J2EEクラスタリングシステムへの アプリケーションパーティショニングの適用による処理性能の向上

夷藤 勇人　根山 亮

Performance Improvement of J2EE Clustering Systems through Application Partitioning

Hayato Itoh　Ryoh Neyama

本論文では, J2EEクラスタリングシステムにアプリケーションパーティショニングを適用した新アーキテクチャを提案する. この新アーキテクチャを用いることによって, クラスタリングシステムの利点である耐障害性・スケーラビリティを保ったまま, アプリケーションにほとんど変更を加えることなくパーティショニングを行うことが可能になる. クラスタリング環境でありながらキャッシュを有効に利用できることで, 従来, 実現が困難であったミッションクリティカルかつ高性能が要求されるシステムが実現できる.

This paper proposes a new architecture which applies application partitioning to J2EE clustering systems. This enables partitioning to be done with almost no changes to applications while maintaining fault tolerance and scalability which are key advantages of clustering systems. Because cashes can be effectively utilized even in clustering environments, mission critical systems requiring high performance which have so far been hard to implement can now be constructed realistically.

Key Words & Phrases：処理性能の向上, J2EEクラスタリングシステム, アプリケーションパーティショニング, アプリケーションサーバー, キャッシュ
performance improvement, J2EE clustering system, application partitioning, application server, cache

1. はじめに

J2EE環境でシステムを構築する際は, クラスタリングシステムが広く用いられるようになった. クラスタリングシステムの利点としては, アプリケーションサーバー（以後, 単に「サーバー」と表記）の台数を柔軟に変更できることから得られるアプリケーションロジック実行のスケーラビリティ, 耐障害性等が挙げられる.

その一方, クラスタリングシステムでは複数サーバーからのSQLリクエストが1台のデータベースに集中するため, データベースがボトルネックになりやすい. データベースボトルネックを解消するためには, データベースの処理性能そのものを改善することに従来は焦点が当てられていた.

別の効果的な手段として, データベースのデータをアプリケーション内にキャッシュすることが考えられる. キャッシュを使用することにより, アプリケーションからデータベースへの参照アクセスの必要がなくなり, データベースへの負荷そのものを低減できる. データベースにアクセスするサーバーがひとつの場合はキャッシュを使用しても問題はない. しかし, サーバーが複数の場合はデータ不整合性の問題がおきる.

複数サーバー環境においてデータ整合性を保つには, 特定のデータに対して特定のサーバーのみが独占的にアクセスすることが保障できればよい. これを実現するため, アプリケーションパーティショニング[1]と呼ばれる方法がある. データをパーティションと呼ばれるデータ集合に分割し, 各パーティションを担当するサーバーをあらかじめ決めておく. あるデータに対する処理リクエストを, そのデータに対応するパーティションを担当するサーバーに割り振るようにすれば, キャッシュを使用してもデータ整合性は保たれる.

このようにアプリケーションパーティショニングはデータベースボトルネックを解消するための方法として効果的ではあるが, 従来のクラスタリングシステム上では実現が不可能であった. リクエスト処理を担当する

提出日：2004年8月31日　再提出日：2005年8月31日

のはどのサーバーでも構わないというのがクラスタリングシステムの前提であり, これはアプリケーションパーティショニングがもつ特定のサーバーが特定の処理リクエストに対する処理を行うという性質とは両立しないためである.

本論文では, J2EEクラスタリングシステムの利点を保ったまま, アプリケーションにほとんど変更を加えることなく, アプリケーションパーティショニングを実現するための新アーキテクチャを提案する. 新アーキテクチャでは, J2EEクラスタリングシステムの利点と, アプリケーションパーティショニングによる処理性能上の利点, 従来両立が不可能であった両方の利点を同時に実現できる. これにより, 従来, J2EE環境において実現が困難であったミッションクリティカルかつ高処理性能が要求されるシステムの実現が可能になる.

本論文では, 最初に, 従来のJ2EEクラスタリングシステム, アプリケーションパーティショニング, この2つのアーキテクチャの一般的特徴について説明する. 次に, 新アーキテクチャがどのように両アーキテクチャを融合するかを説明する. 最後に, 株売買取引業務に新アーキテクチャを適用することにより, その有用性を示す.

2. 従来のアーキテクチャの課題

2.1 J2EEクラスタリングシステム

まず現在広く用いられているJ2EEクラスタリングシステム(**図1**)について見てみる. クラスタリングシステムは全サーバーが同等の機能を持つことを前提としている. そのため, クラスタリングシステム上で動作するアプリケーションは「ビジネスロジックの状態」を特定のサーバーに保存するのではなく, すべてデータベースに保存する必要がある.

各サーバーが状態を持たなければ, クラスタリングシステムを構成するサーバー台数を変化させても, アプリケーションには影響はない.「クラスタリング環境からのアプリケーションの独立性」を保つことは, 柔軟なクラスタリング構成のため必要不可欠である.

その代償として, クラスタリングシステムでは, データベース上の更新されうるデータをメモリ内にキャッシュすることは許されない. データ整合性が保障されないからである.

データの整合性を保証するため, サーバーは, 図1のように, トランザクション[2]ごとに更新されうるデータに対するロックと最新の値をデータベースから取得し, トランザクションの範囲内で値をキャッシュするのが一般的である. つまり, トランザクションの範囲を超えてデータを効率的にキャッシュすることはできなかった.

高いスループットと短いレイテンシが要求されるシステムでは, メモリキャッシングが行えないという制限は致命的な処理性能上の限界につながる.

例として, 証券取引所の株売買取引業務を考える. 株売買取引システムには, 各証券会社から毎秒数百～数千件の株売買オーダーが入ってくる. 株売買取引システムは, 株売買の成立をチェックし結果をデータベースに書き込むマッチング処理を, 銘柄ごとに1トランザクションとして行う必要がある. マッチング処理のたびに処理対象となるオーダーをデータベースからサーバーにロードすると, オーダー数が多い場合データベースがボトルネックとなる.

2.2 アプリケーションパーティショニング

2.1節で述べた処理性能上の問題を解決するために用いられるのがアプリケーションパーティショニングである.

アプリケーションパーティショニングでは, 特定のデータ集合に対する処理を特定のサーバーで行うようにすることで, トランザクションの範囲を超えてデータを効率的にキャッシュできるようになる. この特定のデータ集合を「パーティション」と呼ぶ. 異なるパーティションに属するデータには重なりがなく, 互いに独立して処理を行えることが前提条件である. 前述の株売買処理の例でいえば, パーティションは銘柄ごとに分けることになる. 各銘柄の売買処理は, 他の銘柄の処理とは関係なく独立して行える.

銘柄をキーにしてアプリケーションパーティショニ

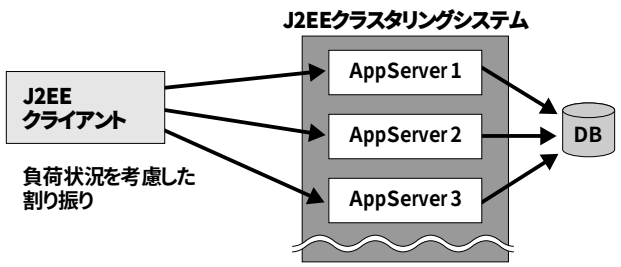

図1. J2EEクラスタリングシステム

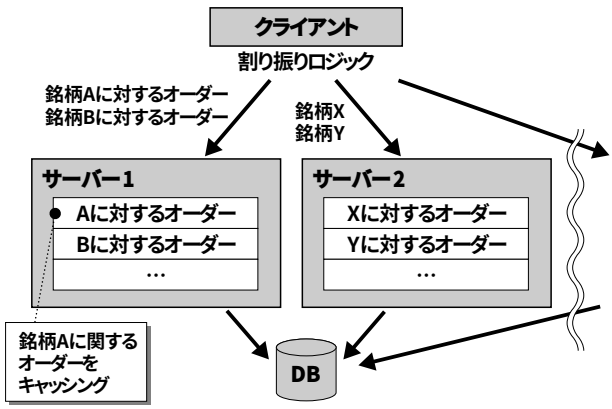

図2. アプリケーションパーティショニング例

ングを行う場合のシステム構成図を**図2**に示す. 各銘柄に対応するパーティションを作成し, パーティションの各サーバーへの配置をあらかじめ決定しておく. クライアント側に, 処理リクエストの各サーバーへの割り振りロジックを作成する. すなわちある銘柄に関する処理リクエストが必ず対応パーティションを管理しているサーバーで処理されるようにする. こうしておけば, あるパーティション内の処理は, 他のサーバーの存在を気にする必要はなく, 1つのサーバーで独占的に処理できる. 図2で言えば, サーバー1が銘柄Aに関するすべてのオーダーを受け付けることになる. 銘柄Aに対するオーダーをキャッシュしておけばマッチング処理のたびに銘柄Aに対するオーダーをデータベースからロードする必要はない. オーダーの参照に伴うデータベースへのアクセスがなくなり, データベースボトルネックを解消できる.

その一方, このシステムはクラスタリングシステムではないため, クラスタリングシステムが持つ利点は享受できない. 例えば, サーバー台数等を変化させる場合は, 手作業でアプリケーションまたは設定情報を書き直してパーティション割り振りロジックを変更する必要があり, 柔軟性に欠けるシステムとなる. またサーバーダウン等からの耐障害性を確保するためには, アプリケーションレベルでサーバーダウンを検知してリカバリー処理を行う, といった作り込みが必要になり, アプリケーション作成の負荷が大きい.

3. J2EEクラスタリングシステムへのアプリケーションパーティショニングの適用

本章では, 2章で述べた両アーキテクチャそれぞれの欠点を解消し, 両方の利点を同時に実現することを目指した新アーキテクチャを提案する. これにより, クラスタリングシステムの利点である1). アプリケーションのクラスタリング環境からの独立性(アプリケーションを変更することなくサーバー台数等を増減できる), 2). 障害からのリカバリーの自動化, をそのまま保ち, なおかつアプリケーションパーティショニングの利点である, 3). 特定のサーバーで特定のデータ集合に対する処理を独占的に行うことによって効率的なキャッシュの利用が可能になる, という従来両立が不可能であった利点を同時に実現できるようになる.

3.1 新アーキテクチャ概要

新アーキテクチャの概要を**図3**に示す. 新アーキテクチャの基本コンセプトは以下の通りである.

基本となるのは従来のJ2EEクラスタリングシステムアーキテクチャである. J2EEクラスタリングシステムは既に成熟したシステムであり, その恩恵はすべて

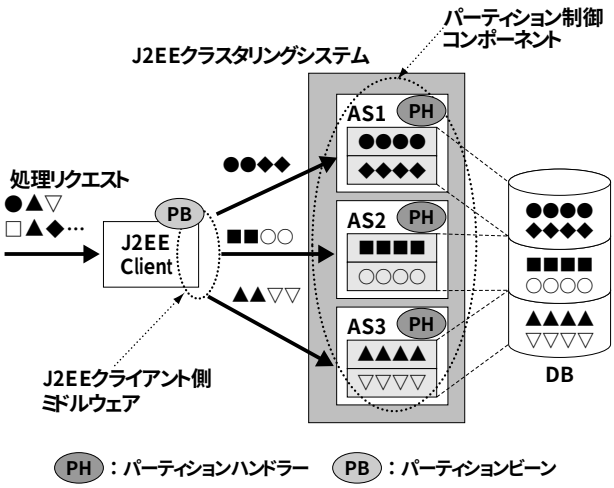

図3. 新アーキテクチャ概要

享受することが目的である.

従来のように静的にパーティショニングを行った場合はクラスタリングシステムのようなサーバー台数等が変化することを前提にした環境に対応できない. そこで新アーキテクチャでは, パーティショニングをミドルウェアが自動的に行う. サーバー台数等のクラスタリング環境を稼動時に把握できるのはミドルウェアである. ミドルウェアにパーティションの制御を担当するパーティション制御コンポーネントを組み込んでおき, 現在のクラスタリング環境を考慮して各サーバーへのパーティションの割り振りを動的に行う.

新アーキテクチャにおけるアプリケーションはパーティションを直接管理しない. その代わり, アプリケーションは, パーティショニングに必要な情報(パーティションの一覧等)を, ミドルウェアに教える必要がある. 逆に, アプリケーションは従来静的に定めていたサーバーに対するパーティションの配置を, 動的にミドルウェアから通知してもらうことになる. このようなアプリケーションとミドルウェア間で必要な情報をやりとりするため, アプリケーションは2つのコールバック・コンポーネントを用意する. 1つはサーバー側で呼び出される「パーティションハンドラー」であり, もう1つはクライアント側で呼び出される「パーティションビーン」である.

パーティションハンドラーとして実装するべき機能は以下の2点である.

- パーティション名のリストをパーティション制御コンポーネントへ通知する
- パーティション制御コンポーネントからのパーティション配置イベントを受け取る

パーティションビーンは以下の機能を実装する必要がある.

- クライアントが送信しようとしている処理リクエストからパーティション名を特定する

新アーキテクチャは以下の利点が得られる.

(1) 従来のクラスタリングシステムの恩恵, すなわち, 耐障害性, 負荷分散, スケーラビリティはそのまま享受できる.

(2) アプリケーションは「パーティション一覧」をミドルウェアに提供するだけで自動的にパーティショニングの恩恵を受けることができ[3], キャッシュを効率的に利用できる.

3.2 新アーキテクチャ処理フロー

本節では, 3.1節で述べた新アーキテクチャの基本コンセプトに対する理解の助けとなるように, 具体的にどのようにパーティション操作が行われるかを説明する.

3.2.1 パーティション初期配置

まずアプリケーション起動時, どのようにパーティショニングが行われるか説明する(**図4**).

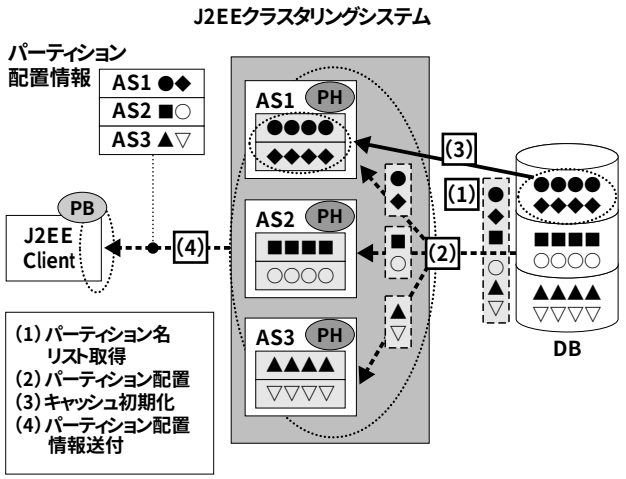

図4. パーティション初期配置

(1) パーティションを識別する文字列である「パーティション名」のリストを, パーティションハンドラーがデータベースから取得する. 前述の株売買業務システムでは, これは銘柄コードの一覧に相当する.

(2) パーティション制御コンポーネントがパーティション名のリストをパーティションハンドラーから受け取る. パーティション名に対応するパーティションを作成して, クラスターメンバーである各サーバーへパーティションを割り振る.

(3) パーティション制御コンポーネントは, 各サーバーのパーティションハンドラーに対して, 割り振ったパーティション名を通知する. これを受け, 各サーバーは担当パーティションに関する初期化処理を行う. 株売買業務の例でいえば, パーティションに対応する銘柄の既存注文データをデータベースからロードしてキャッシュしておく.

(4) パーティション制御コンポーネントは, J2EEクライアント側のミドルウェアに, パーティション配置情報を通知する.

3.2.2 処理リクエストのルーティング

アプリケーション起動後, J2EEクライアント部分では, 以下のようにして, 処理リクエストをパーティションが割り当てられたサーバーにルーティングする.

(1) クライアント側アプリケーションがクライアント側ミドルウェアを通じて処理リクエストを送信する.

(2) クライアント側ミドルウェアが処理リクエストを事前に登録されたパーティションビーンに渡す.

(3) パーティションビーンが渡された処理リクエストからパーティションを特定し, クライアント側ミドルウェアに知らせる.

(4) クライアント側ミドルウェアは, パーティションビーンから得たパーティション名とサーバー側から通知されたパーティション配置情報を照らし合わせ, 処理リクエストを送信すべきサーバーを決定し, 処理リクエストを送信する.

3.2.3 パーティション再配置

パーティショニングを行うと, 全サーバーが同等の機能を持たなくなるため, クラスタリングシステムが本来持つ耐障害性・負荷分散機能は失われてしまう. そこで, 新アーキテクチャでは, パーティション再配置をミドルウェアが自動的に行うことによって, クラスタリングシステムの利点を失うことなくパーティショニングを維持する.

例としてサーバーダウン時について説明する. この場合は, パーティション制御コンポーネントがそのサーバー内に含まれていたパーティションを別のサーバーへ移動させることによってパーティショニングを維持する(**図5**).

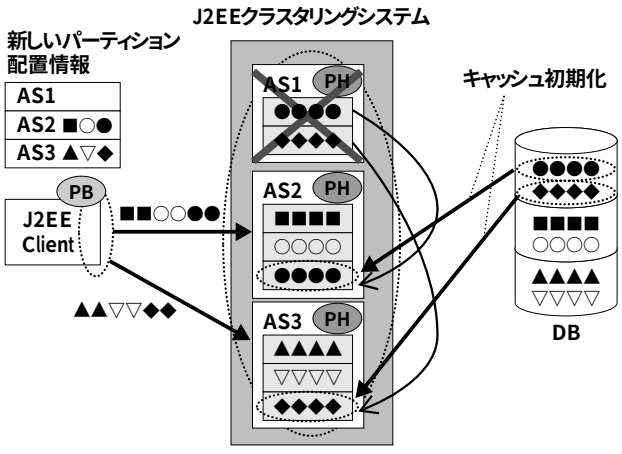

図5. サーバー障害時のパーティション再配置

従来のJ2EEクラスタリングシステムではサーバーの負荷状況を考慮して処理リクエストが割り振られるが, 新アーキテクチャでは処理リクエストの割り振りはパーティション配置情報に基づいて行われる. そのため, 処理負荷が高いパーティションが一部のサーバーに集中してしまった場合は, 特定のサーバーのみが高負荷になる恐れがある. そのため, パーティション再配置を行うことで負荷分散を行う. クラスタリングシステムが本来持つワークロードマネージメント機能がサーバーの負荷状況に偏りがあることを検知して, 負荷が均等になるようにパーティションの再配置を行う. すなわち, 負荷が高いサーバーから負荷の低いサーバーへパーティションを移動させる.

パーティション再配置が起きた時は, パーティション制御コンポーネントは必ず以下の2点を行う.

(1) パーティションハンドラーにパーティションが自サーバーに割り振られた, または, 自サーバーから移動されたことを通知する. アプリケーションは, このイベント通知を受けて, 該当パーティションに関するキャッシュの初期化やキャッシュの開放を行う.
(2) J2EEクライアント側のミドルウェアに新しいパーティション配置情報を通知する.

パーティショニングに必要な処理は, パーティションハンドラーとパーティションビーンに局所化され, アプリケーションに対するパーティショニングのための変更は少なく済む. パーティションの初期配置や再配置などの複雑な処理はミドルウェアが自動的に行うため, パーティショニングはアプリケーションに対して透過的であるといえる.

4. 新アーキテクチャ実装例とその効果

本章では, 3章で述べた新アーキテクチャをどのように適用・実現すればよいのか, 実際の実装例で説明する. ここで取り上げるのは前述の株売買業務システムである.

新アーキテクチャを採用した株売買業務システムを実際に構築し, J2EEクラスタリングシステム上で高性能・高信頼性を実現できることを検証した.

4.1 新アーキテクチャの実装例

新アーキテクチャ上に実装した株売買業務システムの構成図[4][5]を**図6**に示す.

J2EEアプリケーションサーバーとしては, WebSphere® Application Server[6] V5.1betaを使用, 新アーキテクチャを具現化するパーティション制御コンポーネントとして, 当時, 筆者らで開発中であったWPF(WebSphere Partitioning Facility)を今回の検証のため特別に用意して使用した.

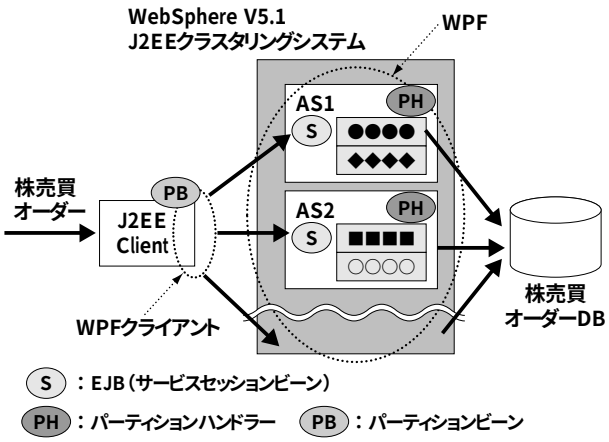

図6. 実装システム構成図

このシステムでは, EJB[7][8]・ステートレスセッションビーンをサービスとしてクライアントに公開する. J2EEクライアントは, このセッションビーンのリモートメソッド呼び出しを行うことによって, 株売買オーダーの処理リクエストを送る.

クライアント側のセッションビーンのスタブ部分が, クライアント処理リクエスト振り分けコンポーネントに該当する. 処理リクエストとパーティションの対応を知らせてもらうため, アプリケーションからパーティションビーンを提供する.

パーティションハンドラーとしては, WPFではEJBのステートレスセッションビーンのローカルインターフェースを用意する. アプリケーションがこのインターフェースの実装を提供することによって, WPFに認識されパーティショニングが有効になる.

4.2 アプリケーションデザイン上の注意点

本節では, 新アーキテクチャで稼動するアプリケーションを作成する際のデザイン上の注意点について説明する. この新アーキテクチャの恩恵を最大限に生かすために重要になるのはキャッシュのデータ構造である. 今回のシステムでは, 新規オーダーが高頻度で入ってくる. また, マッチング処理を行う際には, オーダーを売買優先順位でソートしておく必要がある.

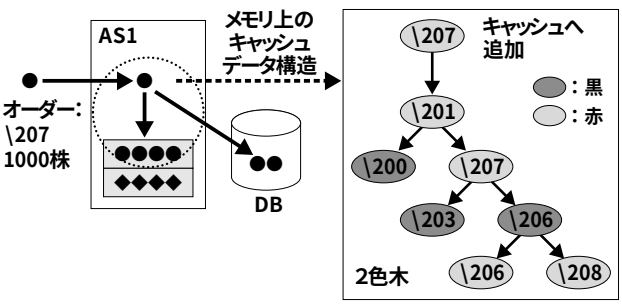

図7. キャッシュデータ構造

このため, 今回のキャッシュデータ構造としては, 平衡2分探索木[9]の1つである2色木を採用した(**図7**). 常にオーダーをソート済みの状態に保つことができることと, 既存オーダー数をnとした場合, 新規オーダーをソート済みのツリーに挿入するコストがO(log(n))[9]で済むことがその理由である. 新アーキテクチャではキャッシュデータ構造によりシステムのパフォーマンスは大きく左右されるので, 適切なデータ構造を採用することが重要である.

4.3 パーティショニングの効果

通常のJ2EEクラスタリングシステムで処理を行う場合(A)と, 新アーキテクチャで処理を行う場合(B), それぞれの株売買処理フローの違いを**図8**に示す.

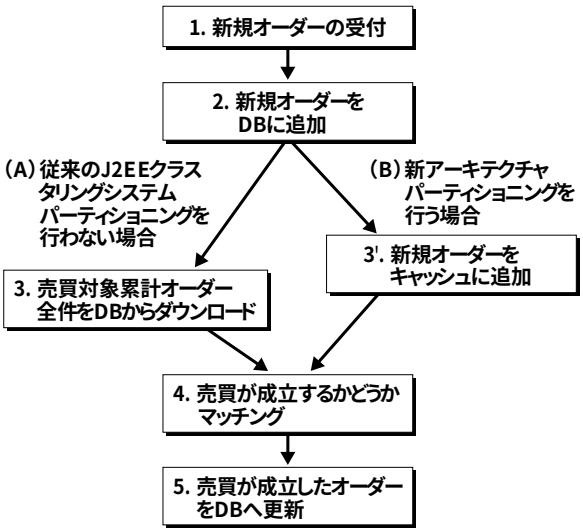

図8. 株売買処理フロー

両者の違い, すなわちキャッシュの使用がどれほどパフォーマンスの向上をもたらすかを評価するためベンチマークを実施した.

ベンチマークは1サーバーを対象に行う. サーバーが扱う銘柄数は20とし, 20クライアントが各銘柄に対して注文を連続して行う. マッチング処理の結果, 売買が成立し取引が完了したオーダーの総数をスループットとして評価した.

ベンチマークに使用したサーバー(以後,「サーバーA」と表記)とデータベースの仕様を**表1**に示す.

入力オーダーは, **表2**に示す値を用いた.

表1. ベンチマーク環境

サーバーA	
H/W	IntelliStation A Pro 6224-3J7 Dual Xeon 2.2GHz w/ 4GB memory
OS	Windows Server 2003 x64 Enterprise Edition SP1
JVM	IBM Java VM 1.4.2
データベース	
H/W	IntelliStation M Pro 6850-50J Dual Opteron 248 2.2GHz w/ 8GB memory 40GB 拡張IDE (7200rpm) Ultra ATA-100
OS	Windows Server 2003 x64 Enterprise Edition SP1
DB	DB2 UDB Enterprise Server Edition V8.2 (V8.1 FP9)

表2. 入力オーダーの設定値

売買指定値段	100 + gaussian()*×10
希望取引量	200 + gaussian()×100

* gaussian() : 平均0.0, 標準偏差1.0のガウス(正規)分布をとる乱数

一度のマッチング処理に投入する新規オーダー数は256ずつ, 売買の比率はそれぞれ50%とする.

ベンチマーク結果を**図9**に示す. 新アーキテクチャによる売買処理では, 通常のJ2EEクラスタリングによる売買処理の4倍以上のスループットが得られた.

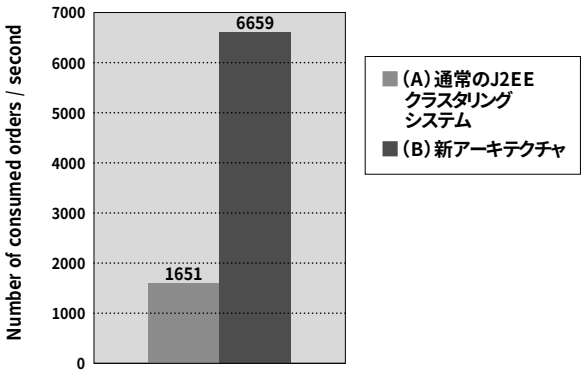

図9. ベンチマーク結果

通常のJ2EEクラスタリングシステムで処理を行った場合(A)は, データベースのCPUがボトルネックとなった(**表3**).

表3. CPU平均使用率

	サーバーA	データベース
(A) 通常のJ2EEクラスタリング システム	5%	95%
(B) 新アーキテクチャ	5%	10%

新アーキテクチャで処理を行った場合(B)は, データベースのディスク書き込みがボトルネックであり, データベースのCPU使用率はまだ余裕があった. 複数のディスクを使用するなどの方法で, ディスクの書き込み性能を向上すれば, スループットの更なる向上が見込まれる.

5. おわりに

本論文で示した新アーキテクチャは, J2EEクラスタリングシステムの利点を保ちながら, アプリケーション開発の負荷は最小限に抑えることができ, データベースのボトルネックを解消できるアーキテクチャである.

新アーキテクチャは, 処理データをパーティションに分けることができるならどんなシステムにでも適用可能である. J2EEクラスタリングシステムとアプリケーションパーティショニングの組み合わせは, システム構成におけるキラーアーキテクチャとなりうる可能性を秘めている.

この新アーキテクチャを実現するにあたりその必要構成要素であるパーティション制御コンポーネントは, WebSphere Application Server XD 製品の WPF 機能として今日利用可能である.

謝辞

検証システムを実装するにあたり, J2EE環境における非同期ビーンを使用したスレッドプールの実装方法について助言くださったUS IBM Rochester研究所 Billy Newport氏に感謝いたします. 最後に, このようなチャレンジングな仕事を与えてくださりまた多方面において各種アドバイスを下さった清水敏正DEに感謝いたします.

参考文献

[1] *Distributed Application Partitioning,* http://www.research.ibm.com/dap/, 1999
[2] Jim Gray, トランザクション処理 概念と技法, 日経BP社, ISBN 4822281027, 2001
[3] *Dependency Injection,* http://www.martinfowler.com/articles/injection.html, 2004
[4] Rod Johnson, 実践J2EEシステムデザイン, ソフトバンクパブリッシング, ISBN: 4797322888, 2003
[5] Deepak Alur et al. J2EEパターン, ピアソン・エデュケーション, ISBN 4894714345, 2002
[6] *WebSphere Application Server InfoCenter,* http://publib.boulder.ibm.com/infocenter/wasinfo/index.jsp, 2004
[7] *EJB2.0 Specification,* http://java.sun.com/products/ejb/docs.html, 2001
[8] Richard MonsonHaefel, Enterprise JavaBeans, オライリー・ジャパン, ISBN 487311098X, 2002
[9] 石畑 清, アルゴリズムとデータ構造, 岩波書店, ISBN: 4000103431, 1989

日本アイ・ビー・エム株式会社
ITスペシャリスト
夷藤 勇人 Hayato Itoh

[プロフィール]
1999年, 日本IBM入社. 2001年からJ2EEアプリケーションサーバーの技術支援を担当. 2002年には, 米国IBM ラーレイ研究所にて, SpecJAppServer2002の開発に従事. 著書に「Eclipse パーフェクトマニュアル」などあり.
ihayato@jp.ibm.com

日本アイ・ビー・エム株式会社
東京基礎研究所 副主任研究員
根山 亮 Ryoh Neyama

[プロフィール]
1999年, 日本IBM入社. 東京基礎研究所にて, Webサービス技術, アプリケーションサーバー上のトランザクショナルキャッシュ技術などの研究に従事. 著書に「Building Web Services with Java」などあり.
neyama@jp.ibm.com