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

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

堀井 洋[†] 田井 秀樹[‡] 山本 学[§]

Hiroshi HORII Hideki TAI Gaku YAMAMOTO

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，(1)各層個別の高可用化機構を連携させた複雑な構成，(2)障害検知時間に依存したサービス停止時間，が必要であった．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 合意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．

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)がミッションクリティカルな業務に利用され始めている中，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. 一般的な OLTP システムは，Web サーバ，データベースサーバから構成されおり(図 1)，End-to-End でシステムを高可用化するには，多重化した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[1] ．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．しかし，このような手法を利用した場合では，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
- システムごとに障害検知後の設定が必要となる
- 完全に障害を隠蔽するためには，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

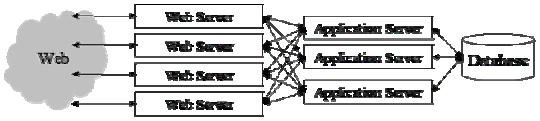

図 1 一般的な OLTP システム
Fig.1 Typical OLTP system

本稿では，PAXOS 合意アルゴリズムを用いてトランザクションログ，セッション情報を複製することで，障害発生後に短時間で take-over の完了が可能な OLTP の高可用化機構を提案する．以下本稿では、この機構を PCHA(PAXOS Consensus based High-Availability)と呼び，PCHA 上で J2EE の HttpServlet 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を示し，評価を行う．

2. PAXOS Consensus による HA 機構

2.1 トランザクション処理

2.1.1 概要

PCHAは，図 2 のようにトランザクションを実行する．システムを構成するプロセスは，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集合*AP*，*m*個のデータベースプロセス集合*DB*である．それぞれのプロセスは，通常実行時，下記のようにトランザクション$t_i \in T$を実行する．

*client*はトランザクションt_i の実行を要求する(*request*)．要求を受けた$ap_j \in AP$は$db_k \in DB$にトランザクションの開始を要求(*begin*)し，db_kにクエリを行いながら(*query*)アプリケーションロジックを実行後, db_kにトランザクションの終了を要求する(*commit*)．*commit*の要求を受けたdb_kは，ap_jがt_iの処理中に生成した更新ログ$log_i \in LOG$を他の$db_h \in DB$(k≠h)に対して複製の要求をする(*replicate*)． *replicate*の要求を受けたdb_hは複製終了を通知後，log_iを反映する(*update*)．複製終了通知を受信したdb_kはlog_iを *update* し，t_iの終了をap_jに通知する．そして，t_iの終了を受けたap_jはt_iの終了を*client*に通知する．

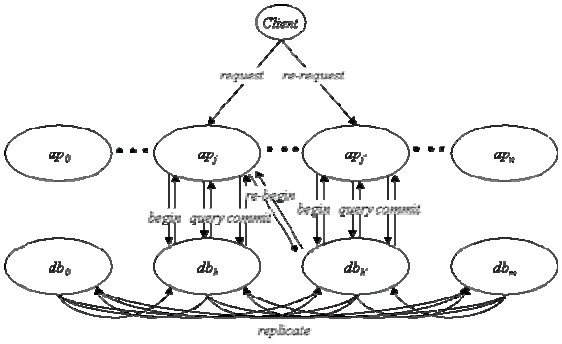

図 2: PCHA 上でのトランザクションの実行
Fig.2 Processing model of transactions on PCHA

[†]正会員 日本アイ・ビー・エム 東京基礎研究所
horii@jp.ibm.com
[‡]日本アイ・ビー・エム 東京基礎研究所
hidekit@jp.ibm.com
[§]日本アイ・ビー・エム 東京基礎研究所
yamamoto@jp.ibm.com

PCHAを利用するOLTPシステムは，ap_jもしくはdb_kにからの返答が遅い場合，下記のようにトランザクション$t_i \in T$の再実行を行う．

$request$後t_iの終了通知が一定時間来ない場合

$client$は$ap_{j'}$にt_iの実行を再度要求する($re\text{-}request$)．$re\text{-}request$を受信したap_jは，正常実行時と同様にt_iの実行を行う．

$begin$，$query$，$commit$ 後，実行の終了通知が一定時間来ない場合

ap_jはdb_kに対し，t_iの再実行を行う($re\text{-}begin$).

2.1.2 原子性

db_kが$commit$処理を終了後，ap_jに$commit$終了通知を行う前に障害が発生した際，二重にlog_iを$update$することを防ぐため，下記のような処理を行う．

$client$は$request$時に，$t_i \in T$に対し一意な値として$tid_i \in TID$を生成する．そして，$client$，AP，DBのプロセスは，全てのt_iに関する処理要求に対し tid_iを付与して行う．また，全てのdb_kは$update$が完了したtid_iとlog_iの対を保存する($cTID$, $cLOG$)

db_kは，ap_jからtid_iのトランザクションt_iの，$begin$, $re\text{-}begin$, $query$，$commit$要求を受ける際，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$commit$済みをap_jに通知する($committed$)．含まれない場合は，処理要求を実行する．

2.1.3 一意性

PCHAでは，トランザクションの一意性を保持するため，DB内のプロセスdb_kが，$begin$，$re\text{-}begin$実行時，下記の条件を満たす際に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

プライマリになれないと判断された場合，ap_jに現在のプライマリdb_kを通知する($navigate$).

プライマリになれると判断された場合，$replicate$されている$cLOG$の内，最後のlsnを記録する($begin\_lsn_i$).

$commit$要求を受けた際，$replicate$されている$cLOG$の内，最後のlsn ($commit\_lsn$)から，$begin\_lsn_i$ の間に$update$した$cLog$の内，db_k以外が$replicate$したlogが存在していた場合，$commit$不可能と判別し，トランザクションの再実行をap_jに要求する．

2.1.4 $committed$，$navigate$ 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．

2.2 PAXOS 合意アルゴリズムを利用した $replicate$ 処理

PCHA では, PAXOS 合意アルゴリズムを用いて, $replicate$ 処理を行う．

2.2.1 PAXOS 合意アルゴリズム

PCHA では, PAXOS 合意アルゴリズム[5] をトランザクションログの複製のために利用する．

合意アルゴリズムとは，それぞれのプロセス Pi が値 Vi を提案し合い，停止性，合意性，妥当性を満たすように，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．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] が，PAXOS 合意アルゴリズムのように，不確実な障害検知器[3] を用いることで，現実的なシステムでは停止性を保障することが可能なアルゴリズムが提案されている．

PAXOS 合意アルゴリズムは，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，通信しあうことで合意を得る．プロセスは合意を得る際，(1) Info-Quorum の構築，(2) Accepting-Quorum の構築，(3) 結果の通知を行う(詳細は[7]を参考) ．PAXOS 合意アルゴリズムは，過半数以上のプロセスに障害が起こらない限り，合意を得ることが可能である．

2.2.2 $replicate$ 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 合意アルゴリズムを利用して，最新 lsn_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，仕掛かり中のトランザクションの内，$begin\_lsn$ がlsn_iより小さい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($navigate$).

2.3 PCHA の性質

PCHA は，PAXOS 合意アルゴリズムを用いてトランザクションログの複製をするため，データベースサーバ間の障害検知に不確実な障害検知器を利用することが可能となる．PCHA が用いる不確実な障害検知器の特性は，PAXOS 合意アルゴリズムと同様，Ω[3] となる．

また，クライアント，アプリケーションサーバの再送機構は，障害検知を行わずに再送するため，障害検知時間に依存することなく take-over を行うことが可能となる．これらの再送機構は，Ωの不確実な障害検知器と同等の機能を持つため, システム全体として停止性を保障することも可能である．

PCHA 上で実行されるトランザクション処理は，PCHA の機構により再実行，take-over が行われるため，障害を考慮することなくアプリケーションを記述することが可能となる．また，アプリケーションサーバのセッション情報を，データベース側で管理することで，一意性を持ったセッション管理も行うことが可能となる．

2.4 PCHA の実行例

PCHAを用いたトランザクションの実行の流れを, 図 3(正常時）図 4（db_0の障害時)， 図 5（ap_0の障害時）に示す．

図 3 に示すとおり，db_0は$commit$時，4 ステップ，(m-1)×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．また，図 4，図 5 のように，$client$，APが，障害の有無に関わらずトランザクションの再実行を要求するため，障害検知に時間がかかったとしても短時間でtake-overが可能となる．また，障害が起きていないにも関わらずトランザクションの再実行を要求してしまったとしても，PAXOS合意アルゴリズムとコミット済みトランザクションの管理機構により二重トランザクションを防ぐことが可能となっている．

3. 性能評価

PCHA を用いたトランザクショナルな Servlet フレームワーク上で，障害発生時の性能評価を TPC-W[8] を用いて行う．

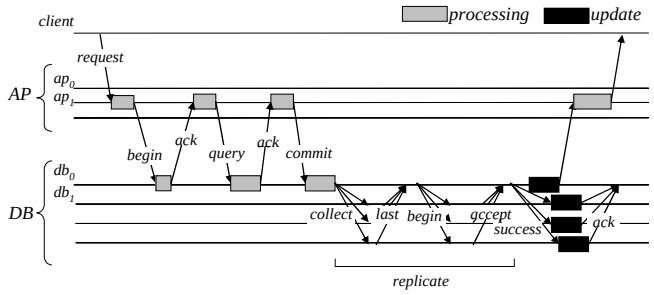

図 3：正常時の処理
Fig.3 Transaction processing flow in a normal case

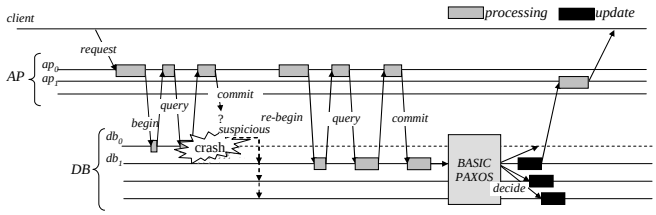

図 4： db_0の障害時の処理
Fig.4 Transaction processing flow in a failure case of db_0

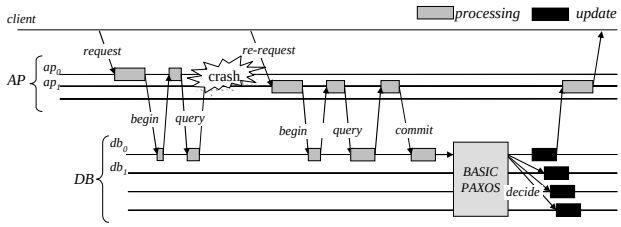

図 5：ap_0の障害時の処理
Fig.5 Transaction processing flow in a failure case of ap_0

3.1 PCHA を用いた Servlet フレームワーク

PCHA 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 から構成される．HTTP サーバは，HTTP リクエストを受信し，Proxy に転送する．Proxy は，HTTP Server から転送されてくる HTTP Request に対し，生成した *tid* を HTTP ヘッダーに追加し，*request*，*re-request* の機能を実装する．

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 から構成される．Servlet Filter は，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 オブジェクトの *commit* 送信を要求する．Message Broker は，Servlet Filter から *begin*，*commit* の要求を，アプリケーションからの *query* 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．

データベースサーバは，Coordinator とデータベースから構成される．Coordinator は，*begin*，*re-begin*，*commit*，*replicate* の要求を受け，*begin*，*re-begin*，*commit* の実行が可能かを判断や，*replicate* 時の分散合意に対する処理を行う．

3.2 検証環境

本検証では，PCHA 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．また，[9] で公開されている TPC-W の Java 実装を，PCHA のデータアクセス用 API に変更した実装を TPC-W の実装として利用した．

表1 サーバ構成
Table 1 Server specification

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

3.3 評価方法

本評価では，(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, 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る中，1 つのブラウザが[9] の（1）TPCW_home_interaction Servlet を起動する URL を指定し，（2）HTML を受信し，（3）コネクションを閉じるまでの，正常時と，障害発生時のレスポンス時間を測定する．なお，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，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，Browsing Mix 実行した際と，正常時の Browsing Mix の実行の際の，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 とし，Proxy が最初の *request* 後に *re-request* を要求する時間を，3 秒と設定する．

本評価の対象とする障害は，データベースサーバの Network Interface Card 障害(データベースサーバ間のネットワークへの接続を遮断する)，アプリケーションサーバ障害(J2EE サーバのプロセスを終了する)とする．

3.4 障害発生時のレスポンス時間

図 6 にデータベースサーバの NIC 障害，図 7 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．

障害発生時のスループット

図 8 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. 本結果は，PCHA 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を示している．

4. 考察

4.1 レスポンス時間

図 6 で示したように *request* と *re-request* の間隔を 1 秒と設定することで, 2.5 秒以内に障害発生時の take-over が行われている．*request* と *re-request* の間隔をより短くすれば，take-over をより高速にすることが可能になると考えられる．しかし，*request* と *re-request* の間隔を短くしすぎると，無駄なシステムリソースの無駄な消費を招く可能性や，実装によっては，Ωの特性を満たせなくなってしまう可能性がある．

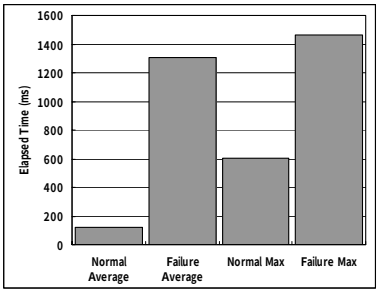

図 6：データベースサーバ NIC 障害時のレスポンス時間の比較
Fig.6 Response time comparison between normal and NIC failure cases

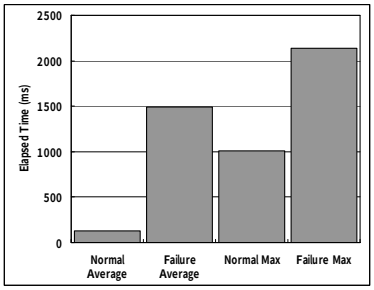

図 7：アプリケーションサーバ障害時のレスポンス時間の比較
Fig.7 Response time comparison between normal and application server failure cases

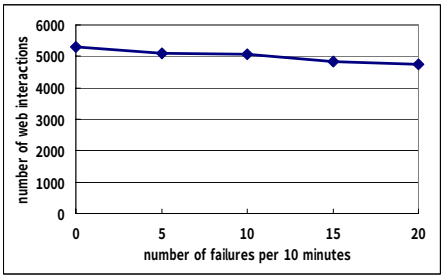

図 8：データベースサーバ NIC 障害時のスループットの比較
Fig.8 Throughput comparison between normal and NIC failure cases

スループット

図 8 で示した PCHA 上での TPC-W の実行スループットは，一般的な OLTP システムとしては低い．現状では，実装上の問題が大きいが，今後，データベース内のキャッシュが有効利用できない，通信の処理コストが高いといった問題が起こる可能性がある．

バックアップとしてデータベースが稼動する間，参照用のクエリに関するキャッシュは有効にはならない．この問題を対処するには，定期的にデータベース間で参照用クエリのキャッシュの同期を持つような機能が必要となる．

PCHA 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ため，通信処理にかかるコストが大きい．この問題を対処するには，複数のログをまとめて合意をとる方法や，分散合意アルゴリズムの最適化[7] ，Piggy-back やブロック通信などの最適化手法を用いる必要がある．

4.2 高可用化の範囲

本手法は *client* 以降のプロセスの障害に対処する．つまり，*client* の障害に対処できない．しかし，本手法は *n* 階層のシステム構成にも拡張可能であり，階層を増やすことで，システムの末端(例えばブラウザ)に *request*，*re-request* を実装し，高可用化の範囲を広げることが可能である．

5. 関連研究

Chandra-Toueg の分散合意アルゴリズム[3] を応用して 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] ．本手法を用いることで，障害検知に頼らずに，非決定的な処理の複製を実現可能であるが，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を行う必要がある．

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[9] ．しかし，トランザクションの一意性を満たす制約がある中で LSN 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．

6. まとめ

本稿では，PAXOS 合意アルゴリズムを利用したトランザクションログの複製手法を提案し，提案する複製手法を用いたトランザクション処理システムを示した．提案したシステムを TPC-W を用いて評価し，(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)短時間で take-over 可能であることを示した．

[文献]

[1] Cristian, F: “Understanding Fault-Tolerant Distributed Systems” Communications of the ACM, 34(2): 56-78, February, 1991.
[2] Defago, X, Schiper, A and Sergent, N: “Semi-Passive Replication” Proc of 17th IEEE Symp, Reliable Distributed Systems, pp.43-50, October, 1997.
[3] Chandra, T, D and Toueg, S: “Unreliable Failure Detectors for Reliable Distributed Systems” Journal of the ACM, 43(2): 225-267, 1996
[4] Chandra, T, D, Hadzilacos, V, Toueg, S: “The Weakest Failure Detector for Solving Consensus ” Journal of the ACM, 43(2): 685-722, 1996
[5] Lamport, L: “The part-time parliament” ACM Trans. On Computer Systems, 16(2): 33-169, 1998
[6] Fischer, M, J, Lynch, N and Paterson, M, S: “Impossibility of Distributed Consensus with one Faulty Process” Journal of the ACM, 32: 374-382, 1985
[7] Prisco, R, D, Lampson, B and Lynch, N “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] Defago, X, Schiper, A and Urban, P: “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey”, ACM Computing Surveys, ACM Press, Vol36, No.4, pp.372-421, 2004

堀井 洋 Hiroshi HORII
2004 年早稲田大学大学院修士課程修了．同年日本アイ・ビー・エム株式会社入社，東京基礎研究所副主任研究員．業務システム基盤技術の研究に従事．情報処理学会正会員．日本データベース学会正会員．

田井 秀樹 Hideki TAI
1997 年筑波大学大学院修士課程修了．同年日本アイ・ビー・エム株式会社入社，東京基礎研究所主任研究員．分散システムのミドルウェアおよび管理機構などの業務システム基盤技術の研究に従事．情報処理学会正会員．

山本 学 Gaku YAMAMOTO
1991 年東京工業大学大学院修士課程修了．同年日本アイ・ビー・エム株式会社入社，東京基礎研究所専任研究員．エージェント基盤技術および業務システム基盤技術の研究に従事．情報処理学会正会員．