社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－DBS－140 (II) (91)
2006／7／14

# PAXOS Consensus による OLTP 高可用化機構の提案とその実装

堀井 洋† 田井 秀樹† 山本 学†

## 概要

近年オンライントランザクション処理システムに対する高可用化の要求が増してきている. 従来, 多階層で構成されるシステムを高可用化するには, 各層個別の高可用化機構を連携させた複雑な構成をとる必要があり, また, 障害検知時間に依存したサービス停止時間も必要であった. 本論文では, オンライントランザクション処理システムを End-to-End で高可用化する機構の提案を行い, その実装と検証を記述する. 本機構では, クライアントは障害検知を待たず, 応答時間が遅くなった時点で同じトランザクション要求を副系のアプリケーションサーバに対して再送する. 正系, 副系のデータベースサーバは PAXOS コンセンサスアルゴリズムを利用してトランザクションログを複製するとともに, 重複したトランザクション要求のコミットを防ぐ. 我々は, 本提案手法を既存の Web アプリケーションに適用し, その評価を行った. その結果, 本システムが一般的な Web アプリケーションに適用可能であることを確認した.

# A Mechanism for Highly-Available OLTP Systems based on PAXOS Consensus

Hiroshi HORII†, Hideki TAI†, Gaku YAMAMOTO†

Demand for highly available (HA) on-line transaction processing system has been increasing these years. However, it is relatively difficult to make modern multi-layered systems HA because the HA mechanisms of each layer need to be coordinated carefully. In addition, service outage is dependent on failure detector which requires some amount of time until it detects a failure. We propose a mechanism that provides end-to-end HA for modern On-Line-Transaction-Processing (OLTP) systems. In our mechanism, when the primary system did not respond in a specific time, client reroutes a request to a secondary system without waiting for failure detection. Primary database server replicates transaction logs to secondary database server using PAXOS consensus algorithm. Primary and secondary database servers avoid duplicate transaction commits while they allow duplicate executions of a transaction by multiple application servers. We applied this mechanism to a Web application and evaluated it to show that it can be used for general Web applications.

## 1. はじめに

Web の普及により, オンライントランザクション処理システム(OLTP システム)が様々なサービスへと展開されてきている中, システムの高可用化は, 数々の社会問題になるほど, 重要な問題となってきている. 現在の Web を利用した OLTP システムは, 企業間取引や, 証券取引, 銀行業務など, 障害の発生が許されず, 障害が起きたとしても迅速な復旧が求められるものが多くなっている.

近年の OLTP システムは, 図 1 のような, Web サーバ, アプリケーションサーバ, データベースサーバの多階層からなるシステムから構築されていることが多い. Web サーバで静的な HTML の要求を処理し, アプリケーションサーバで動的な HTML の要求を処理する. そして, トランザクション処理の要求は, アプリケーションサーバからデータベースを利用するビジネスロジックを実行することで処理される. それぞれのサーバはクラスタリング構成をとることで, 大量の要求を処理することが可能となる.

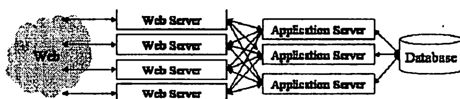


図 1: 一般的な OLTP システム

一般的に分散システムは, システムを多重化させ, 障害が発生したシステムの処理を正常なシステムが処理を継続(Take-over)することで, 高可用化を実現する[1]. 図 1 のようなシステムを高可用化するには, クラスタリングされているアプリケーションサーバ, データベースサーバ間で Take-over 処理を実現する必要がある. そのためには, (1)アプリケーションサーバ, データベースサーバの状態を複製し, (2)障害検出器が障害を検知後, (3)適切なサーバ構成に切り替わる必要がある. つまり, アプリケーションサーバはセッション情報, データベースはデータベース内のログなどを, Web リクエスト単位, トランザクション単位で複製する必要がある. また, ハートビート等を実行し, サーバの状態を監視して, 障害の発生を正確に検知する必要がある. そして, 障害検知後はシステム全体が適切な構成に切り替わる必要がある.

しかし, 図 1 のようなシステムを高可用化するためには, 下記のような問題が存在する.

- 正確な障害検知のためには, 一定の時間を要する
- システムごとに全障害検知後の設定が必要となる
- 完全に障害をマスクするためには, アプリケーションロジック内でも対応することが必要となる

一般的に障害検知は, 一時的に高負荷な状態でも誤検知しないように, 意図的に時間をかけて行う. また, 各サーバ上で誤った障害検知を行い, 同時に2つのデータベースサーバがプライマリとして動作してしまうこと(Split Brain Syndrome)を避けるため. 障害検知はシステム全体で協調する実行される必要がある. このような要件のために, 障害検知には 10 秒以上の時間が必要となる事が多い.

また, 障害検知後の設定は, システムごとに異なることが多い. 例えば, データベースの障害発生後, 障害検知機構から情報を取得し, アプリケーションサーバのデータベースに関する設定を変更するスクリプトが必要となる. このようなスクリプトは, システム構成ごとに記述する必要がある.

トランザクションの実行終了後, その終了の通知を他の層に転送中に転送しているサーバが障害を起こした場合, 他の層のサーバはトランザクションが成功しているのか, 失敗しているのか判断することができない. 2Phase-Commit を利用せずにこのよ

† 日本 IBM 東京基礎研究所
IBM Research, Tokyo Research Laboratory

うな問題を解決するためには，アプリケーションロジック内で，障害発生後のデータベースの状態を確認し，未解決のトランザクションが実行済みかどうかを確認する処理を行わなくてはならない．

本稿では，完全な障害検知に頼らず，かつ，多階層のシステム構成の高可用化にも適用できる Failure Free Facility for Replication(FFF Replication)を提案する．FFF Replication は，(1) At most once のトランザクション実行，(2)高速な Take-over を実現する．

FFF Replication のクライアントは，トランザクション結果の返答がない場合，同じトランザクション ID(*tid*)でトランザクションの実行要求を送信する．また，システムは，実行済みの *tid* でトランザクションの実行要求を受信した場合，保存されたトランザクションの実行結果を返す．もし実行されていなかった場合，トランザクションの実行を開始し，実行結果を保存する．

また，FFF Replication は，不確実な障害検知器[3] 上で投機的にプライマリを決定し，トランザクションの実行を行う．そして，トランザクションのコミット要求処理の際，(1)重複トランザクションの実行を避け，(2)ACID を満たしているか確認し，(3)暫定的な LSN を割り当て，PAXOS コンセンサスアルゴリズムを利用して LSN に対するコミットログの合意を得ることで，ACID を満たし，かつ，高速な Take-over も実現する．

我々は，FFF Replication の実装として，Failure Free Facility を実装した．FFF は，J2EE の HttpServlet の実行を高可用化トランザクションとして実行することが可能なフレームワークである．FFF が提供する HTTP Proxy は，Web サーバあるいはアプリケーションサーバのへのリクエストを，返答が得られるまで自動的に再送し続ける．また，HttpServlet の実行の前後で，FFF が提供する ServletFilter がトランザクションの実行を開始，確定要求を行う．アプリケーションロジックはサーブレット内にトランザクション処理を記述するのみで，全てのサーブレットの実行が，高可用化される．本稿では，FFF のフレームワーク上で TPC-W の実装を行い，その性能評価を行った．

これより，2 において，本稿が対象とするシステム構成と，そのシステム構成が高可用化のために満たすべき条件を示す．次に 3 において，既存の高可用化手法を示し，4 において提案手法で利用する PAXOS コンセンサスアルゴリズム，5 において提案する手法を示す．そして，6 において提案手法の実装である FFF の構成について示し，7 において FFF 上で TPC-W を実行させた際の評価を示す．8 に本提案手法，実装に関する議論を行い，9 に関連研究の紹介，最後にまとめを行う．

## 2. 耐障害性 OLTP システム

本節では，本稿で用いる耐障害性 OLTP システムモデルを示し，耐障害性 OLTP システムが満たすべき性質を挙げる．

### 2.1. 耐障害性 OLTP システムモデル

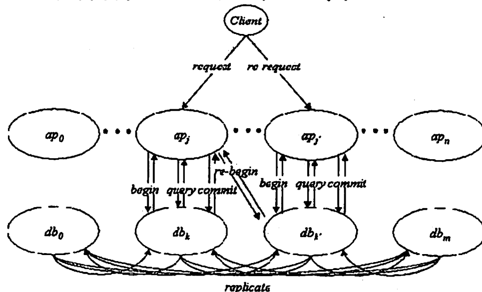


図 2: OLTP システムのモデル

本稿で用いる耐障害性 OLTP システムモデルは，2 回以上同一トランザクションを実行しないことを保証するシステムと，システムからトランザクション結果の返答が遅い場合，何度もトランザクションの要求を行うクライアントから構成される，耐障害性を保証する OLTP システムモデルである．想定する OLTP システムモデルを図 2 に示す．

システムを構成するプロセスは，複数のクライアントプロセス *client* と，*n* 個のアプリケーションプロセス *AP*，*m* 個のデータベースプロセス *DB* である．複数の *client* が *AP* 内のプロセスに複数のトランザクション *T* の実行を要求し，*AP* と *DB* 内のプロセスでトランザクションを実行する．*client* から要求される $t_i \in T$ は，1 つのアプリケーションプロセス $ap_j \in AP$ に要求され，$ap_j$ は 1 つのデータベースプロセス $db_k \in DB$ をプライマリとして $t_i$ を実行する．*DB* 内の全てのプロセスは Take-over 可能なプロセスであり，コミットされた全ての $t_i$ の $log_i \in LOG$ が複製され，データベースに更新する．

*client* は *AP* のプロセスに対して，*request*, *re-request*, *AP* のプロセスは *DB* のプロセスに対して *begin*, *re-begin*, *query*, *commit*, *DB* のプロセスは他の *DB* のプロセスに対して *replicate* のメッセージを送る．なお，それぞれのメッセージは，トランザクション $t_i \in T$ を処理するため，下記のように送受信する．

| | |
|---|---|
| *step 1* | *client* が $ap_j \in AP$ にトランザクションの実行を要求する(*request*)． |
| *step 2* | $ap_j$ は $db_k \in DB$ に対し，$t_i$ の開始を要求する(*begin*)． |
| *step 3* | $db_k$ は *begin* の返答を行う． |
| *step 4* | $ap_j$ はアプリケーションロジックを実行する．ロジック中でクエリを要求する場合は，*step 5* へ，終了した場合は，*step 7* に進む． |
| *step 5* | $ap_j$ はアプリケーションロジック中でクエリを $db_k$ に要求する(*query*)． |
| *step 6* | $db_k$ は要求された *query* の実行結果を $ap_j$ に返す．*step 4* に進む． |
| *step 7* | $ap_j$ は，$db_k$ に対し $t_i$ の確定を要求する(*commit*)． |
| *step 8* | $db_k$ は，$ap_j$ が $t_i$ の処理中に生成した更新ログ $log_i \in LOG$ を $\{DB\text{-}db_k\}$ に複製の要求をする(*replicate*)． |
| *step 9* | *replicate* の終了後，$t_i$ の確定結果を $ap_j$ に返す． |
| *step 10* | *replicate* された $\{DB\text{-}db_k\}$ のプロセスは，$log_i$ を *update* する． |
| *step 11* | $t_i$ の確定結果を返された $ap_j$ は $t_i$ の実行結果を *client* に返す． |

*client* は，*step 1* の後，$ap_{j'}$ に，$t_i$ の実行を再度要求し(*re-request*)，*step 2* を実行することが可能である．また，$ap_j$ は，*step 2* から *step 9* の間の任意のタイミングで，$db_{k'}$ に対して *step 2* を再実行すること(*re-begin*)することが可能である．

### 2.2. 耐障害性 OLTP システムの性質

図 2 で示した耐障害性 OLTP 処理システムは，Atomicity, Consistency, Isolation, Durability (ACID)を保障するため，下記の性質を満たさなくてはならない．

| | |
|---|---|
| *Property 1* | 全ての $t_i \in T$ に関し，$db_k \in DB$ は $t_i$ の更新ログを 2 回以上 *update* していない |
| *Property 2* | 全ての $t_i \in T$ に関し，$t_i$ の更新ログとして DB に update されているログ $log_{ij}$ は、全ての $db_k \in DB$ に渡って同一である |
| *Property 3* | 全ての $db_k \in DB$ は，同じ順番で *LOG* を *update* している[1] |
| *Property 4* | 全ての *update* された $log_i \in$ *LOG* を生成した $t_i$ の *begin* から *update* の間に，$t_i$ を処理する $db_k$ 以外の *DB* 内のプロセスが生成した $log_{i'}$ が *update* されてい |

[1] 依存のないトランザクションは，異なるデータベース上で異なる順番で処理してもデータベースが不整合にならないが，本稿ではより保守的な制約を持たせるものとする．

ない[1]

*Property 5* *client* が同一の $t_i$ の処理を要求した場合，返される結果は常に同じである

## 3. 将来的完全の障害検知器を利用した耐障害性 OLTP システム

2.1 で示した処理モデルの他に，将来的完全(Eventually Perfect)の障害検知器[3] を用いた耐障害性 OLTP システムを示す．

本手法は，将来的完全の障害検知器で *AP*, *DB* のプロセスを監視することで，常に 1 つ以下のプライマリデータベースプロセスを選出して行うシステムである．本手法は，既存の製品等を利用して構築可能なシステムであり，実際のシステムに広く用いられる手法である．

### 3.1. 障害に対する対処方法

障害検知後，システムは要求された処理の他に，下記のように，$ap_j$ , $db_k$ の障害に対する $t_i$ の Take-over 処理を実行することで，2.2 の性質を満たす．

**$ap_j$ 障害に対する手法**

*ap-step 1* $ap_j$ の障害を検知する

*ap-step 2* *client* が $ap_{j'}$ に，*re-request* を行う

*ap-step 3* $ap_{j'}$ のロジック内で，$t_i$ がすでに *commit* しているか判断し，*commit* 済みの場合は *client* に結果を返す．*commit* していない場合は，*re-begin* を行う

**プライマリの $db_k$ 障害に対する手法**

*db-step 1* $db_k$ の障害を検知する

*db-step 2* $db_k$ を Take-over する $db_{k'}$ を決定する

*db-step 3* 全ての *AP* に対してプライマリが $db_k$ から $db_{k'}$ に変更したことを通知する

*db-step 4* 全ての $ap_j \in AP$ が，実行中のトランザクションをロールバックし，$db_{k'}$ に対し *re-begin* を行う

### 3.2. プライマリの選出方法

本手法は，プライマリの $db_k$ の障害を検知した後に，Take-over を行うデータベースプロセスを選出する．選出は，全てのデータベースプロセスに優先順位をつけることで，最も高い優先度のデータベースプロセスを次のプライマリとして選出する方法や，[3] のように，ラウンドロビンでプライマリを選択する方法等がある．

### 3.3. *replicate* の手法

*replicate* は，Atomic Broadcast を用いて行う．

### 3.4. システムの性質

上記で示したシステムは，2.2 で示した性質を全て満たす．

全ての $t_i$ の処理は，$t_i$ を実行している $ap_{j'}$，または，$db_k$ の障害が起こるまで，Take-over が行われることはないため，1 度以上同一のトランザクションが実行されることはない(*Property 1*)．*replicate* も，*Atomic Broadcast* を用いて行うため，全ての $db_k$ は，全ての $t_i \in T$ に対し，同一の $log_i$ を，同じ順序で，1 回ずつ *update* する(*Property 2Property 1, Property 3*)．$db_k$ の障害が起きた場合，実行中のトランザクションは全てロールバックするため，*begin* から *update* の間に，$t_i$ を処理する $db_k$ 以外の *DB* 内のプロセスが生成した全ての $log_i$ を *commit* することはない(*Property 4*)．また，障害時に *re-request*, *re-begin* を行うことで，障害が起こった際に実行不能となった全てのトランザクションの Take-over が実現される．つまり，*AP*, *DB* が全面障害にならない限り，*DB* に記録したデータを基に，毎回同一のトランザクション結果が *client* に返されることとなる(*Property 5*)．

### 3.5. 問題点

上記のシステムは，下記のような問題点が存在する．

- 正確な障害検知のためには，一定の時間を要する
- システムごとに全障害検知後の設定が必要となる
- 完全に障害をマスクするためには，アプリケーションロジック内でも対応することが必要となる

将来的完全の障害検知器を用いた耐障害性 OLTP システムは，障害発生時，障害検知器が障害を検知後しか Take-over 処理を行うことはない．1 で示したように，分散システムにおける障害検知は，10 秒から 20 秒程度の時間をかけることが多い．

*DB* 内のプライマリプロセスに障害が起きた場合，システムは必ず 1 つのプロセスをプライマリとし，Take-over 処理を行う．その際，次のプライマリプロセスを決定する処理や，全ての *AP* に対してプライマリの変更を通知する処理は，システム構成により *DB*, *AP* が異なるため，システム固有の設定となる．

また，*ap-step 3* や，*db-step 4* のように，障害をマスクするためには，アプリケーションロジック内で障害対策用の処理を行う必要がある．

## 4. PAXOS コンセンサスアルゴリズム

本節では，我々が提案する，FFF Replication で利用する，PAXOS コンセンサスアルゴリズム[5] と，その性質を示す．

### 4.1. アルゴリズム

コンセンサスアルゴリズムとは，それぞれのプロセス *Pi* が値 *Vi* を提案し合い，それぞれのプロセスが値を決定していくアルゴリズムである．コンセンサスアルゴリズムは，下記の条件を満たさなくてはならない．

- 停止性: 全ての正常なプロセスが 1 つの値を決定し，停止する．
- 合意: 全ての正常なプロセス間で，同じ値に決定する．
- 妥当性: 決定した値を提案したプロセスが存在する．

非同期システムにおいて，プロセスが 1 つ故障した場合，上記の性質を満たすアルゴリズムは存在しないことが示されている[6]．しかし，非同期システムにおいて，停止性以外の性質を満たし，不確実な障害検知器[3] を用いることで，停止性を保障することが可能なコンセンサスアルゴリズムが提案されている．PAXOS アルゴリズムも，それらの中の 1 つのアルゴリズムである．

PAXOS アルゴリズムは，それぞれのプロセスがラウンドと推定値，提案値を管理し，通信しあうことでコンセンサスを得る．それぞれのプロセスはコンセンサスを得る際，下記のような処理を行う．

(1) Info-Quorum の構築

(1.1) 1 つのプロセス(調停者プロセス)がコンセンサスの準備要求を，全プロセスに送る．その際，自身が知っている最大のラウンドに 1 を加えたラウンドを付与する．

(1.2) 準備要求を受け取ったプロセスは，受け取ったラウンドが自身のラウンドよりも大きい値の場合は，準備の承諾を通知する．承諾の通知には，最後に受理したラウンド数と推定値を付与する．もし小さい値の場合は，未承諾の通知を行う．

(1.3) 調停者プロセスが，過半数以上の承諾の通知を得たら，Info-Quorum の構築を終了する．なお，1 つでも未承諾の通知を受信した場合は，このラウンドの処理を終了し，再度(1.1)に戻る．また，取得した提案値が全て初期値である場合は，自身の推定値を提案値とする．そうでない場合は，最大のラウンド数を返したプロセスの推定値を提案値とする．

(2) Accepting-Quorum の構築

(2.1) 調停者プロセスが，提案値とラウンドを全プロセスに通知する

(2.2) 提案値を受信したプロセスは，受信したラウンドよりも高いラウンドをすでに受理しているのならば，未承諾の通知を行う．そうでない場合は，推定値を提案値に更新し，承諾の通知を行う．

(2.3) 調停者プロセスが，全プロセスのうち過半数以上の承諾の通知を得たら，Accepting-Quorum の構築を終了する．

(3)　結果の通知

(3.1)　提案値を全プロセスに通知する

過半数以上のプロセスが, Accepting-Quorum の構築を受理したが, 提案しているプロセスがまだその承諾通知を得ていない場合でも, 新たなラウンドで Info-Quorum が組まれる可能性がある. しかし, Info-Quorum で決定される提案値は, 構築された Accepting-Quorum の提案値であるため, 新たに Accepting-Quorum 組まれたとしても異なる提案値を決定することはない.

## 4.2. アルゴリズムの性質

PAXOS コンセンサスアルゴリズムは, 過半数以上の crush failure[1] が起こらない限り, コンセンサスを得ることが可能である. また, PAXOS コンセンサスアルゴリズムは, 将来的弱(Eventually Week)以上の障害検知器を仮定し, 調停者プロセスを選出することを前提としている. 複数のプロセスが調停者プロセスとなり, Info-Quorum の構築を続けた場合, ラウンドの更新が停止することがなく, Accepting-Quorum を構築が終了しない.

## 4.3. MULTIPAXOS

複数の値に対してコンセンサスを得る場合, Fisher等は1つの Info-Quorum で得たラウンドを, 複数の Accepting-Quorum に利用する手法を MULTIPAXOS アルゴリズムとし, 4.1 で示したアルゴリズム[7]. MUTLIPAXOS を利用することで, Info-Quorum を組まずに複数の値に対してコンセンサスを得ることができるため, 全体としてメッセージ数を減らすことが可能となる.

# 5. FFF Replication

本稿では, 障害検知を必要せず, かつ, 多階層で構成されるシステムの高可用化を想定したプロセス構成にも容易に適用できる FFF Replication を提案する.

FFF Replication は, 将来的弱(Eventually Weak)以上の障害検知器上で不確実なプライマリのデータベースを選択し, トランザクションを実行する. コミット要求されたトランザクションのコミットログは, 不確実なプライマリのデータベースにて *lsn* が割り当てられ, *lsn* に対するコミットログとして *DB* のプロセス内で分散コンセンサスを得る.

## 5.1. FFF Replication の処理

FFF Replication は, *DB* の階層で不確実な障害検出器の存在を仮定する. 本提案手法では, PAXOS コンセンサスアルゴリズムを拡張しているため, 不確実な障害検出器は将来的弱のクラス以上を想定する. 本提案手法は, 2.1 で示した処理に加え, 下記の処理を行う.

### 5.1.1. *tid* の利用

全ての $t_i$ に関する処理要求に対し, $tid_i \in TID$ を付与して行う. また, 全ての $db_k$ は *commit* が完了した $tid_i$ $log_i$ を保存する(*cTID*, *cLOG*).

### 5.1.2. *re-request*, *re-begin* の処理

*client* は, $t_i$ の実行結果が一定時間返ってこなかった場合, $ap_j$ に対し $t_i$ の *re-request* を要求する. また, $ap_j$ も, $t_i$ の *begin*, *query*, *commit* の実行結果が一定時間返ってこなかった場合, *re-begin* を要求する.

### 5.1.3. *begin* の処理

$db_k$ は, $ap_j$ から $tid_i$ のトランザクション $t_i$ の *begin* 要求を受ける際(*step 2*), 下記の処理を行う.

**コミット済みトランザクションの判別**

$tid_i$ が *cTID* に含まれるか判別し, 含まれる場合は $log_i \in cLOG$ を返す(*committed*).

**プライマリになれるか判別**

下記の条件を満たす際, $db_k$ はプライマリになれるか判断する.

- すでにプライマリである場合
- 前回プライマリであった $db_k$ が不確実な障害検出器で障害と判別された場合
- 前回プライマリであった $db_k$ からの *replicate* が一定時間発行されていない場合

上記の条件を満たす場合, $ap_j$ に対し *ack* を返す. それ以外の場合は, $ap_j$ に現在のプライマリ $db_k$ を通知する(*navigate*).

***begin* 時点で最後に *replicate* された *lsn* を記録**

上記の処理により, *begin* 可能と判別された $t_i$ に対し, *replicate* された *lsn* を記録する($begin\text{-}lsn_i$).

### 5.1.4. *commit* の処理

$db_k$ は, $ap_j$ から $tid_i$ のトランザクション $t_i$ の *commit* 要求を受ける際, 下記の処理を行う.

***commit* 可能か判別**

*commit* 要求を受けた時点で最後に *update* した *lsn* を *commit-lsn* とする. $begin\text{-}lsn_i$ から *commit-lsn* 間に *update* した *cLog* の内, $db_k$ 以外が *replicate* した *log* が存在していた場合, *commit* 不可能と判別する. *commit* 不可能と判断した場合, $t_i$ をロールバックし最新のプライマリ $db_k$ を通知する(*navigate*).

### 5.1.5. *replicate* の処理

$db_k$ が $log_i$ の *replicate* を要求する際, 下記の処理を行う.

**最新 LSN に対する合意**

PAXOS コンセンサスアルゴリズムを利用して, 最新 $log_i$ (*commit-lsn+1*)に対する $log_i$ の合意を得る.

**合意が得られた際の処理**

全ての *DB* 内のプロセスは, $tid_i$ を *cTID* に, $log_i$ を *cLog* に追加する.

**合意が得られなかった際の処理**

$log_i$ の合意が得られなかった場合, $log_i$ を生成した $db_k$ は, $lsn_i$ より小さい *begin-lsn* のトランザクションを全てロールバックする. また, 合意を得た *log* を生成した $db_k$ をプライマリとして通知する(*navigate*).

### 5.1.6. *committed*, *navigate* の処理

$db_k$ に対し $t_i$ の *begin*, *commit* を要求した $ap_j$ は, $db_k$ での要求した処理に失敗すると, *committed* もしくは *navigate* の通知を受ける. *committed* の通知を受信した場合は, $ap_j$ はメッセージ内のから $log_i$ を取得し, $t_i$ の処理結果を *client* に返す. また, *navigate* の通知を受信した場合, メッセージ内からを現在のプライマリ $db_{k'}$ を取得し, 再度 *begin* を実行する.

## 5.2. 耐障害性 OLTP システムの性質

上記で示したシステムは, 2.2 で示した性質を全て満たす. 全ての $db_k$ は, 最新の *lsn* に関して合意を得た後, *commit*, もしくは, *replicate* を合意後に処理するため, 全ての *DB* 内のプロセスは, 同じ順番で同一の *log* を *commit* する(*Property 2*, *Property 3*). また, *begin-lsn*, *commit-lsn* 間に他の DB が生成した *log* が *update* されないことも保障する(*Property 4*). そして, 全ての $db_k$ は $t_i$ を *commit* 時に *cTID* に $tid_i$ が含まれていないか判別するため, 同じトランザクションを 2 回以上 *commit* することはない(*Property 1*), 実行済みトランザクション結果を *cLOG* に保管することにより, *client* が要求した同一の *tid* のトランザクション処理要求は, 必ず実行される(*Property 5*).

## 5.3. FFF Replication の実行例

### 5.3.1. 正常時の実行

障害が発生しなかった場合の, FFF Replication を用いた1つのトランザクションの実行の流れを, 図 3 に示す.

図 3 は, *client* が $ap_0$ にトランザクションの実行を要求し, $ap_0$ が *begin*, *query* を $db_0$ に要求後, *commit* を要求した状況を示している. *commit* を要求された $db_0$ は, 4.1 で示したように, (1) Info-Quorum の構築, (2)Accepting-Quorum の構築を行い, (3)success メッセージの送信を行う. しかし(3)の処理前にすでに *lsn* は確定しているので, *commit* 完了を $ap_0$ に通知する.

図 3 に示すとおり, $db_0$ は *commit* 時, 4 ステップ, (m-1) x 4 のメッセージ数で $ap_0$ にコミット結果を返すことが可能である. しかし, 最新 *lsn* が確定しない限り, 次の *lsn* の合意を得ることができないため, 2 つのトランザクションが *commit* を要求した場合, 同時にコンセンサスを得ることができない. *lsn* の合意に関する最適化に

関しては，5.4 にて議論する．

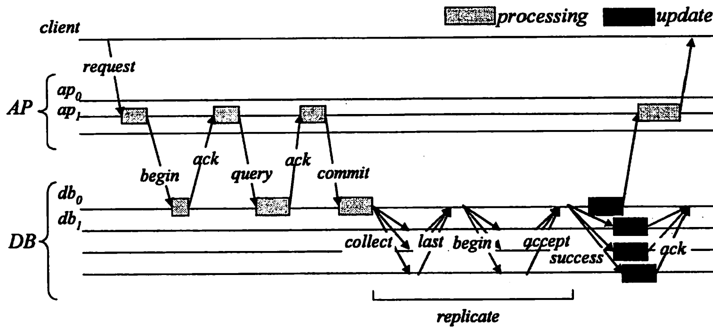


図 3: FFF Replication による正常時の実行

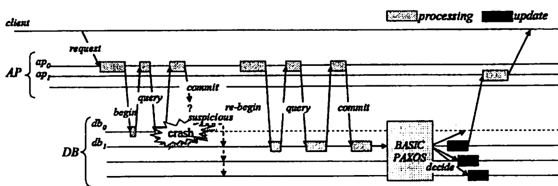


図 4: $db_0$の障害時の処理

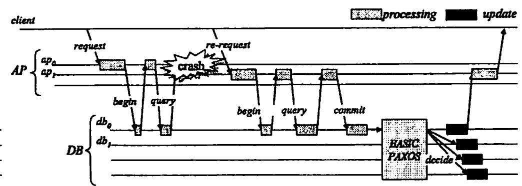


図 5: $ap_0$の障害時の処理

### 5.3.2. $db_k$障害時の実行

プライマリであった $db_0$ に障害が発生した場合の，FFF Replication を用いた 1 つのトランザクションの実行の流れを，図 4 に示す．

図 4 では 5.3.1 と同様，$ap_0$ が $db_0$ に対し，*begin*, *query*, *commit* を発行することで，トランザクションの実行を行い，*commit* 時に $db_0$で障害が起きた状況を示している．このような場合，$ap_0$ は *commit* を要求後，一定時間返答がないため，*re-begin* の要求を $db_1$に対して試みる．$db_1$は，*re-begin* を受信後，プライマリとなれるか判断し，もしプライマリになれる場合は *ack* を返す．その後は，通常通り，$db_1$ をプライマリとして，トランザクションの処理を行う．

*re-begin* されたトランザクションの *commit* を発行する時点で $db_0$ が復旧し，以前 *commit* 要求を受けた際に生成した *log* を *replicate* を試みる場合，自身がプライマリと判断している $db_0$と，$db_1$ が同じ *lsn* に関して合意を試行することがある．その際は，PAXOS コンセンサスを用いることで，1 つの log に決定される．

### 5.3.3. $ap_j$障害時の処理

*request* された $ap_0$ に障害が発生した場合の，FFF Replication を用いた 1 つのトランザクションの実行の流れを，図 5 に示す．

図 5 では，5.3.1 と同様，$ap_0$ が $db_0$ に対し，*begin*, *query*, *commit* を発行することで，トランザクションの実行を行い，*commit* を要求する前に，$ap_0$で障害が起きた状況を示している．このような場合，*client* は *request* を要求後，一定時間返答がないため，*re-request* を $ap_1$ に対し行う．その後，通常通り，$ap_1$ と $db_0$間でトランザクションの処理を行う．

$ap_0$が *commit* をすでに $db_0$に発行していて，*commit* されてしまっていた場合，$ap_1$ で実行されるトランザクションは，*begin*, もしくは *commit* の要求時に，*committed* を返されることになる．

## 5.4. 最適化

上記に示した FFF Replication の最適化手法について示す．

### 5.4.1. *lsn* の順番付けの最適化

5.3.1 に示したように，本手法は，直前の *lsn* が解決しない限り，次の *lsn* の合意を試行することができない．しかし，下記の手法を用いることで，最適化を行うことが可能である．

- 複数の *log* に 1 つの *lsn* を割り当てる
- 直前の *lsn* の合意の Info-Quorum で利用したラウンドを次の合意でも利用する

複数の *log* に 1 つの *lsn* を割り当てることで，スループットの向上が可能である．直前の *lsn* の解決に時間がかかる場合，*replicate* 待ちの *log* が大量にたまることになる．これらの *replicate* 待ちの *log* の集合は，原子的に処理する必要があるため，1 つの *lsn* を割り当て，1 回の合意で *replicate* することが可能である．なお，*replicate* を受信したプロセスは，必ずそれぞれの *log* が生成された順番で，*update* していく必要がある．

また，4.1 で示したとおり，MULTIPAXOS を用いることで，直前の合意で利用したラウンドを，そのまま次の Accepting-Quorum の構築で利用することが可能である[1]．前回の合意で利用したラウンドを共有することで，メッセージが 2 ステップ減少することとなり，レスポンス時間を短くすることが可能である．

### 5.4.2. *client* の *re-request*, *AP* の *re-begin*

*request* から *re-request*, *begin* から *re-begin* の間隔は，障害発生時，$t_i$を実行する *AP* のプロセス，もしくは，*DB* のプロセスの切り替え時間を意味する．つまり，障害発生時，$t_i$ が通常よりこの間隔分だけ遅く実行される．しかし，間隔を短くした場合，必要以上のトランザクション処理をシステムに要求することになるため，全体としてスループットが遅くなる可能性がある．

### 5.4.3. 3 つの *DB* プロセス

分散合意を行う *DB* のプロセス数が 3 の場合，PAXOS コンセンサスアルゴリズムの *success* メッセージを一部削除することが可能となる．Accepting-Quorum を構築する際，リーダー以外のプロセスは 2 つのプロセスしか存在しない．つまり，1 つでも合意すれば，必ず合意が得られることを意味する．すなわち，Accepting-Quorum で合意したリーダー以外のプロセスは，*replicate* されてきたログが確定されることを認識することができ，

*success* のメッセージを待たずに *update* することが可能である．このような処理を行うことで，障害発生時に *update* する時間を短縮することができ，切り替え時間の短縮を行うことが可能である．

#### 5.4.4. *navigate* の高機能

*AP* のプロセスは，*navigate* を受けた際，その *navigate* 先のプロセスが現在のプライマリであることを認識することが可能である．つまり，最後に *navigate* されたプロセスを記録しておき，次回以降のトランザクションは，記録したプロセスに *begin* することで，*navigate* される割合を下げることが可能となる．*navigate* される割合を下げることは，トランザクションのレスポンス時間を短縮させ，かつ，無駄な実行を減少させる効果がある．

#### 5.4.5. *begin* の実行条件

プライマリが障害と疑われた際に，5.1.3 に示したプライマリとなるための条件に加え，プライマリ以外の通信可能なプロセス内で 1 つのプロセスを次のプライマリと仮決定することで，複数のプロセスが次のプライマリとして起動することを避けることが可能である．複数のプライマリの乱立を避けることは，PAXOS コンセンサスアルゴリズムが停止するまでの時間が短くなることを示しおり，障害時のレスポンス時間の短縮が実現可能となる．

## 6. Failure Free Facility

我々は，5 に示した手法を実現する，Failure Free Facility FFF を実装した．FFF は，*update*, *replicate* の処理を実現する Coordinator, Session と様々なリクエストを高可用化するコンポーネントから構成される．なお，本稿では HTTP リクエストを高可用化するための，Proxy, ServletFilter について示す．

### 6.1. サーバ構成

FFF のサーバ構成を図 6 に示す．

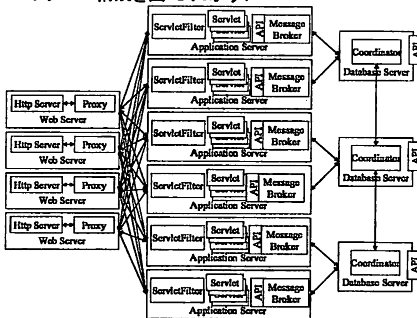


図 6: FFF を利用した Servlet フレームワークの概要

FFF のサーバ構成は，図 1 で示すシステムと同様，Web サーバ層，アプリケーションサーバ層，データベースサーバ層の 3 つの層から構成される．Web サーバは，HTTP サーバと Proxy から，アプリケーションサーバは，Servlet, ServletFilter の J2EE サーバコンポーネントに加え，Coordinator と通信する Message Broker から，そして，データベースサーバは，Coordinator から構成される．なお，データベースサーバは，5.4.3 で示した最適化のため，3 つのデータベースサーバから構成されている．また *AP* による *re-begin* 機能はなく，各 $ap_j$ は 1 つの特定の $db_k$ にのみ通信を行うことができる．

データベースへのアクセスは，Coordinator の API を利用することで行う．また，データベースへのアクセスのための API を利用する場合は，ServletFilter のサブクラスと，Message Broker の API を利用することで，様々なデータアクセスフレームワークを利用することが可能である．

### 6.2. コンポーネントの機能

以下に，6.1 で示したコンポーネントの機能を示す．

**HTTP サーバ**

HTTP Server は，HTTP リクエストを受信し，リクエストされたリソースを返す．トランザクションを実行する URL が指定された場合は Proxy に転送する．

**Proxy**

HTTP Server から転送されてくる HTTP Request に対し，*tid* を HTTP ヘッダーに追加し，*request*, *re-request* の機能を実装する．

**Servlet Filter**

Proxy から受信する HTTP Request から *tid* を抽出し，Message Broker に対して *begin* 送信を要求する．*begin* 成功後，servlet の実行を行う．そして，Message Broker に対し，servlet が生成した HTML と Session オブジェクト[3]の *commit* 送信を要求する．

**Message Broker**

Servlet Filter から *begin*, *commit* の要求を，Coordinator に送信する．また Coordinator が障害時，適切なメッセージを Proxy に伝える．Message Broker には API が存在し，API を通じて Coordinator が公開する API にメッセージを送信することが可能である．

**Coordinator**

*begin*, *commit*, *replicate* の要求を受け，*begin*, commit の実行が可能かを判断や，*replicate* 時の分散コンセンサスに対する処理を行う．また，*begin*, *commit* のイベントの通知や *update* の要求を API の実装に行う．

## 7. 性能評価

6 で示した FFF/TDR のトランザクショナルな Servlet フレームワークにおける障害発生時の性能評価を，ベンチマークアプリケーション TPC-W を用いて行う．

### 7.1. TPC-W

TPC は，トランザクション処理やデータベースに観測するベンチマークで，TPC-W[8] は Web ベースのトランザクション処理を対象にしたベンチマークである．TPC-W を利用することで，データベースだけでなく，アプリケーションサーバ，Web サーバを含めた性能値が測定することが可能である．

TPC-W は，インターネットを用いた書籍販売用の Web アプリケーションとして設計されており，シナリオに沿って 14 種類の Web Interaction を実行し，単位秒あたりの Web Interaction 数 (WIPS: Web Interaction per Second) によって性能を評価する．シナリオは，3 つのシナリオ，Browsing Mix, Shopping Mix, Ordering Mix が用意されており，それぞれによって実行する Web Interaction の比率が異なる．

TPC-W の仕様では，Web Interaction ごとに静的な Web ページ，動的な Web ページを返答することや，決められたテーブルのスキーマが指定されている．しかし，TPC-W の実装は指定されてなく，また，付加的なテーブルの生成は許されており，ブラウザの数，本の在庫数も選択することが可能である．

### 7.2. 実行環境

本検証では，FFF 上で実装した TPC-W を，表 1 に示すサーバ構成で起動する．本検証では，1 つのサーバ上で J2EE サーバプロセスと，Coordinator-Session のプロセス，データベースプロセスの 3 つのプロセスを，1 つのサーバ上で実行する．

なお，[9] で公開されている TPC-W の Java の実装は，データベースへのアクセスに JDBC を用いているため，JDBC を用いた記述から TDR を用いた記述に変更を行った．

[3] なお，現在の実装では Session オブジェクトの複製は実装されていない．

表1 サーバ構成

| CPU | Intel Xeon 3.20 GHz (Hyperthreading off) |
|---|---|
| Memory | 1GB |
| HDD | Ultra 320 SCSI 10000RPM |
| OS | Enterprise Red Hat Linux 4 (kernel 2.6.9) |
| Network | Gb Ethernet |
| Java | IBM J9VM (build 2.3, JRE 1.5.9) |
| Servlet | Tomcat 4.1.3 |
| Database | IBM DB2 UDB V8.1 |

また，本検証では，データアクセスフレームワークとして，Transactional Data Repository（TDR）を利用した．また，TPC-W の実装として，[9] で公開されている Java 実装を用いる．

なお，TDRをFFF上で利用する場合，アプリケーションサーバからデータベースに直接アクセスせず，データベースサーバ上でトランザクション用のスレッドを生成し，生成したスレッドからアクセスする．これは，アプリケーション障害時，もしくは，プライマリからバックアップに変更時に，データベース側で稼動中のトランザクションを強制終了するためである．また，TDRはインストールした最新LSNを，データベースの専用テーブルに書き込むことで，データベースサーバの障害時に同じコミットログを 2 度以上反映することを防ぐ．

### 7.3. 評価方法

#### 7.3.1. 評価内容

本稿では，(1)障害発生時のレスポンス時間と，(2)障害発生時のスループット，に関する性能評価を行う．

障害発生時のレスポンス時間の評価では，1つのブラウザが1つの同じ URL に，コネクションを開き，HTML を受信し，コネクションを閉じるまでのレスポンス時間を測定し，正常時と，障害発生時のレスポンス時間を比較する．本検証における正常時の実行とは，Proxy が要求したリクエストを 5.3.1 で示した処理を行うものとし，それ以外を障害発生時の実行とする．また，本検証では，[9] で実装された Servlet の内，TPCW_home_interaction Servlet に対する URL を利用する．TPCW_home_interaction Servle では関連する本の照会をデータベースに対して行い，動的に HTML を生成するリソースである．なお，比較に際し，30 秒に 1 度ずつプライマリを移動させる障害を発生させ，30 回障害を発生させた際のそれぞれの平均レスポンス時間と，最大レスポンス時間を示す．また，Proxy が最初の *request* 後に *re-request* を要求する時間を，1 秒と設定した．

障害発生時のスループットの評価では，TPC-W の Browsing Mix を，10 分間実行する．本検証では，10 分間に 5 回，10 回，15 回，20 回，プライマリを移動させる障害を発生させた中で Browsing Mix 実行した場合と，正常時の Browsing Mix の実行の場合の，処理済みトランザクション数を比較する．なお，ブラウザ数は 10とする．また，Proxy が最初の *request* 後に *re-request* を要求する時間を，3 秒と設定した．

#### 7.3.2. 対象とする障害

本稿で評価の対象とする障害は，データベースサーバの NIC(Network Interface Card)障害，アプリケーションサーバ障害とする．

データベースの NIC 障害は，データベースサーバのネットワークへの接続を遮断することで実現する．データベースサーバ間のネットワーク障害が発生した場合，障害が発生したコーディネータの *replicate* 処理が不能となり，合意に参加が不可能となる．そのため，障害が発生したデータベースサーバがプライマリであった場合，バックアップであったデータベースサーバが Take-over 処理を行い，トランザクションの実行を行う．

アプリケーションサーバ障害は，アプリケーションサーバで実行する J2EE サーバのプロセスを終了することで実現する．アプリケーションサーバ障害が発生した場合，Proxy は障害が発生したサーバへの *begin*，*re-begin* の処理が不能となり，トランザクションの開始が不可能となる．そのため，障害が発生したアプリケーションが接続するデータベースサーバがプライマリであった場合，バックアップであったデータベースサーバとそのデータベースサーバに接続するアプリケーションサーバが Take-over 処理を行い，トランザクションの実行を行う．

### 7.4. 障害発生時のレスポンス時間

図 7にデータベースサーバの NIC 障害，図 8にアプリケーションサーバ障害を発生させた場合のレスポンス時間と正常時実行時のレスポンス時間の比較を示す．

*request* と *re-request* の間隔は，1 秒と設定しているため，ネットワーク障害時も，アプリケーションサーバ障害時も，正常時と比べ，1 秒程度の遅れでシステムはレスポンスを返している．ネットワーク障害時に比べ，アプリケーションサーバ障害時は全体的にレスポンス時間が遅いのは，Tomcat のプロセスを強制終了しているため，JIT の効果を得られていないことが原因と考えられる．

### 7.5. 障害発生時のスループット

図 9 に，データベースサーバ間ネットワーク障害を発生させた場合の 10 分間のトランザクション数の比較を示す．

本結果より，FFF のデータベース間の NIC 障害に対する Take-over は，10 分に 20回，つまり 30 秒に 1 回障害が発生したとしても，10 分間の実行済みトランザクション数は 10%以下の低下しか起こさないことがわかる．また，実行済みトランザクション数は障害の回数にほぼ比例しており，1 回の障害で 0.4%から 0.7%低下することがわかった．

*request* 後に *re-request* を要求する時間を，3 秒と設定したため，切り替えにかかる時間は，最低でも 3 秒必要である．つまり，全体としてトランザクションが実行できない時間が正常実行時と比べ，5 回の障害で 2. 5%，10 回で 5%と，15 回で 7.5%，20 回で 10%存在する．実行済みトランザクション数の低下が，これらの割合よりも小さいのは，1 つのトランザクションに対して *navigate* された結果から，他のトランザクションの実行先も *begin* に失敗せずに *navigate* するためと考えられる．

## 8. 考察

### 8.1. 障害発生時のコスト

7 で示した評価結果より，提案する手法を用いることで，データベースがプライマリからバックアップに 1 秒以内に切り替わることができ，またスループットも 1 回の障害につき 10%以下であることが示された．しかし，本提案手法を用いた場合でも，データベース内のキャッシュが有効に使えない，全てのトランザクションにつき同期する，といった問題もある．

データベースでは，頻繁に利用されるクエリに対して，内部的にキャッシュを持つ製品が多い．しかしバックアップとしてデータベースが稼動する場合は，更新ログのみが複製されため，プライマリになった直後はキャッシュが有効に使えない可能性がある．

この問題を解決するには，プライマリ側で実行された参照用のクエリに関しても，適度な頻度でバックアップ時も実行するか，データベースの内部実装を変更する必要がある．

また，本提案手法は，1 つのトランザクションにつき，必ず2回以上他のデータベースと同期を取ることが必要である．通信にかかるコストは，空いた CPU 時間を他のトランザクションが利用できるため，スループットの観点からは少ない．しかし，Java のような抽象度の高い言語で提案手法を実装する場合，オブジェクトのシリアライズ，デシリアライズのコストが高く，通信量に比例して CPU 時間を利用することになる．この問題を解決するには，できる限り最小限のデータで，かつ，少ないメッセージ数で通信する必要があり，Piggy-back やブロック通信などの手法を利用して，最適化を図っていく必要がある．

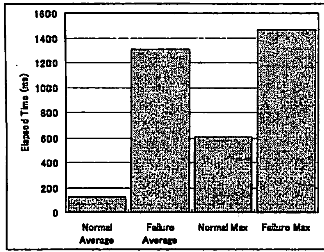


図 7: データベースサーバ NIC 障害時のレスポンス時間の比較

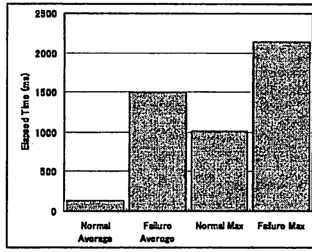


図 8:アプリケーションサーバ障害時のレスポンス時間の比較

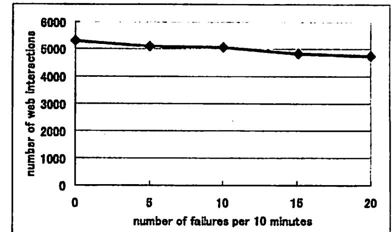


図 9:データベースサーバ NIC 障害時のスループットの比較

### 8.2. トランザクションキャッシュと楽観的トランザクションの利用

データベースの内部実装に手を加えない場合，1 スレッドにつき 1 トランザクションしか実行できない JDBC の制約が，システム全体のスケーラビリティに影響を与えることとなる．現在の TDR を利用した FFF の実装の場合，データベースサーバ上でトランザクション用のスレッドを立てる必要がある．この手法はスレッドの管理に CPU 時間を多く利用することとなる．

スケーラビィリティを向上させるためには，トランザクショナルキャッシュや楽観的ロックを用いたトランザクションの実行を行うことが有効である．近年，SDO や ADO のような非接続型のデータモデルが提案されており，また，様々なトランザクションキャッシュ製品が提供されている．このようなコンポーネントを利用することで，データベース側のスレッド管理のコストを縮小させ，かつシステム内のメッセージ数も減らすことが可能なため，システム全体として大幅な性能向上が見込まれる．

### 8.3. クライアントの高可用化

本提案手法は，クライアントの高可用化に関する手法は含まれていない．しかし，本手法は，*n* 階層のシステム構成にも拡張可能なため，ブラウザが *request, re-request* をすることでシステムの末端まで高可用化することが可能となる．

## 9. 関連研究

完全な障害検知に頼らずに，分散コンセンサス問題を解くことによって Take-over を実現する準受動的多重化(Semi-Passive Replication)の手法が提案されている[1]．この手法は，Chandra-Toueg の分散コンセンサスアルゴリズム[3] を応用した手法で，(1)ある処理を行っているプライマリのプロセスが障害と疑われたら，(2)他のプロセスがプライマリとして処理し，(3)コミット時に処理要求に対してどのプロセスの実行結果が *update* されるべきかをコンセンサスを得ることによって，障害検知に頼らずに，非決定的な処理のレプリケーションを実現可能とする．

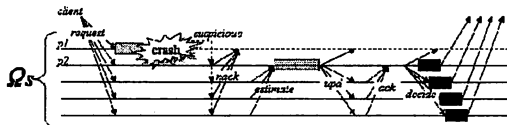


図 10: Semi-Passive Replication の障害時の振る舞い

しかし Semi-passive Replication は，図 1 のような OLTP のシステムに適用するには，アプリケーションサーバの障害対策，同時に実行されるトランザクション要求のシリアライズ，Web サーバの Reliable Multicast のコストを解決する必要がある．

Total Order を，分散で解決する手法は，従来多数提案されてきている[10] [11] [12] [13]．しかし提案手法で示す LSN の決定には，*begin, commit* 間にリーダーが変わってはならないという制約がある中で順序を決定しなくてはならず，従来手法をそのまま応用することは難しい．

## 10. まとめ

本稿では，OLTP システムが耐障害性システムとして満たすべき性質を示し，不確実な障害検知器を利用した手法を持って性質を満たすことで，高可用性システムを提案した．提案したシステムは，PAXOS コンセンサスアルゴリズムを応用した複製機構をもつシステムである．本手法は，(1)既存の階層型システムに適用でき，(2)トランザクションの特性を保ったまま，(3)障害時のコストを低く，実装することが可能であることを，提案手法の実装である FFF を用いて TPC-W を評価することで示した．

## 参考文献


[1] Flaviu Cristian: "Understanding Fault-Tolerant Distributed Systems" Communications of the ACM, 34(2): 56-78, February, 1991.

[2] Xavier Defago, Andre Schiper, Nicole Sergent: "Semi-Passive Replication" Proc of 17th IEEE Symp, Reliable Distributed Systems, pp.43-50, October, 1997.

[3] Tushar Deepak Chandra, Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems" Journal of the ACM, 43(2): 225-267, 1996

[4] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg: "The Weakest Failure Detector for Solving Consensus " Journal of the ACM, 43(2): 685-722, 1996

[5] Lesile Lamport: "The part-time parliament" ACM Trans. On Computer Systems, 16(2): 33-169, 1998

[6] Michel J. Fischer, Nancy A. Lynch, Michel S. Paterson: "Impossibility of Distributed Consensus with one Faulty Process" Journal of the ACM, 32: 374-382, 1985

[7] Roberto D. Prisco, Butler Lampson, Nancy Lynch "Fundamental study: Revisiting the Paxos algorithm" Theoretical Computer Science, 243:35–91, 2000.

[8] TPC Benchmark W Specification v1.0.1, Transaction Processing Performance Council, San Jose, CA, 2000, http://www.tpc.org/tpcw/

[9] TPC-W Java Implementation, http://tpcw.deadpixel.de/

[10] Xavier Defago, Andre Schiper, Peter Urban: "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey", ACM Computing Surveys, ACM Press, Vol36, No.4, pp.372-421, 2004

[11] Rachid Guerraoui: "Indulgent Algorithms" Proc the 11th International Conference on Distributed Computing Systems, pp.222-230, 1991.

[12] Yair Amir, Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, Paul W. Ciarfella: "The Totem single-ring ordering and membership protocol" ACM Trans. On Computer Systems, 13(4): 311-342, 1995

[13] Pedro Vicente, Luis Rodrigues: "An Indulgent Uniform Total Order Algorithm with Optimistic Delivery" Proc. 21st IEEE Intl. Symp. on Reliable Distributed Systems, 2002