を考慮した一般的な転送タイミング制御の方式を図 39 に示す．データベース処理要求が発生してから，指定された時間 t だけ待って，その間に発生したデータベース処理要求をまとめて転送する．この際，時間 t は SLO で指定された最大転送間隔である．

もし次のデータベース処理要求までの時間間隔を予測可能であれば，つまり，次のデータベース処理要求まで t 以上の時間間隔があれば，図 40 のように次のデータベース処理要求を待つことなく，そのデータベース処理要求の直後に転送する．これにより，予備サイト側で最新の複製を保持する時間を長くすることができ，信頼性を向上させることが可能となる．

ここで，本章で対象とする信頼性について説明する．前述したように，ICT サービスを提供する耐障害システムの信頼性指標としては，どのアクセス時点まで遡ってサービスを復旧するかという Recovery Point Objective（RPO）と，サービスの復旧までに要する時間 Recovery Time Objective（RTO）の 2 つである．

システムの信頼性を高めるということは，主サイトへのアクセスの都度，その

[6] RTO は複製トランザクションの非同期処理における未処理 DB アクセスの個数であり，SLO は未処理 DB アクセスの最大許容遅延であるということができる．

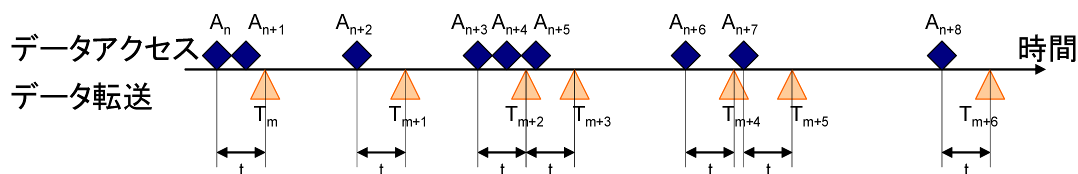


図 39 一般的なバッファデータ転送

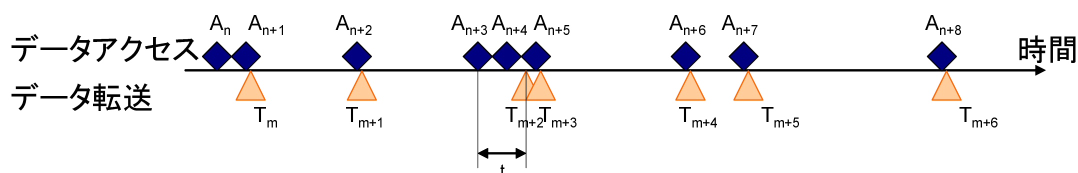


図 40 改善したバッファデータ転送

アクセスを予備サイトに転送し，常に最新状態に復旧可能な状態とし（RPO=0），主サイトでのアクセスの処理と同期して，予備サイトでもそのアクセスを処理することで，常に予備サイトも最新状態を保つ（RTO=0）ことである．しかしながら，前述したようにこのように予備サイトへの転送をアクセスの都度行うことは，システムの処理性能が低下することとなる．通常，システムの信頼性と処理性能とはトレードオフの関係にあるため，信頼性と処理性能とのバランスを考慮して，転送タイミングを制御することが必要である．

サービスによって，要求されるRPO，RTOはさまざまであるが，データを紛失してよいとされるサービスは少なく，何らかの手法により，RPO=0となる運用設定を行うのが一般的である．実際，本データベース複製機構では，アクセスログは別途保存され，もし予備サイトが最新状態でなければ，そのアクセスログを使って，未処理のアクセスをデータベースに投入し，予備サイトを最新状態に更新してからサービスを復旧する機能を備える．以下においては，変動要素であるRTOに絞って，信頼性を議論する．

RTOを計算するためには，予備サイトにおける未処理アクセスのデータベー

スへの投入処理に要する時間を見積もる必要がある．本論文では，各アクセスが同一の処理と仮定し，RTO は未処理アクセスの個数に比例するとする．以下，累積未処理アクセスの個数を RTO として議論を進める．

5.3.3 データベース処理要求間隔時間の推定

前節で述べたように次のデータベース処理要求までの時間間隔を予測できれば，予測に基づいた複製（データ転送）のスケジューリングを行い，RTO を削減することが可能である．データベース処理要求は，一連のトランザクションの中で発生し，その頻度には偏りがあるのが一般的である．例えば，Web を使ったシステムでは，ユーザの操作と操作に間には，いくらかの間隔がある．表 10 は，あるコンテンツ視聴のアプリケーションログの例であるが,「SEARCH」,「CONTENTS」といった各操作の間に間隔がある一方,「DIALOG」のようにある操作に連続して発生するデータベース処理要求があることが分かる．こういった頻度の偏りは，アプリケーションのプログラムによるもので，定性的であると考えられる．つまり，個別のデータベース処理要求の種類と，発生時刻のパターンから，次のデータベース処理要求までの時間間隔は予測可能と考えられる．

こうしたデータ転送スケジューリングの１つの手法として，データベース処理要求を実行するアプリケーションプログラムのコード解析をすることが考えられる．ソースコードが分かれば，どのタイミングでデータ転送すればよいかは自明である．実際，予備機構を組み込む場合，ソースコードに遠隔予備のコードを埋め込み，性能と信頼性を両立したシステムを構築することが多い．しかしながら，プログラムのソースコードを解析，改変することが困難である場合もあり，また多様なアプリケーションに対応するためには，多大なコストを要する．

そこで，機械学習を使って，次のデータベース処理要求までの時間間隔が空く時系列パターンを検出する手法を検討する．パターン学習は，これまでに多くの研究がなされており，代表的な手法としては，ニューラルネットワーク，SVM（サポートベクタマシン），k 近傍識別器，ベイズ分類などが知られる [54].

運用管理への適用においては，システム状態を監視，監視結果に応じた対処という一連の処理をモデル化するが，システムが大規模化するとシステム状態は多

表 10 アプリケーションログ（時刻とコマンドのみ抜粋）

時刻	コマンド
19:27:52	LOGIN
19:27:52	INIT
19:27:52	SEARCH
19:27:53	DIALOG
19:28:28	SEARCH
19:28:28	DIALOG
19:28:32	SEARCH
19:28:32	DIALOG
19:28:40	SEARCH
19:28:40	DIALOG
19:28:43	BACK
19:28:44	DIALOG
19:29:33	SEARCH
19:29:33	DIALOG
19:29:36	BACK
19:29:37	DIALOG
19:29:42	CONTENTS
19:29:42	RELOAD
19:29:44	SEARCH
19:29:45	DIALOG

様となり，学習の要素が多次元となる．また，あらゆる事例が多頻度で発生するとは限らず，少ない事例でも，ロバストに動作することが求められる．よって，学習方式としては，SVM が適していると考え，SVM を使った予測に基づきデータ転送のタイミング制御を行う．

学習データとしては，最近 n 個（今回は n=3 とした）のデータベース処理要求の種類とその発生時間間隔を入力とし，次のデータベース処理要求までの発生時間間隔が一定時間以内（つまり，アクセス転送を抑制すべき）であれば正解としたベクトルに変換して，学習モデルの生成を行う．図 41 は入力データのベクトル変換例である．

まず，今回のデータベース処理要求時に転送すべきであったかを次回のデータベース処理要求時間との間隔から得る．さらに，前回，前々回とのデータベース処理要求時間間隔とデータベース処理要求の種類を抽出する．

次に，ベクトル化では各項を正規化して記述する．行頭の値は正解（1）か不正解（0）かを示す．インデックス 0 は前回のアクセスとの時間間隔，インデックス 1 は前々回のアクセスとの時間間隔であり，最大転送間隔で正規化する．最大転送間隔より大きい値はすべて 1 となる．以後のインデックスは，今回，前回，前々回のデータベース処理要求において，k 種類のデータベース処理要求の何が発生したのかを 1，0 で示す．この例では 3 種類のデータベース処理要求とし，インデックス 2〜4 で処理要求 2 であること，インデックス 5〜7 で処理要求 3 であること，インデックス 8〜10 で処理要求 3 であることを示す．つまり，k 種類のデータベース処理要求であれば，インデックス n〜k+n-1 に今回発生したデータベース処理要求の種類，インデックス k+n〜2k+n-1 に前回発生したデータベース処理要求の種類，インデックス 2k+n〜3k+n-1 に前々回発生したデータベース処理要求の種類を記述する．このように，アクセスログをベクトル化し，SVM を適用する．

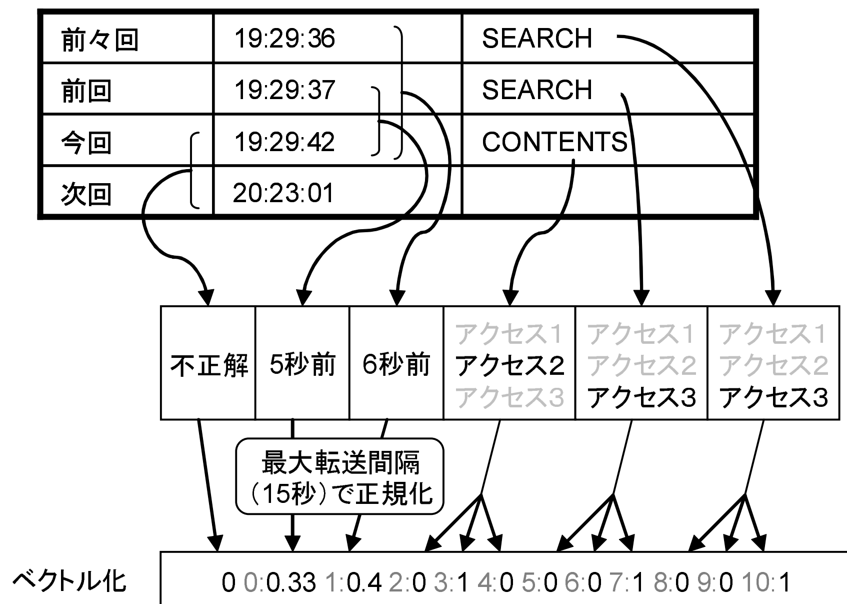


図 41 アクセスログのベクトル変換

5.4 方式検証実験および結果

5.4.1 予備実験

前節までに提案した方式の有効性を確認するため，まず TPC-C[41] の New-Order シナリオに沿って取得したアクセスログを使って学習し，次のデータベース処理要求までの発生時間間隔が，一定時間以上空くか，否かを予測した．なお，本実験で用いた SVM は，libsvm[8] を用い，デフォルトパラメータで実行した．つまり SVM の形式に C-SVC，カーネルの形式に RBF（radial basis function）を用いた．

TPC-C の OLTP ベンチマークツールキット [15] において，各トランザクション間に 1，2，3，4，5 秒の中からランダムの間隔が空くように設定し，また同一セッションのオーダー間に 18 秒の間隔が空くように設定した．各トランザクションは，あまり時間をおかずに利用者は入力し，また各オーダーはある程度時間間隔が空くと仮定する．TPC-C の制約では，各トランザクションに関し，New-Order は最低でも 18 秒の入力時間を要し，また 5 秒以下の応答時間を持たねばならないとされているためである．また，TPC-C では，10 端末が仕様であるが，アクセスログでは，各セッションを識別することが可能で，セッションごとに予測することから，本方式では端末数は関係なく，端末数 1 で実験を実施した．アクセスログは，Log4J を用いて，時刻と各データベース処理要求の種別を取得した．

5 時間実行したアクセスログを用いて学習し，別の 5 時間実行したアクセスログの各データベース処理要求に対して，次のデータベース処理要求の発生までの間隔が，SLO（最大転送間隔）に指定された時間以上かどうかを予測する．予測結果に基づいて，ただちに転送するか，次のデータベース処理要求まで待つかを判断する．従来手法は，前述したように，常に SLO に指定された時間だけ待って，その間に発生したデータベース処理要求をまとめて転送する方式である．手法の比較に関しては，遠隔複製の性能パフォーマンスに関する指標として転送回数，信頼性として RTO の累積値を比較する．

図 42，図 43，図 44 は，横軸に最大転送間隔をとり，それぞれ縦軸に予測精度，転送回数，RTO をとった場合の提案手法と従来手法との比較を示している．予測精度に関しては，ランダムなアクセス間隔となる 5 秒以下の範囲で精度は落ちる

が，それ以外では，ほぼ100 %の精度である．この精度の低下は，最大転送間隔が5秒以下の場合，同一オーダー内のトランザクション間隔が1〜5秒のランダム値を取り，その学習に失敗が見られるためである．

転送回数およびRTOに関しては，5秒以下の範囲で，多くのデータベース処理要求を転送すべきと推定し，転送回数の削減には寄与しておらず，RTOは0に近い値となっている．最大転送間隔=6以降では，オーダーの合間を正しく予測し，各オーダー処理の終了時に直ちに転送と推定するほど，従来手法よりRTOは減少する．最大転送間隔=18以降では，予測に関係なく，すべてのデータベース処理要求で待つようになるので，従来手法との差異はなくなる．

従来方式との比較のため，横軸に転送回数，縦軸にRTOをとったグラフと，提案手法と従来手法との転送回数比とRTO比との積（{提案転送回数×提案RTO}／{従来転送回数×従来RTO}）を図45に示す．RTOと転送回数がトレードオフの関係にあり，転送回数とRTOの関係は手法による差異が少ないことが分かる．しかしながら，転送回数比とRTO比との積を見ると，1以下の値となっており，転送回数の増加に比べRTOの低減の効果が大きいことが分かる．

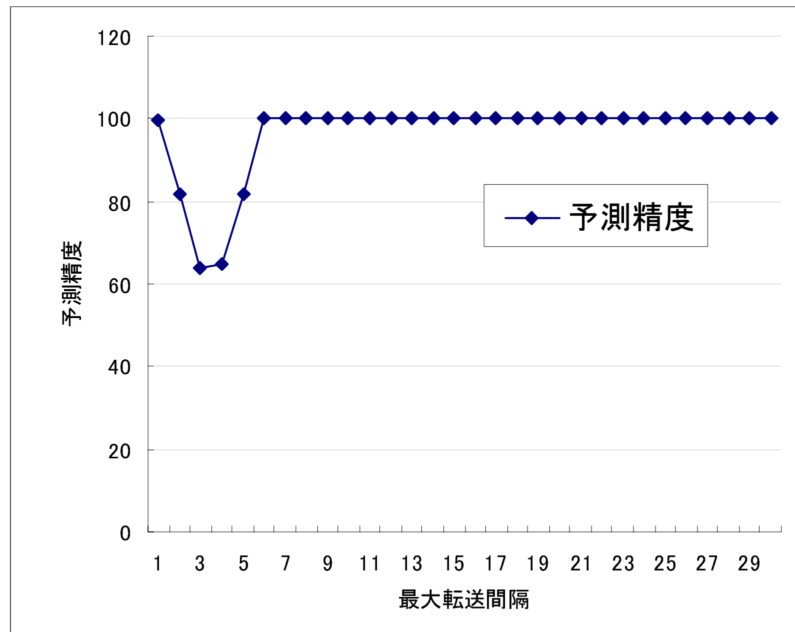


図 42 New-Order シナリオにおける予測精度の評価

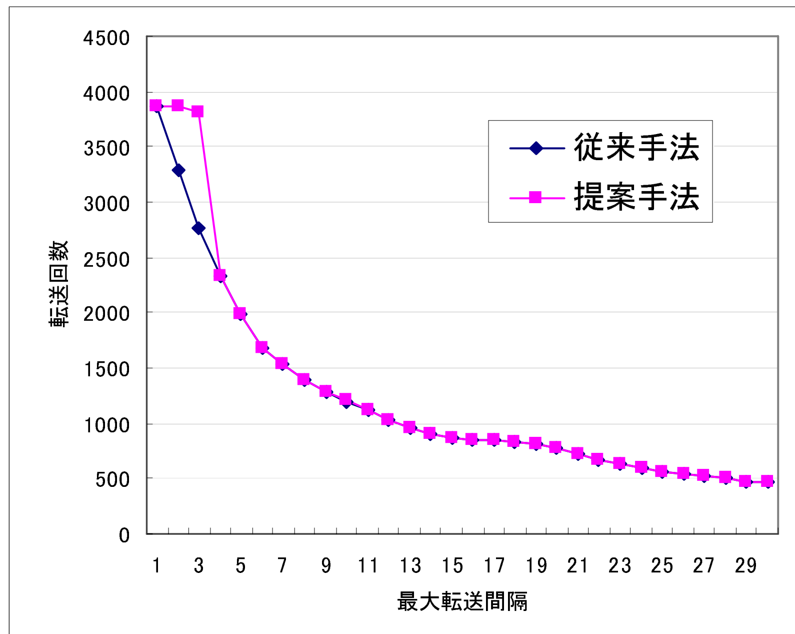


図 43 New-Order シナリオにおける転送回数の評価

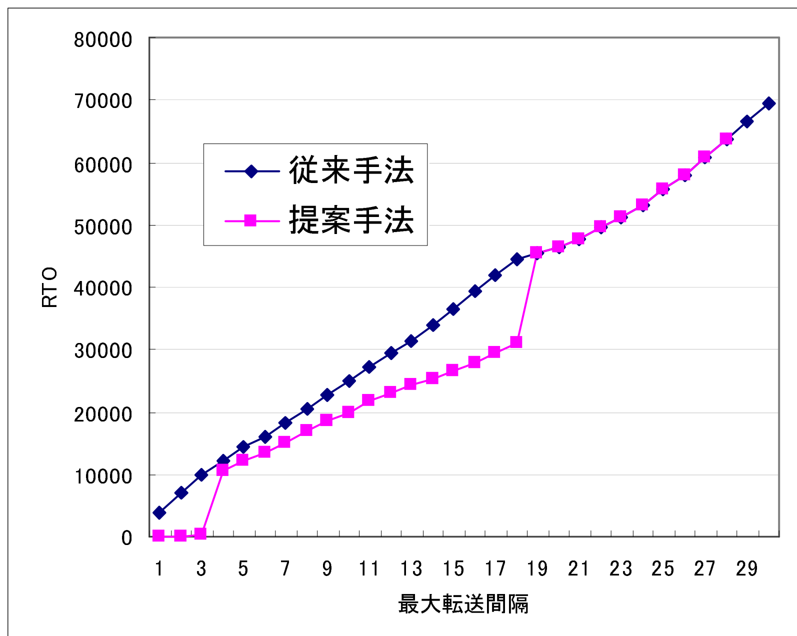


図 44 New-Order シナリオにおける RTO の評価

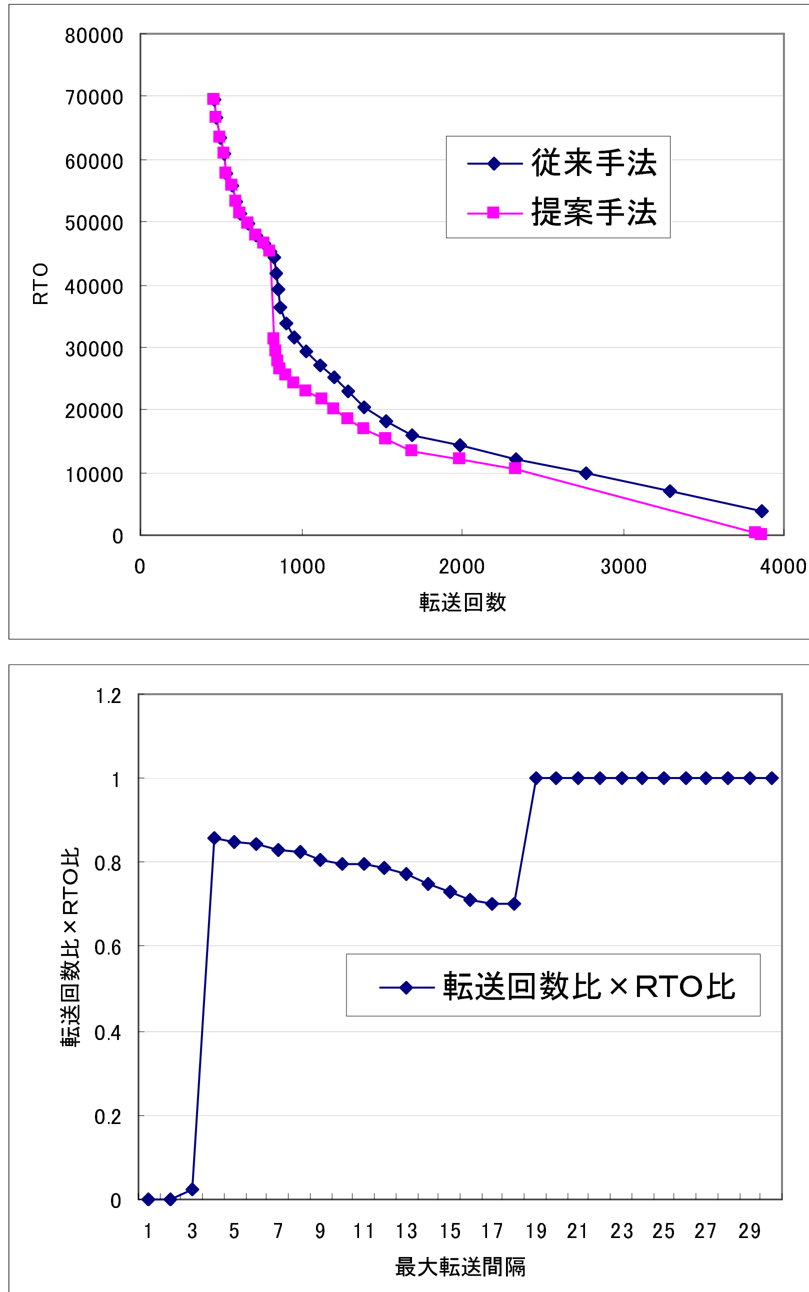


図 45 New-Order シナリオにおける RTO と転送回数との関係を示す図

5.4.2 アプリケーション実験

予備実験では，データベース処理要求間隔の特性が既知であったが，実際のシステムにおいては，データベース処理要求間隔にはバラつきがある．アクセスログに，表10に示したコンテンツ視聴システムの１ヶ月間の実際のログを使い，オフラインでデータベースへのデータベース処理要求を想定した方式検証を行った．ログは，合計9022（2.1Mバイト），一日あたり平均291（69Kバイト），最大931（219Kバイト），最小24（5.7Kバイト）である．10日間のログで学習し，別の21日の予測を行い，その平均を算出した．

予備実験と同じSLO（最大転送間隔）を変化させての実験結果を図46，図47，図48に示す．予備実験に比べ，予測精度は下がっているが，転送回数およびRTOは，従来手法から大きく離れてはいない．転送回数は，全体的に従来手法より増加し，RTOは低減している．予備実験と同様に，図49に示すように，RTOと転送回数との関係を見ると，やはり，RTOと転送回数がトレードオフの関係にあり，転送回数とRTOの関係は手法による差異は少ない．転送回数比とRTO比との積も予備実験と同様に，1以下の値となっており，転送回数の増加に比べRTOの低減の効果が大きい．

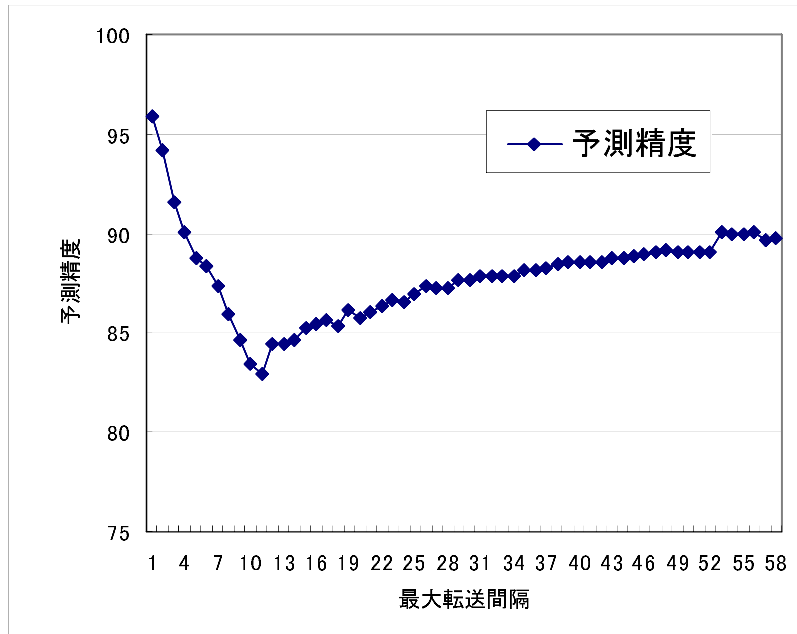


図 46 アプリケーション実験における SLO（最大転送間隔）に対する予測精度

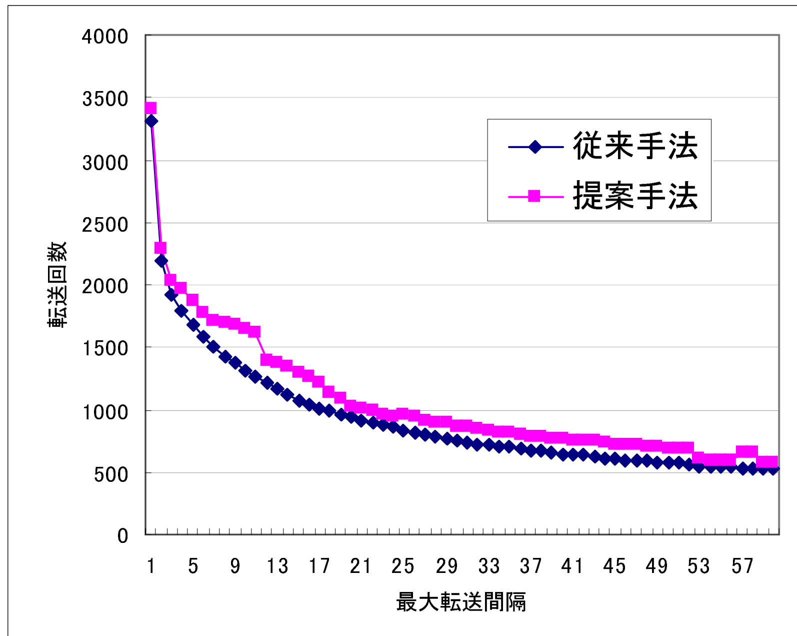


図 47 アプリケーション実験における SLO（最大転送間隔）に対する転送回数

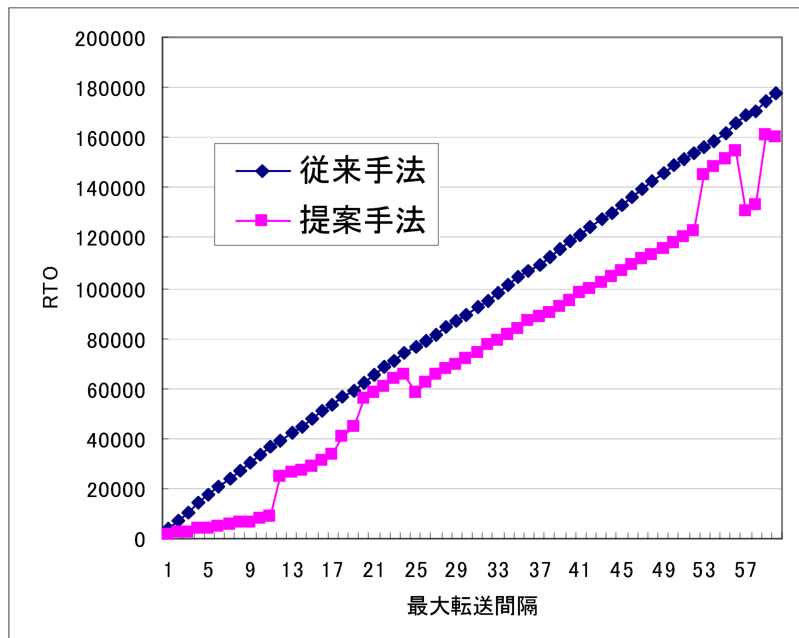
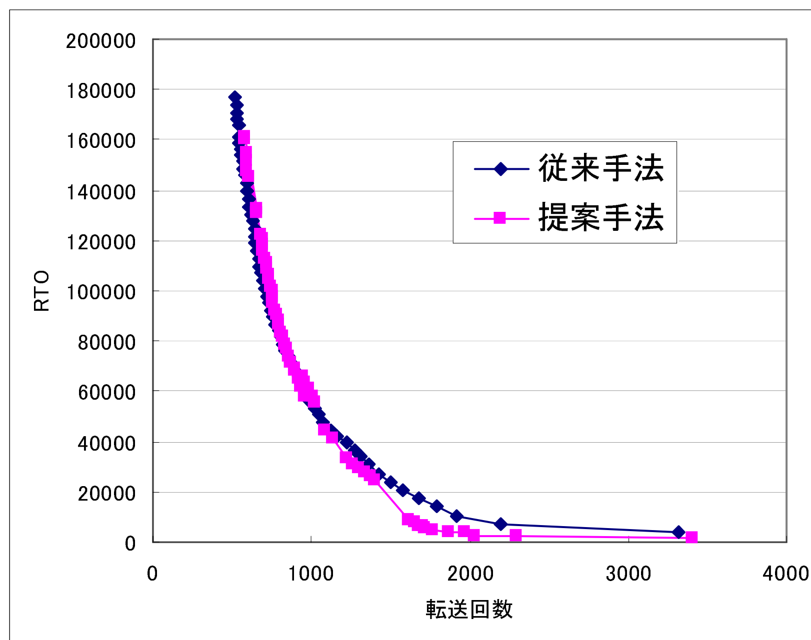


図 48 アプリケーション実験における SLO（最大転送間隔）に対する RTO



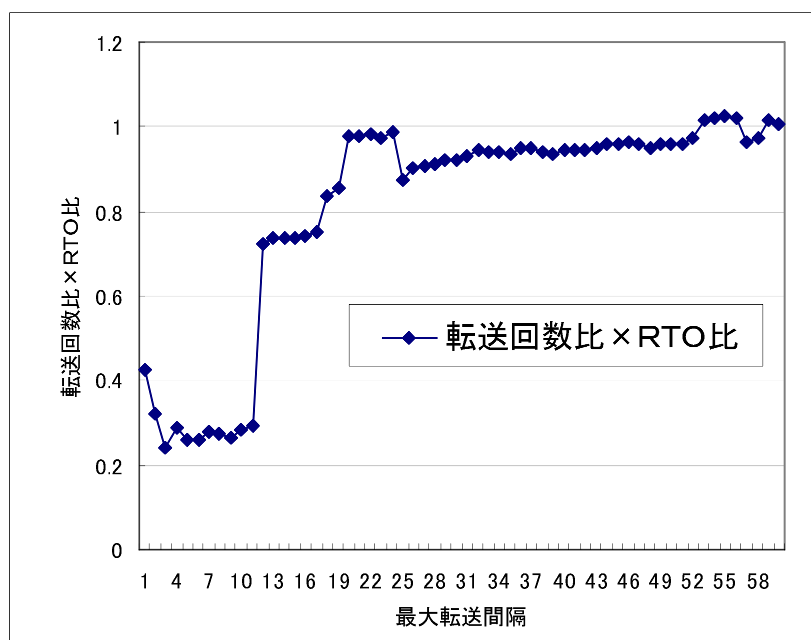


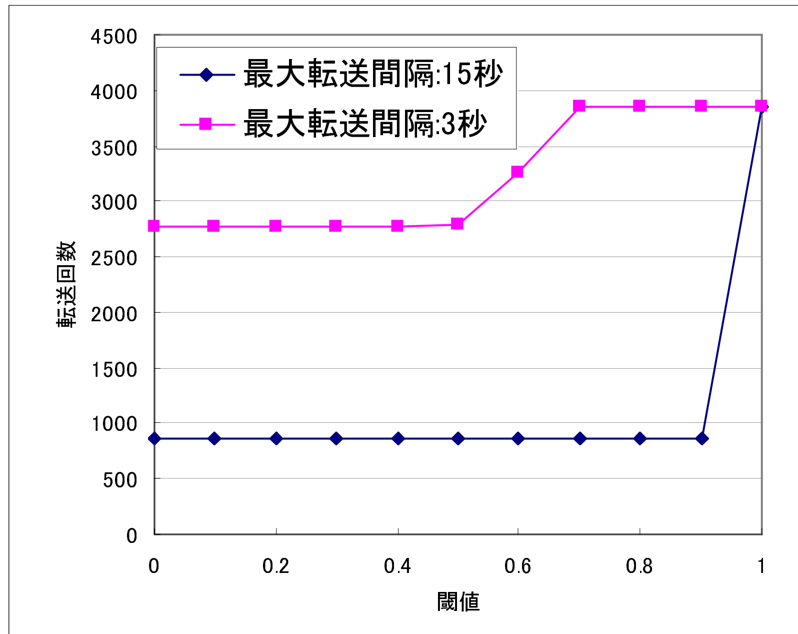
図 49 アプリケーション実験における RTO と転送回数との関係

5.4.3 予測閾値によるシステム制御

上述の実験結果から分かるように，RTO（信頼性）と転送回数（性能）はトレードオフの関係にある．このバランスを制御することで，多様なサービスの要件に対応することが求められている．本提案手法では，予測に基づいて転送の可否を判断するが，予測の確かしさの閾値を変更することで，転送回数およびRTOを制御できる可能性がある．

SVM の予測閾値は，0.5 を用いるのが標準であり，これまでに述べた実験結果もその値を用いた．そこで，この SVM 予測閾値を，0 から 1 まで変化させて予測を行い，転送回数および RTO がどのように変化するかを予備実験の New-Order シナリオと，アプリケーション実験のアクセスログを使って，評価した．その結果をそれぞれ図 50，図 51 に示す．

閾値を 0 とすれば，予測しないことと同じであり，従来方式と同じとなる．閾値を 1 とすれば，すべてのアクセスの都度転送することとなり，RTO はほぼゼロとなる．閾値を変化させることによって，RTO を制御することが可能だが，予備実験のように，アクセスの時間間隔に極端な偏りがある場合には，滑らかな制御は不可能である．



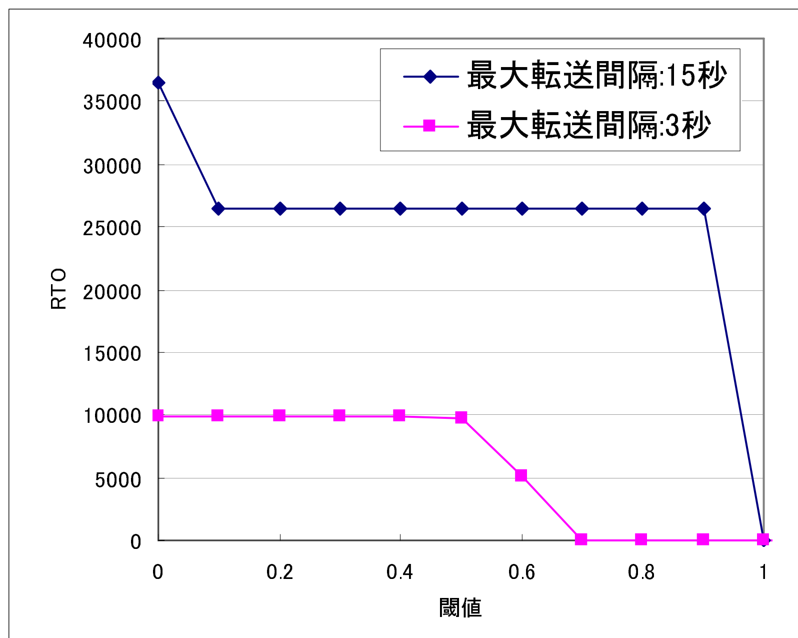
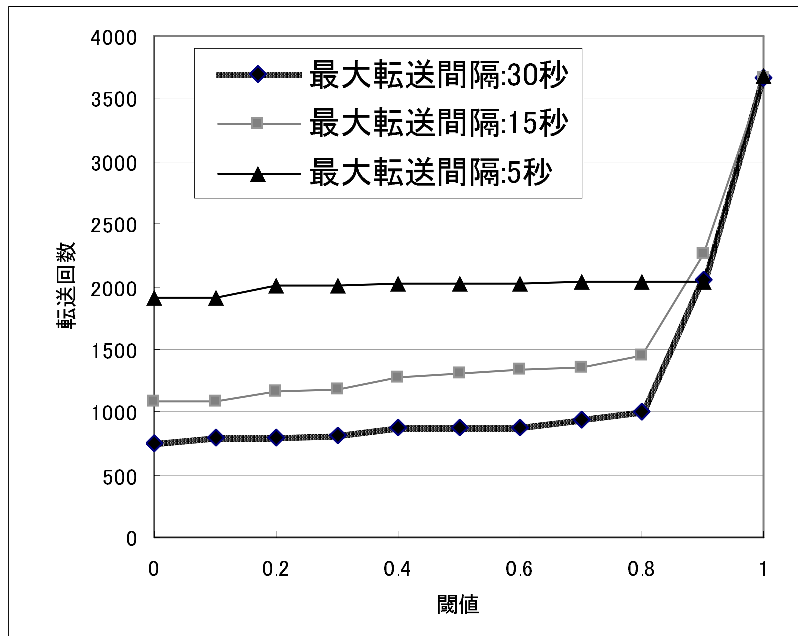


図 50 New-Order シナリオにおける予測閾値に対する転送回数，RTO の評価



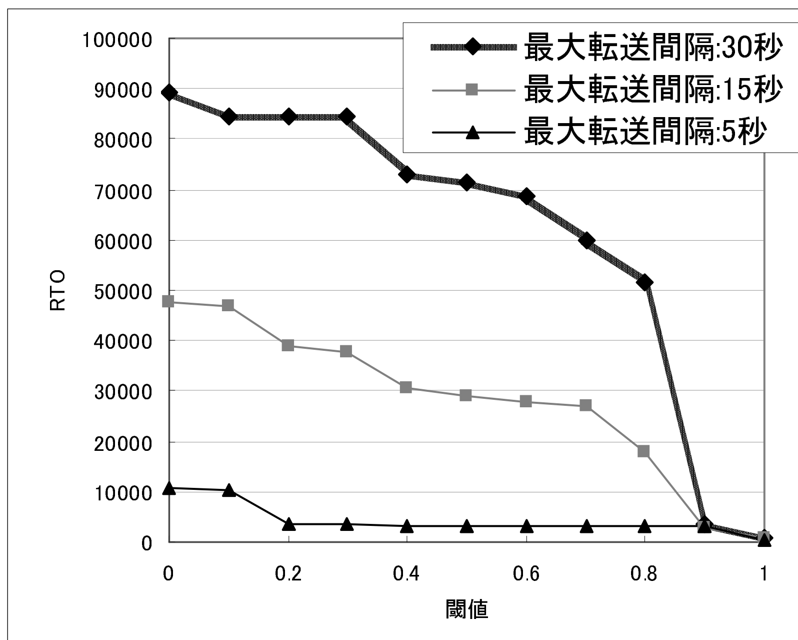


図 51 アプリケーション実験における予測閾値に対する転送回数，RTO の評価

表 11 学習量およびテスト量に対する予測精度，転送回数，RTO

	（1）			（2）			（3）		
	従来	提案	比率	従来	提案	比率	従来	提案	比率
転送回数平均	2280.97	2429.32	106.56	1575.48	1833.39	116.63	75.1	86.94	116.05
RTO平均	99662.45	85303.39	85.48	68818.48	45018.61	65.187	3284.68	2207.1	69.75
予測精度平均		80.78			84.32			82.79	

5.4.4 学習効果の評価

学習の効果検証のため，上述のアプリケーション実験と同じシステムのアクセスログを用い，下記の通りの異なる学習量で比較評価を行った.

1. 1 日のログで学習し，別の 30 日間の予測

2. 10 日間のログで学習し，別の 21 日の予測

3. 30 日間のログで学習し，別の 1 日の予測

本実験結果を表 11，図 52 に示す．グラフは，異なる学習量で，それぞれ 31 パターンの実験を行い，その平均，最小値，最大値を示している．なお，SLO（最大転送間隔）は 15 秒とした.

実験（1）から，学習数が少ないと，予測精度が不十分で，転送タイミングを制御できていないことが分かる．また，実験（3）から，学習数が多くとも，部分的（1 日のみの予測）にみれば，予測は外れることもあり，転送タイミングを制御できていない．しかしながら，実験（2）から分かるように，平均的には提案手法は有用である．転送回数の増加に比べ，RTO は低減，つまり信頼性の向上の効果があると見なせる.

5.5 SVM を用いたデータベースアクセスのタイミング制御のまとめ

本章では，データベース複製機構における性能と信頼性とを柔軟に制御するため，学習型の遠隔データ転送のタイミングを更新する方式について説明した．従

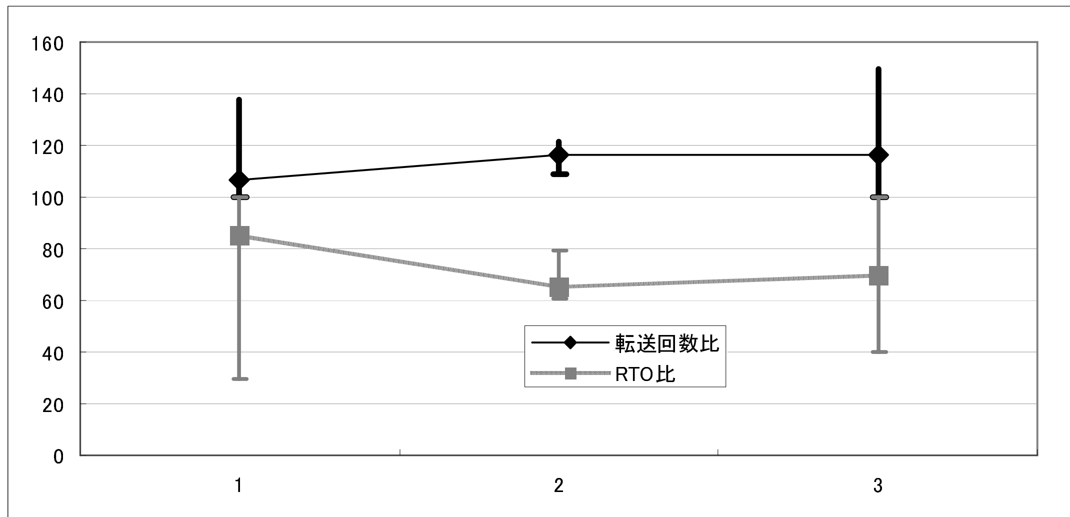


図 52 学習量およびテスト量に対する転送回数，RTO の評価

来のルール記述型の運用管理においては，システムの状態，イベントのログに応じて，その対処を指定する必要があったが，本提案方式では，ログからイベントの発生パターンと転送タイミングとの関係を学習し，各イベントに対して，データ転送を行うべきか否かを判断する．本提案方式によって，ルールを記述することなく，データ転送を制御でき，判断の閾値を変化させることで，システムの性能と信頼性とを柔軟に制御することが可能となる．

6. データベース置き換えのためのデータベース接続層の拡張

6.1 データベース置き換え支援

6.1.1 背景

近年，PostgreSQL，MySQL 等のオープンソースのデータベースの処理性能，信頼性および機能の向上は著しく，ビジネスの場面でも広く使われるようになってきた．そのため，高価な商用データベースを用いて構築したシステムを，サポート費用の削減等のために低価格のオープンソースデータベースに移行したい，また逆に，当初は低機能だが手軽なオープンソースデータベースを用いて構築したシステムから，業務拡大により，高度なデータベース機能，処理性能，信頼性が必要等の理由から，商用データベースに移行したいというニーズが高まりつつある．

このように，稼動しているシステムのデータベースを，異なる種類のデータベースに置き換えたいというニーズに対し，従来は，置き換え前の旧データベース用に開発されたアプリケーションプログラムを，置き換え後の新データベースに対応するように改造することで対処してきた．

しかし，データベースを置き換える際には，データベースだけでなく，システムの様々な構成要素も変更する必要があり，複雑なアプリケーションの一部を，他に影響なく安全に改造するのは容易ではなく，また，改造が不可能な場合もある．さらに，改造したアプリケーションを用いた置き換え後のシステムが，安定して動作するか検証することも困難である．

以上のような理由から，データベースの置き換えはリスクの高い作業であり，一度採用したデータベースを継続的に使用する傾向がある．

そこで本章では，アプリケーションを改造するのではなく，アプリケーションが発行したデータ処理要求を途中で変換することで，データベース置き換えを支援する方式について述べる．また，データベース置き換え後のシステムの動作を検証する方式についても議論する．

6.1.2 置き換え処理

データベースへの処理は，大きくは以下の３つの手順からなる．

1. データベース接続プロトコルでデータベースに接続する．
2. データベースに対しデータ処理要求（SQL）を発行する．
3. データ処理を行う．

そのため，データベースを置き換える際には，これら接続プロトコル，データ処理要求，データの３つを，旧データベース用から新データベース用に変換する必要がある．それぞれの変換方法について，以下で述べる．

データ 多くのデータベースが汎用的な CSV 形式にエクスポートおよびインポートする機能を備えている．また，一部のデータベースにおいては，特定のデータベースとのデータ変換を行うデータ移行ツールが提供されている．このため，データを新データベース用に変換することは比較的容易である．

接続プロトコル 近年，多くのアプリケーションは，直接データベースに接続するのではなく，JDBC[27]，ODBC[13]，ADO.NET[34] 等のデータベース接続層を介してデータベースに接続する．データベース接続層は，アプリケーションに標準的な接続インタフェースを提供し，アプリケーションが標準インタフェース対応の標準プロトコルで発行した SQL を，データベース固有のプロトコルに変換する．プロトコル変換機能はデータベースに応じてライブラリとして入れ替えられるようになっている．このため，プロトコルの変換も比較的容易である．

データ処理要求 データ処理要求の記述言語である SQL は，データベース間で共通の ISO SQL99[16] 等，標準仕様が規定されているため，理想的には変換を行う必要は無い．しかし，実際には，データベース固有の独自拡張された SQL が使われることが多いため，旧データベース用の SQL は，新データベースでは解釈できない場合がある．従来は，SQL を変換するのではなく，SQL を発行するアプリケーション自体を改造することで対応していた．

一般に，SQL は，アプリケーション内に散在してハードコーディングされたロジックにより生成されるため，ソースコードレベルで，それら生成ロジックを適切に書き換える必要がある．しかし，複雑なアプリケーションの一部を，他に影響なく安全に改造するのは容易ではない．ライセンスの問題，ソースコードが物理的に存在しない，多大な改造費用がかかる等の理由から，改造が不可能な場合も多い．

6.2 データベース置き換えの課題と解決方針

前述したように，データベース置き換えにおいては，データベース固有 SQL の変換が必要となる．本章では，変換処理の位置，変換処理の内容，変換の検証について議論する．

6.2.1 SQL 変換処理の位置

データベース処理要求のフローは，大きくアプリケーション，データベース接続層，ネットワーク，データベースとなる．

アプリケーションを改変することは，データベースアクセス部が独立しているようなプログラム構造となっていれば，その部分だけの改変で済むが，多くの場合，プログラムを書き直すのは，改造すべき部分の抽出，改造の影響の範囲の調査，テストなどたいへんな手間である．また，アプリケーションの改変は禁止，もしくは保証外となるなど制約がある場合も多く，アプリケーションの改変は望ましくない．

オープンソースのデータベースそのものを改変することも可能である．つまり，Oracle 用の SQL を処理可能な PostgreSQL を開発するようなことである．しかし，SQL 変換処理の負荷が増加するため，データベース・サーバがボトルネックの場合は，この位置での変換をするべきではない．また，オープンソースがバージョンアップされる都度，変換処理部への影響，正常動作の確認など手間がかかり，データベースの改変も望ましくないと考える．

ネットワーク上での変換は，ネットワークプロトコルから，データベース処理要求を組み立てなおすため，オーバーヘッドが大きくなる．

データベース接続層における変換は，アプリケーションの改変や，データベースそのものの改変と比べると，プログラムのロジックの変換は困難であるので，性能的には劣る可能性はあるが，ネットワーク上での変換よりは，アプリケーションが呼び出す関数ライブラリレベルで直接変換することができるので，効率的であると考える．

そこで，前章までに，データベース接続における多重化管理を説明してきたが，その複製処理の代わりに変換処理を行うことで，SQL 変換を実現することを検討する．データベース接続層の拡張は，1) アプリケーションの変更が不要，2) データベース接続ライブラリを置き換えるだけで，多くのアプリケーションに適用可能，3) データベース・サーバに負荷をかけない，といった特徴があり，この特徴はデータベースの置き換えにおいて，有利である．さらに，データベース接続層における SQL 変換と複製処理を組み合わせることで，複製先に異なるデータベースを用いたいというニーズにも応えられる．

6.2.2 SQL 変換ルール

新データベース用 SQL を発行するようにアプリケーションを改造するのではなく，アプリケーションが発行した旧データベース用の SQL を，途中経路であるデータベース接続層において，新データベース用 SQL に変換する．SQL は文字列であるため，変換処理そのものは，特定の SQL 文字列をパターンマッチで特定し，置換処理を行う．この置換パターンを変換ルールと呼ぶ．変換ルールには以下の４つの種類が考えられる．

予約語変換 旧データベース固有の予約語を，同等の新データベースの予約語に変換する処理．例えば，Oracle 用 SQL から PostgreSQL 用 SQL に変換する場合，Oracle における日時を返す関数である「SYSDATE」という予約語を，PostgreSQL における同様の機能をもつ関数の予約語である「CURRENT_DATE」に変換する．

構文変換 予約語だけでなく，パラメータを含む構文を，同等の構文に変換する処理．単純な置換だけでなく，パターンマッチで特定したパラメータを後方参照した上で，置換後のパターンに含んで置換する必要がある．例えば，Oracle における独自拡張関数である DECODE を用いた「DECODE(key, val, res1, res2)」という構文を，PosgreSQL における同様の意味をもつ構文である「CASE key WHEN val THEN res1 ELSE res2 END」に変換する．

未実装機能変換 新データベースに置き換える予約語, 関数, 構文が無い場合, データベースが備えるストアドプロシージャ，ユーザ定義関数等，ユーザが独自にロジックを定義できる機能を用いて，等価のロジックを定義し，そのロジックを呼び出すように SQL を変換する．例えば，Oracle における文字列を連結する関数である「CONCAT(string1, string2)」と等価のロジックは，PosgreSQL には存在しない．そこで，PosgreSQL のユーザ定義関数機能を用いて，事前に以下のように等価のロジックを定義し，それを呼び出す.

```
CREATE FUNCTION CONCAT (char, char)
RETURNS char AS 'SELECT $1 || $2' LANGUAGE ‘ SQL';
```

事前定義変換 旧データベースにおいて，ストアドプロシージャ，ユーザ定義関数等, ユーザが独自にロジックを事前に定義していて，それを呼び出している場合, 事前定義ロジックと呼び出し SQL の両方を変換する必要がある. 例えば, 呼び出し SQL に関しては, Oracle におけるストアドプロシージャ呼び出しである「EXECUTE procedure(attr1, attr2,...)」という構文を, PostgreSQL におけるユーザ定義関数呼び出しである「SELECT procedure(attr1, attr2,...)」に変換する．事前定義ロジックに関しては，事前に変換して定義しておく．

6.2.3 SQL 変換処理の検証

SQL 変換を行うことができても，さらに，その変換した SQL が正しく動作しているかを検証する必要がある．ここで「正しい動作」とは，変換後の SQL を置き換え後の新データベースに投入した結果が，変換前の SQL を置き換え前の

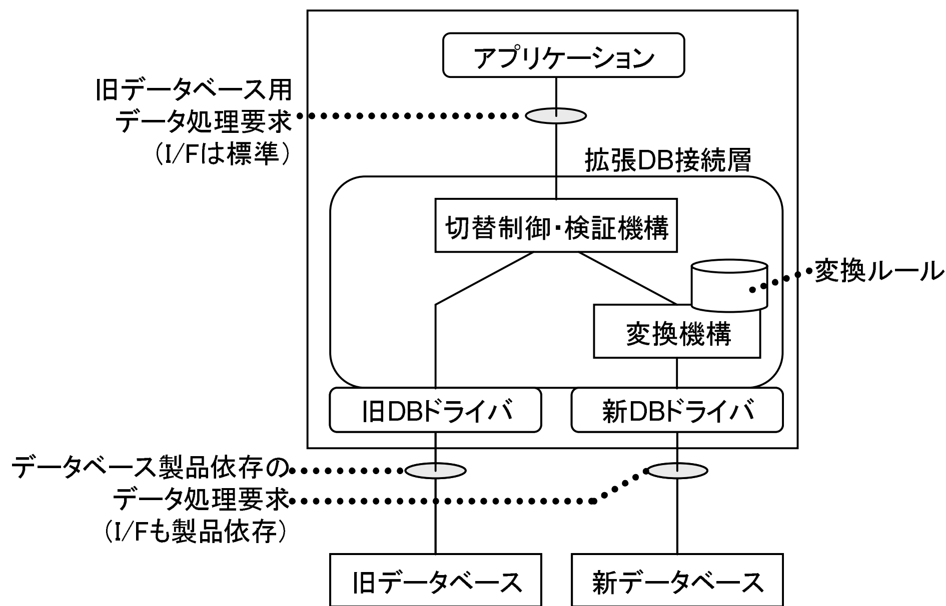


図 53 SQL 検証方式

旧データベースに投入した結果をと同じであるということである．すなわち，新データベースがエラーを返さずに正常な応答をしたとしても，それが正しい動作であるとは限らない．正しい動作か検証するためには，旧データベースを用いたシステムと，新データベースを用いたシステムとが，同一の振る舞いであることを確認するが，実際のシステムでは，その作業は困難である．オンライン中のシステムを停止させ，あらゆるパターンの応答について，テストするには時間がかかる．また，もし不具合があった場合に，元のシステムに容易に戻すことができることも必要である．

そこで，図 53 に示すように，SQL 変換の前に，さらに SQL を多重化できるようにデータベース接続層を拡張し，2 つのデータベースと同時に接続できるようにすることで，SQL 変換処理の検証を行う機構を検討する．

旧データベースと新データベースを併用し，多重化した SQL を，一方はそのまま旧データベースに投入し，もう一方の SQL を変換した後，新データベースに投入して，双方の処理結果を比較する．検証を含めた置き換えの手順は図54 の

ようになる．

1. 置き換え前：旧データベースのみで動作している状態．
2. データ移行：アプリケーションを一旦停止し，旧データベース内のデータを変換して新データベースに移行．
3. 検証：SQL 変換，検証方式を導入し，旧データベースと新データベースを併用して動作を検証する．検証の結果，問題があれば変換ルールを追加，修正する．なお，仮に問題が発生しても，新データベースを切り離し，旧データベースでデータ処理を続けられるので，アプリケーションに対して影響は無い．
4. 置き換え完了：検証期間を経て，安定性を確認した後，旧データベースを切り離し，新データベースのみで運用することで，データベース置き換えを完了する．

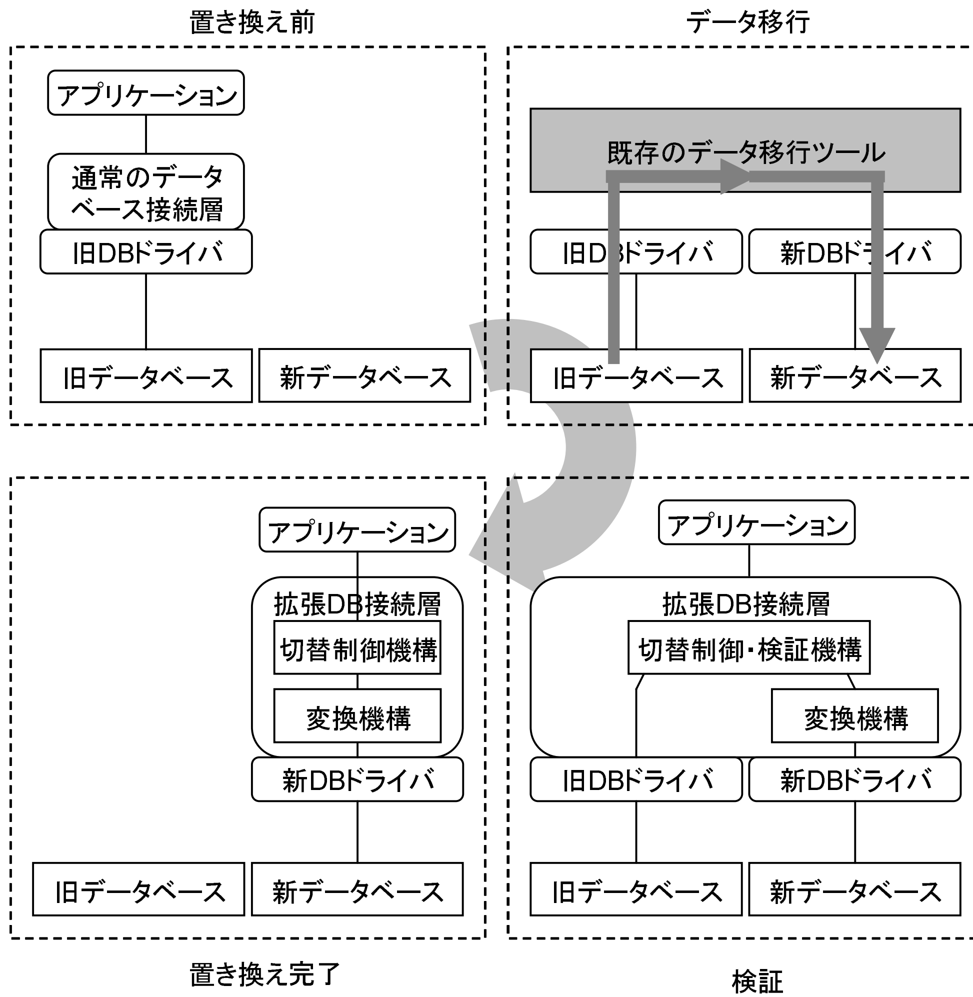


図 54 データベース置き換え手順

表 12　有効メソッド

クラス	メソッド
Connection	nativeSQL(), prepareCall(), prepareStatement()
Statement	addBatch(), execute(), executeQuery(), executeUpdate()

6.3　SQL 変換・検証機構の実装

Java におけるデータベース接続層である JDBC を拡張し，SQL 変換機構と SQL 検証機構を実装した． 具体的には，SQL 変換機構および検証機構を JDBC ドライバに組み込んだ．これを拡張 JDBC ドライバと呼ぶ．アプリケーションから，拡張 JDBC ドライバを利用するには，JDBC ドライバのロードパスを変更する．

今回，SQL 変換処理を組み込んだのは， 表 12 に示すように，Connection クラスと Statement クラスの 2 つである．各メソッドは，SQL 文字列を引数として取る java.sql パッケージ内のクラスのメソッドである．

変換処理としては，java.util.regex パッケージに実装された正規表現クラスを利用した．すなわち，変換前のパターンを正規表現で記述し，それにマッチするパターンを，後方参照を含んだパターンに変換する．

検証処理としては，JDBC の各メソッドの返り値を比較することで，検証を行う．検証結果は，Log4J パッケージを介して，ファイル等に出力される．

6.4 SQL 変換・検証機構の評価

本節では，提案方式を実装した拡張 JDBC ドライバの機能と性能の評価について述べる．

6.4.1 評価方法

提案方式を用いることにより，システムのデータベースを置き換えても，アプリケーションを変更することなくシステムを運用することができる．しかし，同時に，SQL 変換による変換オーバーヘッドにより，データベース処理の遅延が予想される．そこで，Oracle 対応のデータベースアプリケーションにおいて，実際にデータベースを Oracle から PostgreSQL に置き換えて，提案方式の機能と性能を評価する．機能に関しては，データベースを PostgreSQL に置き換えた後，拡張 JDBC ドライバを適用しなかった場合，および適用した場合の動作について評価する．性能に関しては，データベースを PostgreSQL に置き換えた後，アプリケーションを PostgreSQL 用に改造した場合を基準とし，拡張 JDBC ドライバを適用した場合の処理性能について評価する．

なお，SQL 変換・検証の評価アプリケーションの詳細は，付録に記載する．

6.4.2 評価結果

データベースを PosgreSQL に置き換えた後，評価用アプリケーションをそのまま使用した場合，評価用アプリケーションを PostgreSQL に改造して使用した場合，そして提案手法を適用し，かつ変換ルール数を変更した場合で，それぞれ 10000 回の商品発注を行い，その処理時間を計測した．結果を表 13 に示す．

なお，今回の無改造 Oracle 対応評価用アプリケーションが PostgreSQL で動作するためには，最低 4 つの変換ルールが必要であるため最小値は 4 となる．(4)～(9) はルール数による処理時間の変更を評価するため，今回は試行されるが適用されないルールを追加した．また，割合は (2) 提案方式未適用時を基準とした場合と適用時との処理時間の差分の割合である．

表 13　評価結果

#	使用アプリケーション	変換ルール数	平均処理時間	割合
①	無改造Oracle用AP	未適用	動作しない	--
②	PostgreSQL用改造AP	未適用	63.884	基準
③	無改造Oracle用AP	4	76.594	19.90%
④		5	80.198	25.50%
⑤		10	86.779	35.80%
⑥		15	93.218	45.90%
⑦		20	98.257	53.80%
⑧		30	110.038	72.20%

提案方式適用時の (3) は約 20 %程度の処理性能低下が確認された．変換ルール数を変化させた (4)〜(8) の割合の増分から，今回は変換ルールを一つ増やす毎に，約 2 %の性能低下が起きていることが分かる．今回の実装では，Java 標準の正規表現処理ライブラリを用いたルールの照合を行っており，その処理がオーバーヘッドの大きな要因となっている．今後，ルール照合処理の改善が必要である．

6.5　データベース置き換えのためのデータベース接続層の拡張のまとめ

本章では，データベース接続層を拡張することで，アプリケーションを変更することなくデータベースの置き換えを支援する方式について述べた．本方式は，データベース接続層において，旧データベース用の SQL を新データベース用の SQL に変換することで，データベースの置き換えを可能とする．さらに，旧データベースと新データベースを併用し，双方の応答を比較することで，SQL 変換の動作を検証することを可能とした．

また，Java におけるデータベース接続層である JDBC を拡張して提案方式を実装した拡張 JDBC ドライバを開発し，Web ショップにおけるデータ処理を想定したデータベースアプリケーションに適用して，評価を行った．その結果，提案方式を用いることにより，処理性能は 20 %程度低下するが，データベースを変更しても，アプリケーションを一切変更することなく，JDBC ドライバのロード

パスを変更するだけで，正常に動作することを確認した．

7. アプリケーションの多重実行のためのデータベースアクセス管理

7.1 アプリケーション多重実行

3 章，4 章においては，データを保護する目的で，データベース・サーバの多重化について議論した．これに対して，本章では，処理そのものを保護する手法について議論する．つまり，アプリケーションを多重化し，同一の処理を冗長に実行することで，処理途中で障害が発生し，一方のアプリケーションの処理が停止したとしても，もう一方のアプリケーションの処理で，サービスを継続する．

従来のサービス継続手法としては，[20][24][60] 等がある．しかし，多くのアプリケーションは，外部リソースと連携することで処理を行うため，単純にアプリケーションを多重実行するだけでは耐障害機能を実現できない．例えば三層モデルに基づく Web アプリケーションの場合，アプリケーションはデータベースと連携しサービスを提供する．しかし，図 55 に示すように，多重実行されたアプリケーションが１つのデータベースを共用する場合，複数のアプリケーションがデータベースを重複して更新してしまうという問題がある．データベースを系毎に用意するという回避策もあるが，データベースを複数用意することは，コストの面で不利である．

本章では，このような問題を解決するために，アプリケーション多重実行環境において，共用する単一のデータベースへの矛盾の無いアクセスを実現する方式を提案する．以下に，提案方式の構成および動作について述べる．

7.2 データベースアクセス管理

前述したアプリケーションの多重実行の課題を図 56 に示すように制御することで解決する．つまり，データベースを多重に更新しないように，主系のデータベース接続層と予備系のデータベース接続層が連係して動作するようにし，予備系のデータベース処理要求が，データベース・サーバに発行されないようにする．その代わり，主系のデータベース接続層から，データベース・サーバの応答を受

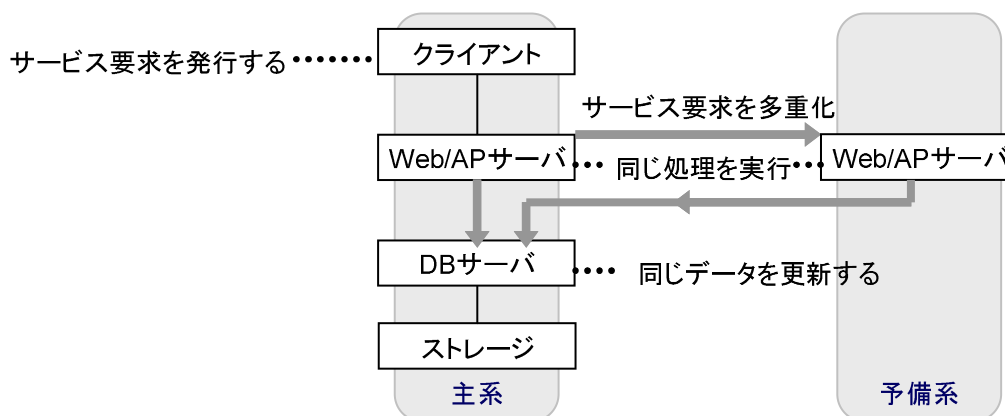


図 55　データベースの重複更新

け取り，それをアプリケーションに返すことで，予備系のアプリケーション処理を続けられるようにする．

以上のように，予備系アプリケーションは実際にはデータベースアクセスを行わないが，あたかもデータベースアクセスを行ったかのようにデータベース応答を受け取るようにすることで，データベースを重複して更新することなく，主系と同様の処理を行うことができる．

もし，主系アプリケーションにおいて障害もしくはその予兆を検出した場合，異常動作の可能性のある主系アプリケーションからのデータベース処理要求は破棄し，予備系アプリケーションからの処理要求をデータベースに転送するようにする．主系アプリケーションと予備系アプリケーションは同じ処理状態にあるので，このようにアクセス経路を切り替えるだけで，初期化処理等を行うことなく，予備系アプリケーションが高速に処理を引き継ぐことができる．

7.3 データベースアクセス管理機構の評価

提案したデータベースアクセス管理方式を実装し，実用性を検証するために，実運用されている製品に適用して，評価実験を行った．

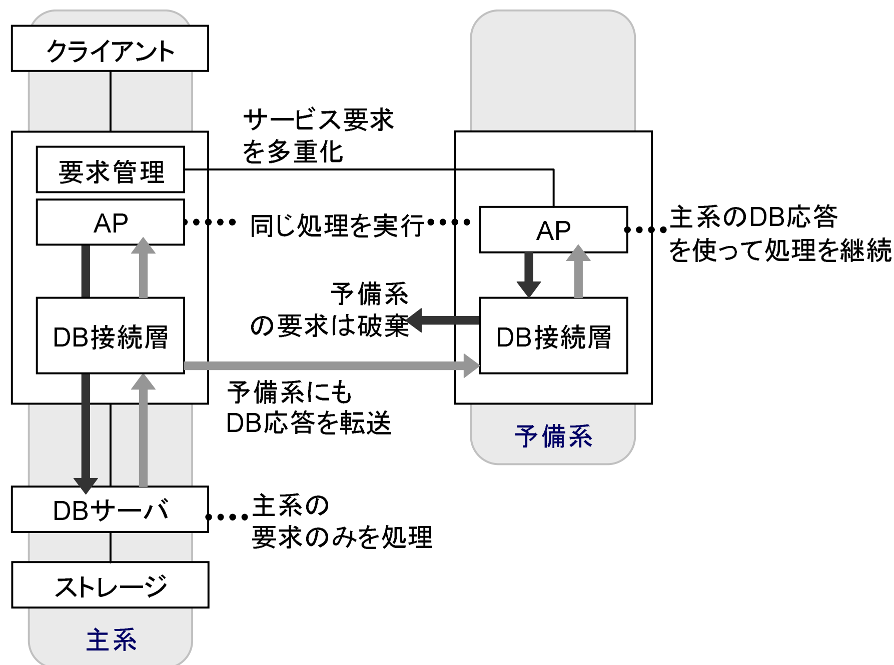


図 56 データベースアクセス管理

7.3.1 評価環境

図 57 に示すように，主系アプリケーション・サーバ，予備系アプリケーション・サーバ，データベース・サーバ，そしてクライアントの４台をネットワークで接続した．アプリケーション・サーバには，BEA WebLogic Server 8.1，Java 実行環境として Sun JDK 1.41，Web アプリケーションとして住宅業向け工事発注 ASP サービス [62] をそれぞれ使用した．データベース・サーバとして Oracle9i Database Standard Edition を使用した．クライアントには，Web クエリ発生ツールとして Microsoft Web Application Stress Tool 1.1[23] を使用した．

7.3.2 実験方法

提案方式のオーバーヘッドを評価するため，管理機能の無い通常状態と，管理機能を適用した状態とで，Web アプリケーションに対し，以下の動作を行う Web

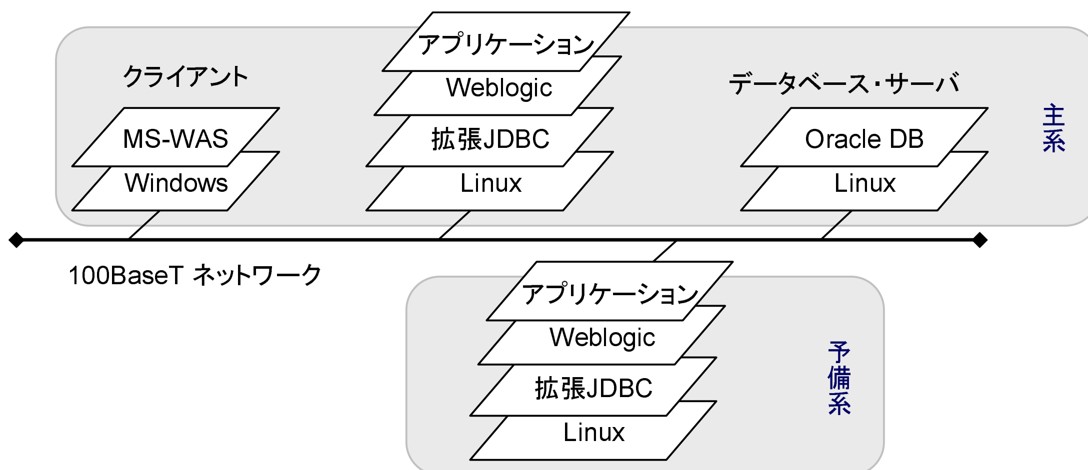


図 57　データベースアクセス管理実験システム構成

クエリを，それぞれ 30 秒間連続で投入し，その処理数を計測した．

検索 1000 件の発注データから特定の発注データを検索

登録 発注データを新規で登録

なお，Web クエリには，Web アプリケーションへの処理要求だけでなく，イメージの取得要求など，データベース処理要求を行わないものも含まれる．また，データベース処理要求を行う Web クエリも，その処理内容によって発行するデータベース処理要求数は異なる．そこで，オーバーヘッドの詳細を検討するために，データベース処理要求を行う Web クエリについて，個々の処理時間と，発行されるデータベース処理要求数も計測した．

7.3.3 実験結果

Web クエリの処理数および処理時間を計測した結果を表 14 に示す．データベースアクセス管理機能を適用することにより，全体で 25〜35 %程度のオーバーヘッドが発生している．

表 14 各 Web クエリの処理時間

WEBクエリの種類		発行データベース処理要求数	Webクエリの処理時間(ミリ秒)		データベース処理要求の処理時間(ミリ秒)		
			通常時	適用時	通常時	適用時	差分
1	ログイン画面	22	70	74.1	3.2	3.4	0.2
2	ログイン処理	36	85.1	134.2	2.4	3.7	1.3
3	ユーザ情報	22	64.9	70.1	3	3.2	0.2
4	操作メニュー	22	47.7	70.7	2.2	3.2	1
5	初期画面	22	72	85.5	3.3	3.9	0.6
6	検索画面	127	197.2	294.8	1.6	2.3	0.7
7	検索処理	132	166.6	297.6	1.3	2.3	1
8	登録画面	721	490	1005.5	0.7	1.4	0.7
9	仮登録処理	154	279.9	372.8	1.8	2.4	0.6
10	登録＆電子署名処理	721	507.8	1126.7	0.7	1.6	0.9
合計		1979	1981.2	3532	平均: 1.0	平均: 1.8	平均: 0.8

さらに，実際にデータベーストランザクションを発行する Web クエリ 10 種の処理時間と，発行されるデータベーストランザクションの数を計測した結果を図 58 に示す．通常時と適用時との Web クエリの処理時間の差分は，発行されるトランザクションの数にほぼ比例しており，１トランザクションあたり平均 0.8 ミリ秒のオーバーヘッドが発生している．

インターネットアプリケーションでは，ネットワーク応答に数百ミリ秒かかるものであり，また人とのインタラクション操作を含む．それらの遅延に比べれば本手法のオーバーヘッドは十分小さく，許容範囲であると考える．

しかし，バッチ処理プログラムのように連続して，大量にデータベース処理要求が発行されるような場合には，１データベース処理要求あたりの処理時間は短く，本提案手法では，オーバーヘッドが大きく，性能的に大きな問題がある．今回の実装では，すべて同期処理するようにしたが，4 章で述べたように，必要最小限のデータベース処理要求のみ同期するような機構を導入することで，性能向上を図ることが可能であると思われる．

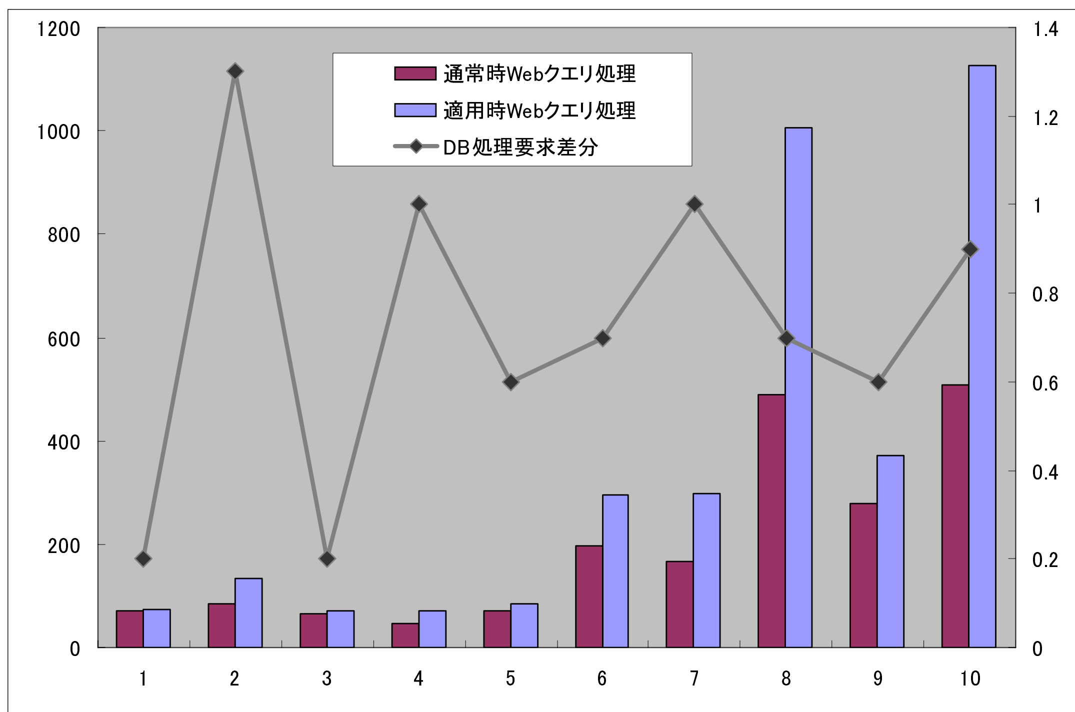


図 58　Web クエリの種類と提案方式によるオーバーヘッド処理時間

7.4　アプリケーション多重実行に関する考察

7.4.1　環境多様型多重実行

上記では，アプリケーション・サーバの停止，特にハードウェアの故障を想定したが，データ保護を目的とした耐障害と異なり，処理プロセスを保護する場合，ソフトウェアの不具合や，利用者・管理者の操作ミスといった障害の可能性もある．その場合，単に多重化しても，予備系でも障害は顕在化してしまう．

そこで，予備系では，主系とは異なる環境で実行するようにし，障害の顕在化を抑制することを検討した．

異なる環境とは，異なる OS，異なるメモリ量，異なるアプリケーション設定などの構成情報，さらに，ソフトウェアバグの同時顕在化の回避という意味で，異なる実行タイミングで冗長系を動作させる．実行タイミングを遅らせることで，

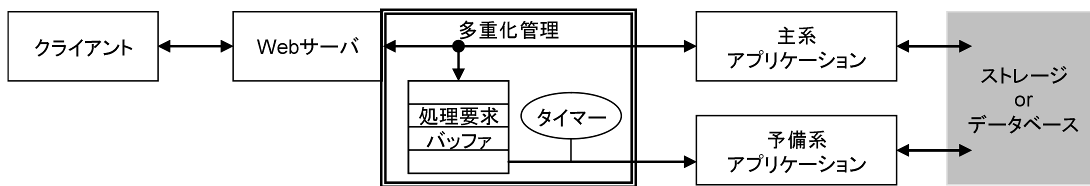


図 59 提案方式の基本構成

主系において障害が発生した後，予備系でその障害が顕在化する前に，何らかの障害対処のアクションを実行できるようにする．

図59に，本提案の基本構成を示す．クライアントからの処理要求を主系と予備系に分配する．各系で，多重に処理が行われるが，予備系への処理要求は，バッファを経由する．基本的に主系の処理結果のみがクライアントに送信される．このようにバッファを含む構成とすることで，予備系と主系とが非同期で動作することを可能とする．

このような構成においては，図60のような障害対処が考えられる．主系は，処理要求 Tx を受信し，処理応答 Rx を送信する．予備系は，主系の処理を確認後に複製された処理要求 TX を処理し，検証用応答 RX を返す．

予備系は主系に比べ，ある時間 t だけ遅延して処理を実行する．遅延の度合いは，処理要求数を設定してもよいし，時間を指定してもよい．

主系において，パニック障害を検知した場合，予備系において，実行状態のイメージ保存を実行する．実行状態のイメージ保存は，OS の機能を利用する．予備系は，ネットワークの設定のみを変更することで，主系として機能を開始する．

必要であれば，予備系では，何らかの障害対処を行う．例えば，問題が発生した処理要求を拒否する，ログレベルを変更する，システムダンプを実行する，などが考えられる．

もし，代替リソース (別のコンピュータ) を用意することができるのであれば，前述の実行状態のイメージをマイグレーションし，新たな予備系として動作させてもよい．予備系の起動に時間がかかるかもしれないが，処理要求はバッファに保存されているので，これを順次処理し，漸次通常状態に復帰する．

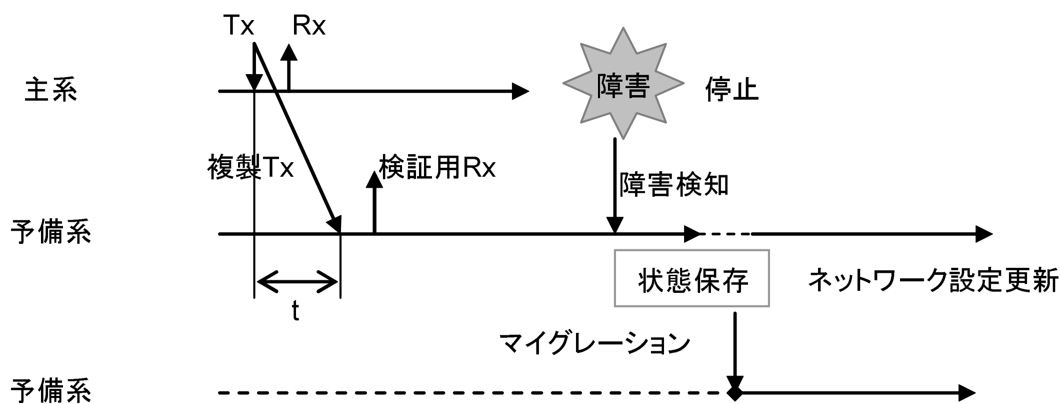


図 60 耐障害のシナリオ例

なお，VMware，Virtual PC，Xen といった仮想マシン技術を用いることで，多様な環境を容易に構築することが可能である．状態保存，マイグレーションも仮想マシン管理の機能に含まれているため，このような耐障害システムを構築することが容易となる．しかしながら，アプリケーションの障害は多様であり，このような耐障害機構がどの程度有用かを評価することは困難である．例えば Fault Injection をランダムに行うようなツールを利用するなどして，信頼性を評価するなどが考えられる．

7.4.2 操作ミスに対する対処

障害の原因が操作ミスである場合，その再発防止の対処として，マニュアル（手順）の見直し等が多いが，どのような手順であっても，どのような訓練を受けていても，ミスは不可避である．適切な情報提示や対話手法によってミスを低減する機構など“ 間違いを未然に防ぐシステム ”であるだけでなく，操作ミスが起こることを前提としたアンドゥ（UNDO/後戻り）機構など“ 間違えても大丈夫なシステム ”であることが重要である．

また，障害の種類によっては，障害原因の追究，再現性検証，復旧計画立案と顧客承認を経て，対処が実行されるように，障害対応はいくつかのステップを踏

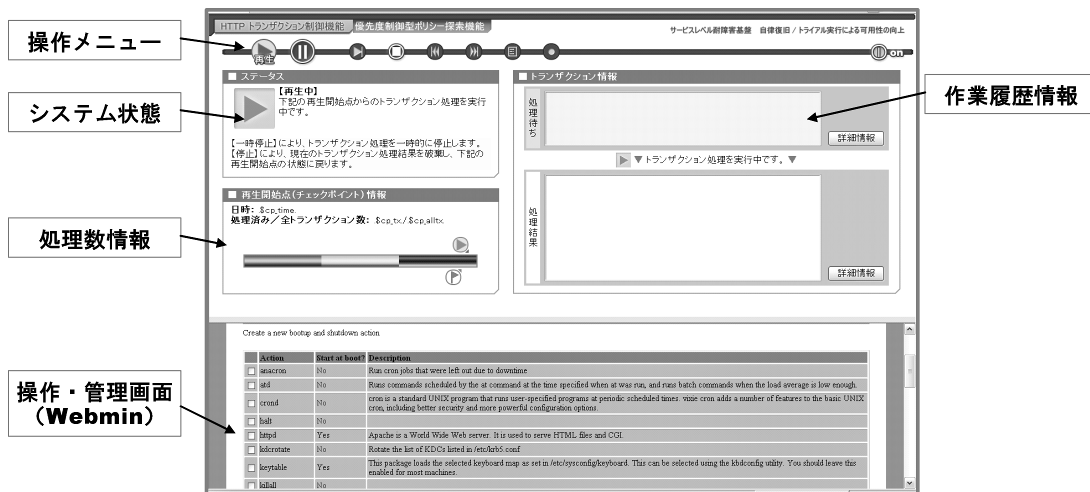


図 61　障害管理画面例

む．自動化する場合，このように人間（管理者）が必要に応じて割り込み，システムと対話しながら，障害対処を進めることができるようなユーザインタフェースが重要である．

このようなアンドゥ機能および対話機能を踏まえて，アプリケーション多重化管理画面を設計した．システム管理画面の一例を図61に示す．ウィンドウの下半分が既存のWeb運用管理システムで，上半分がアプリケーション多重化管理画面である．管理者がWeb運用管理システムで管理作業を行うと，その作業（Webクエリ）はすべて記録され，作業履歴情報，処理数情報に反映される．

操作メニューには,「再生」,「記録」,「停止」,「巻き戻し」などのボタンがある．「記録」で，その時点でのシステムのスナップショットを保存できる.「巻き戻し」は，操作ミスなどがあった場合に用いる.「巻き戻し」を行うと，以前に記録した状態にシステムを復旧できる.「再生」は，巻き戻し後に再度同じ作業を適用する場合に用いる．作業履歴のウィンドウでは，一覧リストから各作業の選択および解除を指定でき，問題（操作ミス）のあった作業を除外して再生すれば，システムの管理作業を継続することができる．

7.5 アプリケーションの多重化のまとめ

以上，多重実行された複数の AP が，共用する単一のデータベースに矛盾無くアクセスを行うためのデータベースアクセス管理方式について述べた．本方式は，JDBC の拡張であるため，OS やデータベース・サーバを限定せずに適用可能である．さらに，実運用されている製品に適用して評価実験を行い，提案方式のオーバーヘッドが，データベース処理や人間の操作による待ち時間に比べれば，十分に小さく，Web アプリケーションにおいては，本方式は十分実用的であることを確認した．

また，サービスアプリケーションを多重に実行することで，システムの耐障害性を高める技術に関して，HTTP 層で処理要求を多重化し，多様な実行環境で，特に実行タイミングを遅らせて処理する耐障害機構をそのユーザインタフェースも含め検討した．

8. 結言

8.1 研究の成果

本論文においては，現在の社会生活を支える情報通信技術（ICT）サービスの信頼性向上のため，システムを構成するモジュールを多重化する技術について議論した．障害が発生した場合に，その障害の発生した部位を迅速に発見し，切り離し，同等の機能を有する代替モジュールに交換するのが，一般的な障害復旧手順であり，構成モジュールの多重化管理機構は，信頼性向上の基本的な技術として重要である．

本論文では多重化管理技術の１つとして，データベース接続層を拡張する方式を提案した．ICT サービスのシステム構成の多くが，サービスとデータを分離，つまりアプリケーション・サーバとデータベース・サーバとから構成されることに注目し，アプリケーション・サーバにおける汎用的なインタフェースとなっているデータベース接続層を拡張することで，主要構成要素であるデータベース・サーバを多重化，あるいはアプリケーション・サーバを多重化することを可能とする．

具体的には，以下の３つの方式を開発した．

1. 負荷分散構成のアプリケーション・サーバにおいては，多重化管理が分散処理となるが，その整合性管理のためにセマフォサーバ方式を開発した．多重化管理を安全にアプリケーション・サーバにおいて実行できるため，データベース・サーバの負荷を増大させることがない．さらに，同時接続数が増えた場合には，拡張したデータベース接続層が，データベースへのアクセスを平準化するため，性能低下は発生しないことを確認した．

2. アプリケーションの多くは，標準的なデータベース接続 API を利用しており，この API を提供しているライブラリを，本研究で開発したものと置き換え，冗長構成部と接続することで，容易に冗長構成に変更できる．つまり，既存アプリケーションのプログラムソースの変更が不要である．また，データベースのアクセスログをデータベース接続層でバッファ管理するこ

とによって，冗長のデータベースの接続，および切り離しを，サービスを停止させることなく行うことができる．このようにサービスを停止させることなく，無停止で構成変更を可能とする無停止フェイルオーバー・フェイルバック方式を開発し，柔軟な多重化管理を可能とした．

3. データベース接続層で交換されるデータは，一般的にSQL形式であるので，処理種別などの監視・可視化が容易である．その監視結果に応じて，複製処理を柔軟に制御，あるいはログを柔軟にフィルタリングすることを可能とする．つまり，処理要求ごとに，同期複製，順序管理非同期複製，非同期複製，複製なしの切り換え，あるいは予備系への転送のタイミング制御を柔軟に行うことが可能であることを確認した．

上記に示した方式を，Java のデータベース接続層である JDBC に適用し，ベンチマークソフトウェア等を利用して，動作評価および機能評価を行った．従来のデータベース複製方式と比べ，提案方式は処理性能に優れ，また多様な信頼性要件に応じての複製制御が可能であることが確認できた．

8.2 展望

データベース接続層の拡張　本論文で述べたデータベース接続層の拡張は，汎用的なトランザクション層における意味的な管理である．一方，従来，ストレージ層もしくはファイルシステム層で多重化管理を実現してきた．データもしくはファイルそのものを複製するには十分であるが，大規模複雑化したICTシステムでは，不十分である．その理由は，構成要素が複雑に連係しており，また依存関係があるため，複数のデータ，ファイルを同時に更新するような作業が増大しているためである．データベースの場合は，チェックポイントを設定して，データベース全体で整合性がとれた状態で，バックアップを行う．同様に，システム全体で整合性を管理し，バックアップを取得するような仕組みが必要である．もし，外部のサービス，データを用いているのであれば，ますます整合性管理は困難である．

サービスが高度に連係するようになると，各種サービス間でトランザクションが発生し，トランザクション処理は増大し，従来のデータ単位，ファイル単位の

多重化管理では，障害発生時の不整合は避けることはできない．システムソフトウェアレベルで，適切なタイミングでスナップショットを取得する手法も考えられるが，そのタイミングの取得には，アプリケーション処理の意味，トランザクションの意味をモニタリングする必要がある．また，予備系への転送処理に遅延があれば，その遅延にあわせて，アプリケーション処理，トランザクション処理を制御する必要がある．つまり，システムソフトウェアレベルとサービスレベルとを連係させる必要がある．

実際，Web サービスにおいては，従来のトランザクションよりも大きな単位で，一連のビジネスロジックを捉えるトランザクション・コンテキストが一般的になりつつあり，トランザクションレベルも，2 フェーズコミット，ロング・トランザクションといった高度なトランザクションが広く使われ始めてきた．トランザクション・コンテキストは，SOAP メッセージでコンテキストをやり取りし，上位のサービスレベルで，コンテキストを管理するものであるため，本提案技術とは異なるレイヤを対象としている．コンテキストを管理するコンテキスト・サービスやアクティビティ・ライフサイクル・サービスと通信することで，将来的には，コンテキストを考慮したデータ複製制御も可能であろう．

従来，Web サービスレベルのトランザクションプロトコルとしては，BTP（Business Transaction Protocol），WS-Transaction，WS-CAF があり，標準化は混沌としている状況であった．2007 年 5 月に，Web サービス関連の主要企業が支持する WS-Transaction 1.1 が OASIS に承認され，またロイヤリティ・フリーであることから，今後有力となると思われる．本論文で提案した機構と連携させる手法の具体化が今後の課題である．

多重化構成管理　本論文では，主にデータベース・サーバの多重化構成管理を議論し，一部アプリケーションの多重化実行にも言及した．これは，現状のサービスの多くが，三層モデル構造となっていることを前提としている．しかし，将来的にも，同様のサービスモデルが継続するとは限らない．例えば，計算機およびネットワークの仮想化技術が広まりつつあり，システムの論理構成と物理構成を分離する動きがある．サービスの位置やリソースを気にすることなくサービスを構築するようになりつつある．

そのような場合，本論文ではサイト単位でディザスタリカバリするように想定したが，論理構成レベルと物理構成レベルで，それぞれでディザスタリカバリ，あるいは多重化管理をすることとなる．物理構成レベルでは，従来のストレージレイヤの多重化管理の発展形が考えられる．論理構成レベルでは，意味レベルとなるので，本論文で議論したトランザクション層での多重化管理が一つの候補となる．トランザクション層としては，データベース接続層の他に，WS-RM といった Web サービスのトランザクション管理もある．この時，多重化管理の対象は，サービス全体であり，トランザクションを監視し，仮想化マシン，仮想ネットワーク等の構成要素を制御することで，多重化管理を行うことが考えられる．

ただし，論理構成と物理構成を分けることで，開発者の負担が少なくなるなどの効果がある一方で，処理のオーバーヘッド，サービス障害発生時の障害部分の特定ができないなどシステム全体動作の把握が困難，といったデメリットも大きく，ビジネスにおける普及には時間がかかると思われる．

信頼性（耐障害性，障害回復性）の定量化指標の確立　本文においては，サービス要件に応じて，適切な多重化方式を選択し，また柔軟な多重化方式が必要であることを述べた．しかしながら，どのようにしてサービス要件が導出されるのかは明確でない．サービスの売り上げ，あるいは ROI（Return of Investment）は，サービス事業主には理解しやすい指標である．単位時間あたりの売り上げは，もしサービスが停止すれば，利益の逸失となる．それに RTO を乗じた金額とサービスの構築・運用コスト [49] とを比較することで，RTO は意味のある数字となる．

論文中では，可用性について，停止障害が発生する確率等のパラメータを導入して算出する数式を定義した．しかし，各値は，サービスの種類，環境に大きく依存し，それらの値をどのように設定すればよいかが明確になっていない．

処理中率に関しては，ハードウェアの機種選定のために，アクセス数を見積もることはしているので，大まかな値の設定は可能である．通常は，ピーク性能にあわせて，機種選定がなされるため，処理中率は数%程度である．タイムアウト時間に関しては，通常のインターネットサービスでは，数十秒を要するような重たい処理を行うことはないため，数秒程度で十分である．fsck，ブート，ロールバック，ロールフォワードも，システム設計時に定まる．

それに対し，ストレージ，データベース・サーバ，アプリケーション・サーバの障害が発生する確率について，設定することは困難である．１つには，標準的な環境を用意して，メーカーあるいは第三者機関が測定することが考えられる．しかし，その費用負担は，製品価格に反映されることになるが，どれだけ受け入れられるかは疑問である．また，利用者の環境はさまざまであり，その多様性を考慮することができないことも課題である．

別の案としては，製品の修理に関して，各メーカーの記録を公開することである．重大な障害の場合には，国や自治体が調査をし，調査結果を公開するが，一般的な障害情報は企業秘密であり，公開することは考えられない．

そこで，インターネット上に，各利用者個人が障害事例を報告するサイトを用意し，各個人に協力を求める案を提起したい．このように事例を収集することで，信頼性（耐障害性，障害回復性）の定量化が可能となると考える．障害状況を正しく把握することで，製品やシステムの信頼性向上に役立つものと考える．

いかに簡単に事例を報告可能とし，またいかに正確に報告可能とするか，報告者のプライバシやセキュリティ，さらにはサイトの運営主体，運営費をどうするかなど，検討すべき課題も多い．

地域分散性　本論文の対象としたのは，2 地域に分散したディザスタリカバリまでである．企業活動がグローバルとなり，インターネットが広く利用されている現在，ICT サービスの予備拠点もグローバルであるべきで，またより強固な信頼性を実現するには，多地点であることが望ましい．

本文では，東京，大阪間の 2 地点間でのデータ転送を想定し，10 ミリ秒という遅延を設定してシミュレーション実験を実施したが，上で述べたように，グローバルな多地点間となると数百ミリ秒の遅延を想定する必要がある．また多地点となると，遅延の幅も大きくなる．このような場合の複数データベース間での効率的同期手法についてはいくつかの研究開発が進んでいる．しかしながら，フェイルオーバーは可能でも，冗長体制に復帰するフェイルバック機能は発展途上である．例えば，複数のアプリケーション・サーバが多地点に分散していて，１つのデータベースに障害が発生した場合，残りのデータベースを使って，処理を続けることは難しいことではない．しかし，障害を起こしたデータベースを復旧する

ためには，アクセスログが必要であり，そのアクセスログは，継続サービス中のデータベースのアクセスログと整合していなければならない．また，そのアクセスログが単一障害点とならないようにする必要がある．

今後，提案技術を，性能と信頼性とのバランスを保ちながら，多地点分散を実現できるように拡張，発展させたい．

運用管理の自動化に向けて　運用管理の自動化は，Autonomic Computing や Utility Computing のように，各システムベンダで研究開発が進められている．その基本技術は，監視，判断，対処の自律ループにある．本論文では,（障害）対処の実行手段として，多重化構成管理に注目して議論した.「監視」に関しては，SNMP, WBEM といった標準技術の利用,「判断」に関しても，SML（Service Modeling Language）の標準化が進められている．運用管理の自動化のためには，このような標準と組み合わせていく必要がある．

また,「対処」に関しても，ITIL や ISO/IEC 20000 にあるようなプラクティスが標準的な運用管理手法として，広く認知しはじめており，これらに準拠することが考えられる．

しかしながら，すぐにあらゆる状況において適用可能な自動化を実現することは不可能である．まずは，限定的な範囲で，また運用管理者の作業を支援するような形で，段階的に自動化を進めていく必要があると考える．例えば，運用管理者が次にどの部分の何を調べるべきか，次に何の行動をするべきかの候補を示して，ナビゲーションするようなシステムが考えられる．

さらに，運用管理においても，上述したサービスの連係に関連し，複数の計算機やシステムソフトウェアの設定を整合させて，安全に設定変更させることが難しくなってきている．適切なタイミングでシステム全体に対して，矛盾なく設定変更を可能とするような機構が必要であろう．

また，自動化の１つのステップとして，学習型運用管理方式を議論した．従来の自律運用管理の研究の多くは，ルールベースであるが，そのルールをいかに作成するかについては，ほとんど議論がなされていない．機械学習を利用することで，ルールを作成することなく，運用管理の自動化を実現できる可能性を示したが，本論文で採用した SVM は，入力ベクトルと結果の関係，たとえばどの入力

ベクトルが結果に大きな影響を与えているかなどが，ブラックボックス化するために，ある事象に対して，どのように改善すべきかの分析が困難である．つまり，本論文では，ある定型のパターンに応じて，転送する，しないの判断を行うものであったが，どのくらい定型パターンがあって，またどのくらいランダムなパターンがその定型パターンの予測判断に影響を与えているかは，別途サービスごとに分析する必要がある．このようなサービスの分析は，多重化管理の性能，信頼性改善だけでなく，サービス全般の改善にも不可欠であり，サービスの分析手法についても，今後研究を深めていく必要があると考えている．

総合的なサービス品質改善　サービスの信頼性に関連する領域，技術は，図 1 に述べたように幅広く，本論文で議論したのは，一部分にすぎない．真のサービスの信頼性向上を実現するには，サービスシステム構築の観点では，要件定義，開発（ソフトウェア），テスト，運用管理手法まで含めて議論する必要がある．設計・開発ツール，テストツール，運用管理ツールが連携することで，サービスの要件に応じて，何をテストすればよいか，何を監視すればよいか，さらには何を改善したらよいかの支援が効率的に行えるようになる．

　また，信頼性の観点では，セキュリティ，保守性など，その他の信頼性をあわせ品質評価，改善が必要である．さらには，クライアント（端末）やネットワークも含めた信頼性の評価も必要であろう．もちろんビジネスの観点から，ICT システムの振る舞いだけでなく，サービス実務者（ヒト）の振る舞いも考慮すべきである．このようなサービスの品質を総合的に評価するための研究は，今後ますます重要となる．

謝辞

三年半の博士課程在籍期間中、多くの方々からご支援・ご指導を賜り、本博士論文を無事作成することができました。ここに感謝の意を表します。

奈良先端科学技術大学院大学情報科学研究科情報システム学専攻の砂原秀樹教授からは，本研究を行う機会を与えていただき，研究の節目節目で重要の助言をいただくだけでなく，インターネット全体に関し幅広いご指導をいただき，深く感謝します.

奈良先端科学技術大学院大学情報科学研究科の山口英教授からは，研究指導をいただき，また有用な助言，提言，示唆をいただき，深く感謝します.

日々の研究活動において，有用な助言をいただいた奈良先端科学技術大学院大学情報科学研究科の藤川和利准教授に深く感謝します.

研究に関する方針から，研究の進め方，研究内容の仔細な議論，論文の書き方に至るまで，多くの助言をいただきました奈良先端科学技術大学院大学情報科学研究科の河合栄治特任准教授に深く感謝します.

東京大学大学院新領域創成科学研究科基盤情報学専攻の中山雅哉准教授からは，遠隔ながら研究指導をいただき，また有用な助言をいただき，深く感謝します.

社会人学生ということで，スタッフの方々，研究室のメンバーには，多大な迷惑をかけてきましたが，いつも親切に，また丁寧に対応していただき，深く感謝します.

著者が勤務している日本電気株式会社の上司，同僚の協力なくしては，本研究を進めることはできませんでした．三年半の博士課程在籍期間中，博士号の取得に理解を示し，ご支援をいただいた上司，山之内徹所長，山田敬嗣所長，平池龍一部長，柏谷篤部長，福島俊一副院長，京都大学経営管理大学院の原良憲教授に深く感謝します．また，さまざまな面で協力をいただいた同僚，河野泉主任研究員，喜田弘司主任，原雅樹主任，宮崎陽司主任，藤山健一郎氏，大野允裕氏，今井照之氏に深く感謝します.

本研究の初期段階では，片山博支配人，笠原裕支配人，中田登志之主席技術主幹，妹尾義樹統括マネージャ，杉山高弘統括マネージャから多くの助言をいただきました．また，社内の事業部門の方々から，本研究を進めるに当たっては，関連

事業状況，関連製品市場動向など有用な情報をいただきました．深く感謝します．

平池部長，加藤清志主任，藤山氏とは，本研究をゼロから立ち上げ，事業化に向けての活動を共に進め，多くの成果をだしてきました．特に，藤山氏は，共同研究者として，研究のアイデアを共に考え，方式を考案し，試作を進め，結果に悩み，論文執筆を進めてきました．深く感謝します．加藤主任は，現在，事業部において，本研究の関連領域で事業化に取り組んでおり，ぜひ大きな事業成果に結びつけて欲しいと期待しています．また，池上輝哉主任とは，ユーザインタフェースの面からICTシステムの信頼性向上の議論を進めてきました．深く感謝します．

最後に，研究活動に理解を示し，応援してくれた家族に何よりも感謝します．

参考文献

[1] D. Andersen A. Snoeren and H. Balakrishnan. Finegrained failover using connection migration. *Proc.3rd USENIX Symp. on Internet Technologies and Systems(USITS)*, pp. 97–108, 2001.

[2] Tigran Aivazian. Linux kernel 2.4 internals, 2002.

[3] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 290–300, 1985.

[4] J. D. Kubiatowicz B. Y. Zhao and A. D. Joseph. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing,*. Univ. California, Berkeley, CA, Tech. Rep. CSD-01-1141, 2001.

[5] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, III W.N. Mills, Y. Diao. Able: a toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 2002/9.

[6] M. Carson and D. Santay. Nist net: A linux-based network emulation tool. *ACM SIGCOMM Computer Communications Review*, Vol. 33, No. 3, pp. 111–126, 2003.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. *Proc. of USENIX Annual Technical Conference, Freenix track*, 2004.

[8] C. Chang, C. Lin. Libsvm: a library for support vector machines, 2001.

[9] Fay Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon E. Perl, and Li Zhang. Myriad: Cost-effective disaster tolerance. *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 103–116, 2002.

[10] Oracle Corporation. High availability and disaster recovery. www.dlt.com/oracle/pdf/huntsville/highavaildr.pdf, 2002.

[11] P. R. Croll, et al. Dependable, intelligent voting for real-time control software. *Engineering Applications of Artificial Intelligence*, Vol. 8, No. 6, pp. 615–623, 1995.

[12] K. Marzullo D. Zagorodnov and T. Bressoud. Engineering fault tolerant tcp/ip services using ft-tcp. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, 2003.

[13] Kyle Geiger. *Inside ODBC.* アスキー, 1996.

[14] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD*, Vol. 25, No. 2, pp. 173–182, 1996.

[15] IBM. Oltp benchmark toolkit v 2.0, 2004.

[16] ISO. Iso/iec 9075 database language sql, 1999.

[17] ISO/IEC10026-1:1992. *Information technology – Open Systems Interconnection – Distributed Transaction Processing – Part 1: OSI TP Model.* 1992.

[18] James E. Johnson and William A. Laing. Overview of the spiralog file system. *Digital Technical Journal of Digital Equipment Corporation*, Vol. 8, No. 2, pp. 5–14, 1996.

[19] B. Kemme and G. Alonso. Don't be lazy, be consistent: a new way to implement database replication. *Proceedings of the 26th International Conference on Very Large Databases*, Vol. 26, pp. 134–143, 2000.

[20] R.R. Koch, S. Hortikar, L.E Moser, P.M. Melliar-Smith. Transparent tcp connection failover. *Proceedings of the International Conference on Dependable Systemsand Networks (DSN2003)*, pp. 383–392, 2003.

[21] Robert E. Shostak Leslie Lamport and Marshall C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382–401, 1982.

[22] Joshua MacDonald and Hans Reiser. *Reiser4 Transaction Design Document*. Namesys, 2001.

[23] Microsoft. Ms web application stress ツール, 2000.

[24] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. *Proceedings of the International Conference on Dependable Systemsand Networks (DSN2003)*, pp. 425–434, 2003.

[25] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). *Proceedings of the 1988 ACM SIGMOD*, pp. 109–116, 1988.

[26] Brian Randell. System structure for software fault tolerance. IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 220–232, 1975.

[27] George Reese. JDBC による Java データベースプログラミング. オライリー・ジャパン, 2001.

[28] Mikael Ronstrom and Lars Thalmann. Mysql cluster architecture overview. *MySQL Technical White Paper*, 2004.

[29] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *Proc. SOSP*, pp. 188–201, 2001.

[30] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, Vol. 2218, p. 329, 2001.

[31] David A Rusling. The linux kernel, 1999.

[32] D. Geels H. Weatherspoon B. Zhao S. Rhea, P. Eaton and J. Kubiatowicz. Pond: The oceanstore prototype. *Proc. FAST*, pp. 1–14, 2003.

[33] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 447–459, 1990.

[34] David Sceppa. プログラミング Microsoft ADO. NET. 日経 BP ソフトプレス, 2002.

[35] OASIS Standard. Web services reliable messaging tc ws-reliability 1.1, 2004.

[36] OASIS Standard. Web services reliable messaging (ws-reliable- messaging) version 1.1, 2007.

[37] Michael Stonebraker and Gerhard A. Schloss. Distributed raid— a new multiple copy algorithm. *Proceedings of 6th IDEC*, pp. 430–437, 1990.

[38] N. Suri, et al. Advances in ultra-dependable distributed systems. *IEEE Computer Society Press*, 1995.

[39] Yutaka Tanida and Atsushi Mitani. Pgcluster. http://pgcluster.projects.postgresql.org/, 2005.

[40] Wilfredo Torres-Pomales. *Software Fault Tolerance: A Tutorial.* NASA/TM-2000-210616, 2000.

[41] TPC. Standard secification revision 5.2. Technical report, Transaction Processing Performance Council(TPC) benchmark C, 2003.

[42] Jan van Bon, 水野康彦監訳. IT サービスマネージメント. itSMF, 2004.

[43] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transaction on Database Systems*, Vol. 16, No. 1, pp. 181–205, 1991.

[44] （財）日本規格協会. JIS Q 20000-1:2007 情報技術ーサービスマネジメントー第 1 部：仕様. 2007.

[45] 菅谷みどり. Linux ファイルシステム技術解説, 2003.

[46] 大野允裕, 加藤清志, 平池龍一. 自律運用管理に向けた障害対処ポリシの適用制御/流用手法 (高信頼システム, swopp 武雄 2005). 電子情報通信学会技術研究報告. DC, ディペンダブルコンピューティング, Vol. 105, No. 227, pp. 13–18, 20050729.

[47] 小野田英樹, 波多野賢治, 宮崎純, 植村俊亮. ウェアラブルコンピューティングのための追記型ファイルシステムの実装. 第 15 回データ工学ワークショップ予稿集, Vol. DEWS2004, pp. 1–A–02, 2004.

[48] 原田季栄, 保理江高志, 田中一男. プロセス実行履歴に基づくアクセスポリシー自動生成システム. *Network Security Forum 2003*, 2003.

[49] 中塚京子. Toc を応用した情報システム部門の生産性向上実現へのアプローチ. *ProVISION*, No. 33, pp. 90–99, 2002.

[50] 山西健司, 竹内純一. 統計的外れ値検出によるデータマイニングとネットワーク侵入検出への応用. 電子情報通信学会技術研究報告. IN, 情報ネットワーク, Vol. 102, No. 132, pp. 19–24, 2002.

[51] 大竹孝幸, 平野泰宏, 井上潮. 分類学習を用いた機能モジュールの自動選択法に関する検討. 情報処理学会研究報告（DBS-97）, Vol. 94, No. 30, pp. 11–19, 1994.

[52] 勝山恒男, 木村康則. 仮想自律化のためのシステム技術. *FUJITSU*, Vol. 56, No. 1, pp. 75–80, 2005/1.

[53] 熊本浩, 中嶋宏, 前川浩二. ファジィ推論を利用したsql診断システム. ファジィシステムシンポジウム講演論文集, Vol. 12, pp. 885–886, 1996.

[54] 甘利俊一, 麻生英樹, 津田宏治, 村田昇. パターン認識と学習の統計学—新しい概念と手法 統計科学のフロンティア6. 岩波書店, 2003.

[55] 種村昌之, 新城靖, 板野肯三, 千葉滋. ネットワークの監視によるバックアップシステムの実現. 情報処理学会シンポジウム論文集, Vol. 2001, No. 16, pp. 57–64, 2001.

[56] 総務省. 平成17年度版　情報通信白書. 2005.

[57] 総務省. 平成18年度版　情報通信白書. 2006.

[58] 石井達夫. pgpool : Connection pool server for postgresql. http://pgpool.projects.postgresql.org/, 2006.

[59] 石井達夫, 山田精一. オープンソース・ソフトによるシステム開発　postgresql構築・運用ガイド. 日経BP, pp. 132–209, 2003.

[60] 中村暢達, 加藤清志, 平池龍一. 耐障害に向けたサービスアプリケーション多重実行方式の提案. 情報処理学会第66回全国大会, pp. 5D–1, 2002.

[61] 当麻, 南谷, 藤原. フォールトトレーラントシステムの設計. 槇書店, 1991.

[62] 日本電気. 住宅業向け工事発注aspサービス プレスリリース, 2002.

[63] 松田芳樹, 玉田篤次, 鎌田義弘, 牧口邦治. システム運用管理の展望. 日立評論, Vol. 87, No. 4, pp. 339–344, 2005/4.

付録

A. TPC-C

本論文の信頼性向上の対象アプリケーションとして，TPC-C に準拠したベンチマークアプリケーションを使用した．本付録においては，実験に利用した TPC-C の詳細について説明する．

A.1 TPC-C とは

IT 用語辞典から引用すると，

> TPC とは、トランザクション処理システムの性能指標（ベンチマーク）テストの策定と検証とを目的とした業界団体のことで，正式には「トランザクション処理性能評議会，Transaction Processing Performance Council」のことである．TPC によって策定された性能指標には、テストの内容によって TPC-A,B,C,D,H,R,W の 7 種類に分かれる。中でも実際の物流業務におけるトランザクション処理に近い環境で行われる TPC-C が特に重視され、サーバ機器の性能を顕示するなどの目的でもっぱら用いられている。

とある．

2007 年 6 月の時点では，Revision 5.9 となっており，毎年２，３回のペースで，サービスの実状にあわせて更新がなされている．本実験で用いた IBM 社から提供されているツールキットは，2002 年にリリースされたものであり，2001 年にリリースされた Revision 5.0 に準拠しているものと思われる．

TPC-C には，データベースの論理設計が示されており，そのデータベース上の 5 つのトランザクションが定義されている．New-Order，Payment，Order-Status，Delivery，Stock-Level であり，本論文では，New-Order を用いた．

ベンチマーク評価のために，データベースのスケール，密度，想定するユーザの操作とユーザインタフェース，リカバリーログが必須であること，追加ソフトウェア，代替コンポーネントを用意すること等が規定されている．

A.2 トランザクション詳細

本論文で用いた TPC-C ツールキットの New-Order トランザクションは次の９種類 SQL 文を実行する．ただし，接続，コミットを除く．各パラメータ（?で示されている部分）は，ランダムな数値が与えられる．一トランザクションでは，下記１～５の SQL を１回と，６～９のトランザクションを 5 回繰り返す設定で実行した．

1．
```
SELECT c_discount, c_last, c_credit, w_tax FROM CUSTOMER,
WAREHOUSE  WHERE w_id=? AND c_w_id = w_id AND c_d_id = ?
AND c_id = ?
```

2．
```
SELECT d_next_o_id,d_tax FROM district WHERE d_w_id=? AND
d_id=?
```

3．
```
UPDATE district SET d_next_o_id=?+1 WHERE d_w_id=? AND d_id=?
```

4．
```
INSERT INTO ORDERS(o_id, o_d_id, o_w_id, o_c_id, o_entry_d,
o_ol_cnt, o_all_local VALUES(?,?,?,?,?,?,?)
```

5．
```
INSERT INTO NEW_ORDER(no_o_id, no_d_id, no_w_id) VALUES(?,?,?)
```

6．
```
SELECT i_price, i_name, i_data FROM item WHERE i_id = ?
```

7．
```
SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03,
s_dist_04, s_dist_05, s_dist_06, s_dist_07, s_dist_08,
s_dist_09, s_dist_10 FROM stock WHERE s_i_id=? AND s_w_id=?
UPDATE stock SET s_quantity=? WHERE s_i_id=? AND s_w_id=?
```

8．
```
UPDATE stock SET s_quantity=? WHERE s_i_id=? AND s_w_id=?
```

9. INSERT INTO order_line(ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) VALUES(?,?,?,?,?,?,?,?,?)

B. SQL 変換・検証の評価アプリケーション

評価対象となるデータベースアプリケーションとして，Web ショップにおける発注処理を想定したデータ処理を行う評価用アプリケーションを Java で作成した．評価用アプリケーションの処理を以下に示す.

1. ユーザ操作：ユーザは，商品発注として，発注者名：order_name（String），発注者住所：order_addr（String），性別：order_sex（int，1:男 2:女），そして発注する商品の商品番号：item_id（int，1〜1000） を入力する．括弧内はデータの型と範囲.

2. 取引開始：アプリケーションはデータベースに接続し，(3)〜(7) の一連のデータ処理からなるトランザクションを開始する.

3. 取引番号：データベースから順序オブジェクトを用いて，システム全体でユニークな取引番号：session_id（int）を取得する.

4. 在庫検索&処理：商品は，全部で 1000 種類あり，10 個の倉庫：warehouse01〜10 にそれぞれランダム数だけの在庫がある．すなわち，全部で１万件のデータが存在する．倉庫を warehouse01 から順に発注商品番号：item_id で検索し，在庫があれば，在庫数：stock を（int）１つ減らし，在庫のあった倉庫の番号：wharehouse（int，1〜10）を取得する．すべての倉庫を検索した結果，在庫が無い場合は取引を中止し，トランザクションをロールバックする．上記の処理を１つのストアドファンクションとしてデータベースに処理させる.

5. 発注日付：データベースから発注を行った日付：order_date（Date）を取得する.

6. 発送日付：データベースから発送する日付：send_date（Date）を取得する．発送日付は発注日付の１ヶ月後であり，データベースの日時関数を用いて計算する．

7. 発注記録：発注データとして，取引番号：session_id（int），発注日付：order_date（Date），発注者名：order_name（String），発注者住所：order_addr（String），性別：order_sex（String），商品番号：item_id（int，1〜1000），在庫倉庫番号：wharehouse（int，1〜10），発送日付：send_date（Date）を，発注記録：orders に追記する．なお，追記の際，性別：order_sex は 1 なら male，2 なら female に変換する．

8. 取引完了：(3)〜(7) の一連のデータ処理からなるトランザクションを完了し，データベースとの接続を閉じる．

著者業績

論文誌

1. 中村 暢達，藤山 健一郎，河合 栄治，砂原 秀樹，耐障害システムのためのデータベース接続層の拡張による柔軟な複製機構，情報処理学会論文誌，Vol.48, No.4, PP. 1562-1572，2007.04 掲載
2. 中村 暢達，里田 浩三，平池 龍一，根本 啓次，インターネット対応 3D マルチユーザシステム Ladakh，電子情報通信学会 論文誌 D (情報・システム)，Vol.81, No.5, PP.982-991，1998.05 掲載

国際発表

1. Nakamura, N.，Fujiyama, K.，Kawai, E.，Sunahara, H.，A Flexible Replication Mechanism with Extended Database Connection Layers，5th IEEE International Symposium on Network Computing and Applications，pp.212–219, Jul. 2006
2. Nakamura, N.，Hiraike, R.，Active Projector: Automated Augmented Reality Display Anywhere，ACM，User Interface Software & Technology (UIST)，2002.10
3. Nakamura, N.，Kennmochi, A.，Satoda, K.，Nemoto, K.，Tanaka, M.，A Virtual Skiing System for Interactive System for Interactive Training and Entertainment, ACM/SIGCHI Int Conf on Artificial Reality & Tele-Existence/Conf on Virtual Reality，1995.11
4. Nakamura, N.，Nemoto, K.，Virtual Skiing System Based on Distributed Processing and Network Communication，IEEE Int. Workshop on Networked Reality in Telecommunication (Networked Reality)，1995.10
5. Nakamura, N.，Nemoto, K.，Shinohara, K.，Distributed Virtual Environment System for Cooperative Work, IEEE Int Workshop on Networked Reality in Telecommunication (Networked Reality)，1994.05

国内発表

1. 中村 暢達，藤山 健一郎，河合 栄治，砂原 秀樹，柔軟な復旧要件を満たすデータベース接続層データ複製手法の提案，情報処理学会 第 17 回コンピュータシステム・シンポジウム Comsys2005， 2005.11

2. 中村 暢達，加藤 清志，平池 龍一，自律運用管理におけるサービス指向多重実行方式，並列/分散/協調処理に関するサマーワークショップ (SWoPP)，情報処理学会/電子情報通信学会，2004.07

3. 中村 暢達，加藤 清志，平池 龍一，耐障害に向けたサービスアプリケーション多重実行方式の提案，情報処理学会 全国大会，2004.03

4. 中村 暢達，平池 龍一，アクティブプロジェクタ: 凹凸面上を移動する映像の歪み補正，情報科学技術フォーラム (FIT)，情報処理学会/電子情報通信学会，2002.09

5. 中村 暢達，仁野 裕一，谷 幹也，市山 俊治，コンテンツ利用管理システム: モバイル RightsShell －システム概要－，情報処理学会 全国大会，2001.09

6. 中村 暢達，仁野 裕一，谷 幹也，市山 俊治，コンテンツ利用管理システム: モバイル RightsShell －コンテンツ配信方式－，情報処理学会 全国大会，2001.09

7. 中村 暢達，中尾 敏康，アクティブプロジェクター，ヒューマンインタフェースシンポジウム (HIS)，ヒューマンインタフェース学会，2000.09

8. 中村 暢達，國枝 和雄，プロジェクタ投影画像補正に関する一考察，日本バーチャルリアリティ学会 全国大会，1999.09

9. 中村 暢達，國枝 和雄，パノラマ画像を利用した三次元シーンデータの生成，電子情報通信学会 画像工学/パターン認識・メディア理解/マルチメディア・仮想環境基礎合同研究会，1999.07

10. 中村 暢達，共有仮想空間 Virtual-Playground の開発 －多人数参加型仮想空間における映像コミュニケーション手法－，仮想都市研究会シンポジウム，1999.02

11. 中村 暢達，平池 龍一，VRML 対応多人数参加型システムアーキテクチャの提案，情報処理学会 全国大会，1997.09

12. 中村 暢達，平池 龍一，サービス品質制御のためのＶＲＭＬ階層化方式，情報処理学会 全国大会，1996.09

13. 中村 暢達，根本 啓次，金子 朝男，篠原 克也，生体情報を利用した人工現実感スキーシステム (2)　―処理モジュール間の通信と制御―，情報処理学会 全国大会，1994.09

14. 中村 暢達，根本 啓次，篠原 克也，分散型仮想現実感システム構築環境の一検討，情報処理学会 全国大会，1994.03

15. 中村 暢達，ネットワーク対応仮想現実感による分散共同作業支援，情報処理学会 オーディオビジュアル複合情報処理研究グループ研究発表会，1993.09
16. 中村 暢達，篠原 克也，ネットワーク対応仮想現実感システムにおけるグループ通信制御機能，情報処理学会 全国大会，1993.03
17. 中村 暢達，篠原 克也，ネットワーク仮想現実感システムにおける通信制御方式，電子情報通信学会 秋季大会，1992.09

共著国際発表

1. Fujiyama, K.，Kida, K.，Nakamura, N.，Yanagisawa, M.，Takemura, T.，High-Performance Load Forecasting on Large-scale VM-type Thin Client System using Data Stream Processing Technology，Int Conf on Communication Systems, Networks & Applications (CSNA)，Int Assoc of Science & Technology for Development (IASTED)，Oct. 2007
2. Fujiyama, K.，Nakamura, N.，Hiraike, R.，Database Transaction Management for High-Availability Cluster System，IEEE Pacific Rim Int Symp on Dependable Computing (PRDC)，2006.12
3. Paul Schwartz，Lauren Bricker，Bruce Campbell，Tom Furness，Kori Inkpen，Lydia Matheson，Nobutatsu Nakamura，Li-Sheng Shen，Susan Tanny，Shihming Yen，Virtual Playground: Architectures for Shared Virtual World，ACM Symp on Virtual Reality Software & Technology，1998.11
4. Satoda, K.，Nakamura, N.，Kennmochi, A.，Nemoto, K.，A networked virtual skiing system -system overview-，ACM SIGGRAPH (Int Conf on Computer Graphics & Interactive Techniques)，1996.08
5. Shinohara, K.，Hiraike, R.，Hashimoto, M.，Nakamura, N.，A Virtual Reality System for Network Communications, Multi-party Co-operative Work in Real-time，Int Conf on Artificial Reality & Tele-Existence (ICAT)，1992.07

共著国内発表

1. 大野 允裕，加藤 清志，中村 暢達，自律運用管理に向けたルールベースによる設定情報管理手法の提案，並列/分散/協調処理に関するサマーワークショップ (SWoPP)，情報処理学会/電子情報通信学会，2007.08

2. 原 雅樹，水口 弘紀，中村 暢達，感性表現を用いたコンテンツ検索支援手法，マルチメディア、分散、協調とモバイル (DICOMO) シンポジウム，情報処理学会，2007.07

3. 喜田，藤山 健一郎，三津橋 晃丈，中村 暢達，次世代プローブ情報システム (2) ー大規模高速マップマッチングアルゴリズムの提案ー，マルチメディア、分散、協調とモバイル (DICOMO) シンポジウム，情報処理学会，2007.07

4. 三津橋 晃丈，藤山 健一郎，喜田 弘司，中村 暢達，次世代プローブ情報システム (1) ースケーラブルなプローブ情報の収集・配信アーキテクチャの提案ー，マルチメディア、分散、協調とモバイル (DICOMO) シンポジウム，情報処理学会，2007.07

5. 大野 允裕，加藤 清志，中村 暢達，業務サービスメトリックのロジスティック回帰分析に基づく運用管理モデルの提案，情報処理学会 全国大会，2007.03

6. 加藤 清志，大野 允裕，中村 暢達，自律運用管理に向けた設定情報管理手法に関する一考察，情報処理学会 全国大会，情報処理学会 全国大会，2007.03

7. 藤山 健一郎，中村 暢達，データベース接続層における SQL 変換による DBMS 置き換え支援方式の提案，情報処理学会 全国大会，2007.03

8. 池上 輝哉，加藤 清志，中村 暢達，自律運用管理における協調型ユーザインタフェース，ヒューマンインタフェースシンポジウム (HIS)，計測自動制御学会 (SICE)，2005.09

9. 藤山 健一郎，中村 暢達，平池 龍一，DB 接続 API 拡張による異種 DB 間データ同期複製方式，情報科学技術フォーラム (FIT)，情報処理学会/電子情報通信学会，2005.09

10. 藤山 健一郎，中村 暢達，平池 龍一，クラスタ対応高可用ＤＢトランザクション管理基盤，並列/分散/協調処理に関するサマーワークショップ (SWoPP)，情報処理学会/電子情報通信学会，2005.08

11. 藤山 健一郎，中村 暢達，平池 龍一，データベース接続 API 拡張によるデータ同期複製方式，ディペンダブルソフトウェアワークショップ，日本ソフトウェア科学会 (JSSST)，2005.01

12. 藤山 健一郎，中村 暢達，平池 龍一，データベース同期複製のための JavaAPI の拡張，情報処理学会 全国大会，2005.03

13. 池上 輝哉，加藤 清志，中村 暢達，平池 龍一，障害対処のためのシステム状態管理ユーザインタフェース，情報処理学会 全国大会，2005.03

14. 池上 輝哉，加藤 清志，中村 暢達，平池 龍一，システム運用管理における UNDO 操作ユーザインタフェース，情報処理学会 ヒューマンインタフェース研究会，2004.11

15. 藤山 健一郎，中村 暢達，平池 龍一，Java アプリケーション多重実行におけるデータベースアクセス管理方式，情報科学技術フォーラム (FIT)，情報処理学会/電子情報通信学会，2004.09

16. 加藤 清志，中村 暢達，平池 龍一，自律運用管理に向けたポリシー適用優先度の制御に関する一考察，情報処理学会 全国大会，2004.03

17. 仁野 裕一，中村 暢達，田口 大悟，谷 幹也，携帯電話向け Java 実行環境を用いた著作権管理システム，情報処理学会 電子化知的財産・社会基盤研究会 (EIP)，2003.03

18. 野田 潤，中村 暢達，田口 大悟，谷 幹也，セキュア P2P グループコミュニケーション基盤の提案，情報科学技術フォーラム (FIT)，情報処理学会/電子情報通信学会，2002.09

19. 仁野 裕一，中村 暢達，丸家 誠，カメラと液晶プロジェクタの簡易キャリブレーション手法，画像センシングシンポジウム，画像センシング技術研究会，2000.06

20. 里田 浩三，中村 暢達，平池 龍一，根本 啓次，インターネット上の仮想空間におけるマルチユーザコミュニケーション，仮想都市研究会シンポジウム，1998.03

21. 寺野 香織，剣持 聡久，中村 暢達，根本 啓次，バーチャルスキーシステムの滑走生成，電子情報通信学会 総合大会，1996.03

22. 仁野 裕一，中村 暢達，根本 啓次，生体情報を利用した操作インタフェース装置，電子情報通信学会 総合大会，1996.03

23. 阿部 豊子，前野 和俊，篠原 克也，阪田 史郎，福岡 秀幸，中村 暢達，マルチメディア分散会議システム MERMAID における分散協調機構とそのネットワーク VR への応用，情報処理学会 グループウェア研究グループ，1992.09

24. 篠原 克也，中村 暢達，平池 龍一，仮想現実感のネットワーク化，情報処理学会 マルチメディア通信と分散処理研究会 (DPS)，1992.05

25. 篠原 克也，中村 暢達，橋本 守，平池 龍一，仮想作業スペースにおける遠隔インタラクション手法，情報処理学会 全国大会，1991.10