Web n 層アーキテクチャとクラスタリング技術

アーキテクチャ研究グループ

１．はじめに

技術研究室、インフラ構築本部では、Java アプリケーションサーバーに関する調査・研究活動を行っている。本稿では、1999年９月から10月にかけて実施した n 層クラスタリング実験について報告する。

２．技術動向－Web n 層とクラスタリング

2.1 C／S２階層から Web n 層へ向かう技術動向

アプリケーション・アーキテクチャは、メインフレーム指向からデスクトップ指向を経て、ネットワーク中心の世代へ移行している。デスクトップ指向のパラダイムは、２層 C／S アーキテクチャによって実現されてきたが、このアーキテクチャは２つの問題点を持っている。１つは TCO（Total Cost of Ownership）に関わる課題で、クライアントソフトウェアの配布・インストール、バージョンアップなどの保守負荷が高いことである。もう１つは実行性能に関する問題で、クライアントとサーバーの間に十分高速なネットワークを必要とする点である。広域で稼動するシステムの場合、ネットワーク速度がボトルネックとなり実行性能が出ないという問題が発生する。

Web n 層アーキテクチャは、２層 C／S が抱えるこれらの問題に対して解を与えるアーキテクチャとして登場した。Web n 層アーキテクチャは、ユーザーインタフェース（UI）、ビジネスロジック（BL）、データベース（DB）の３つの層から構成される。BL 層は Web サーバーとアプリケーション用サーバーに分離するなどしばしば多階層になるため、一般化して Web n 層アーキテクチャと呼ばれる。UI 層のメインコンポーネントとして Web ブラウザを用いることで thin クライアントを実現、保守負荷を低減する一方で、UI 層と BL 層の間のプロトコルは低速ネットワークでも稼動する HTTP となるため、２層 C／S の問題を回避することができる。

Web n 層アーキテクチャの中核コンポーネントは、Web アプリケーションサーバーと呼ばれる製品群である。著名なプロダクトとしては BEA WebLogic、IBM WebSphere、Netscape／Sun Alliance の iPlanet などが挙げられるが、これらは BL 層のエンジンとしての役割に加え、メインフレームや ERP などのレガシーシステムとインターネットを仲介するハブの機能も担おうとしている。また、Enterprise Java Beans や CORBA、MS DCOM などの標準をベースとしているため、アプリケーションは分散コンポーネント・システムとして構築される。

今後３年間を考えた場合、TCP／IP ネットワークをベースとする企業システムは、クライアントとしてブラウザを利用する形態の Web n 層システムが主流となり、その後ネットワーク・インフラの充実に伴い、UI 層まで含めた分散オブジェクト環境が構築されていくと予測される。

2.2 非システム的なクラスタリング技術

クラスタリングは、複数のコンピュータを１つのコンピュータであるかのように結合することにより、負荷分散とフォールトトレランスを実現する技術である。結合されたコンピュータの集合はクラスタ、各コンピュータはノードと呼ばれる。一般的に、クラスタリング技術はシステム（OS＋ハードウェア）レベルで実現される機能であり、メインフレームを頂点として、UNIX、NT がキャッチアップを目指している。しかし、システムレベルのクラスタリングは、OS のコアに絡む高度な技術であるため、その性能を向上させることは容易ではない。

一方、Webアプリケーションの急速な普及は、システムレベルとは別種のクラスタリング技術を発展させた。Internetアプリケーションでは、広告１つでアクセス頻度が10倍になることも珍しくない。また、“24hours×7days”の無停止運用が当然のように要求される。特に、Web n層アーキテクチャが主流となりつつある現在、ますますミッション・クリティカルなサービスがインターネットを介して提供され、そのようなシステムの停止は巨額の損失に結び付く。このような、高スケーラビリティ、高可用性の要求から、DNS－Round Robin、Dispatcher方式によるWebサーバーのクラスタリングやアプリケーションサーバーによるソフトウェアベースの負荷分散・フォールトトレランス機構が発展した。これらの非システム的なクラスタリング技術は、OSレベルのクラスタリングに比較して利用が容易であり、現在、急速に普及しつつある。

３．実験の概要

3.1 実験の主目的

以上のようなWeb n層アーキテクチャへの変遷、非システム的なクラスタリング技術の普及動向を踏まえ、アプリケーションサーバーの技術検証実験を行った。主要な目的は次の４点である。

⑴ アプリケーションサーバーのリソースプーリング機能の検証
⑵ n層アーキテクチャが実行性能にもたらすインパクトの把握
⑶ クラスタ技術が実行性能にもたらす効果の把握
⑷ チューニングに関する技術獲得

3.2 実験システムの構成

100MbpsのTCP／IPネットワーク上に、６台のサーバーマシンと２台のクライアントマシンを配置した実験環境を構築した（図１）。

⑴ Web／APP Server

ハードウェア	PentiumⅢ 550MHz、768MB（２台） PentiumⅡ 300MHz、224MB（２台）
OS	Linux（kernel 2.0.36、2.2.9）
ソフトウェア	BEA WebLogic Server 4.03

⑵ DB Server

ハードウェア	PentiumⅡ 450MHz、512MB（１台）
OS	Linux（kernel 2.2.12）
ソフトウェア	Linux上で稼動するDBサーバー

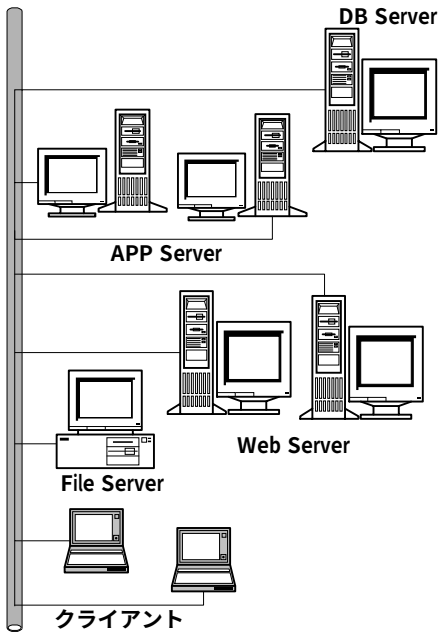

図１　実験システムの構成

⑶ File Server

ハードウェア	Pentium 60MHz、40MB（１台）
OS	Linux（kernel 2.0.36）

共有ディスクとして設けてWebLogic Serverをセットアップし、Web／APP ServerからNFSでマウント

⑷ クライアント

ハードウェア	PentiumⅢ 550MHz、128MB（１台） PentiumⅡ 450MHz、128MB（１台）
OS	Microsoft Windows NT Workstation 4.0
ソフトウェア	Microsoft Web Application Stress Tool 1.1

3.3 計測対象アプリケーション

Transaction Processing Performance Councilは、Web Commerceサイトのベンチマーク仕様であるTPC－Wを作成している。現在ドラフト段階であるが、静的・動的ページアクセス／商品検索／ショッピングカート処理／決済処理などを織り交ぜたテストスイートになっている。今回の実験では、このTPC－W（ドラフト）を一部参考にして、商品一覧処理（DB検索）、ショッピングカート一覧処理（DB検索）、購入処理（DB更新）の３つの機能で構成される独自の仕様を作成、実装した。われわれの作成したテスト用アプリケーションは、ユーザーからは３つのWebページとして見える（図２、図３、図４）。

このアプリケーション仕様に対して、異なるJava技術を組み合わせた複数パターンの実装を作成した。どのパ

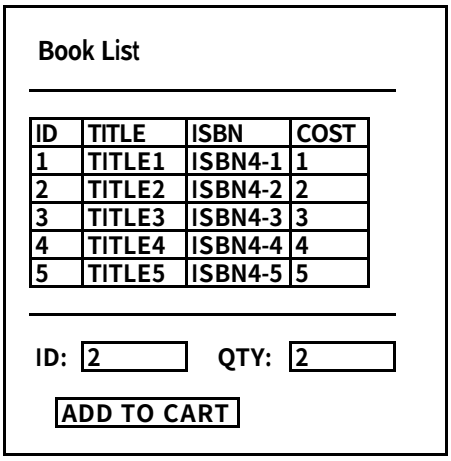

図２　ページ１：商品一覧表示、商品をカートに入れる入力エリアとボタン

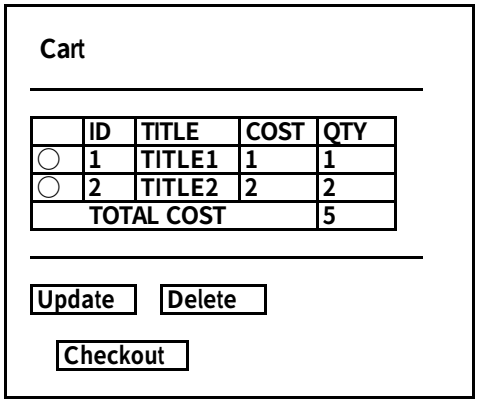

図３　ページ２：カート内の商品一覧と購入ボタン

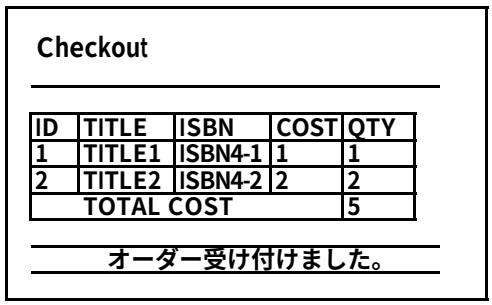

図４　ページ３：購入商品の一覧表示

ターンもユーザーインタフェースは同じであるが、サーバー側アプリケーションの内部構造が異なる。

⑴ パターン１：Servlet 版
　Java Servlet[*1]技術のみを使用した単層構造

⑵ パターン２：Servlet＋Session Bean 版
　Java Servlet と Enterprise Java Beans（EJB）[*2]の Session Bean[*3]技術を使用した２層構造

⑶ パターン３：Servlet＋Session Bean＋Entity Bean 版
　Java Servlet、EJB の Session Bean と Entity Bean[*4]技術を使用した２層構造

3.4 テストスイート仕様

テストツールとしては、Microsoft Web Application Stress Tool を使用した。

計測と分析の対象データとしては、サーバーが１秒間当たりに処理したリクエスト数である RPS（Request per Second）を主として用いた。同時アクセスユーザー数をツールで900まで疑似的に作り出して、各４分間（ウォームアップとクールダウンに各30秒）ずつのロードテストを行った。

疑似ユーザーはリクエストを発行し、そのレスポンスが得られると次のリクエストを発行するというループを実行する。サーバーサービスが無限の処理性能を持っているならば、理論的には同時アクセスユーザーを増加させるとその分 RPS は増加するはずである。しかし、現実のサービスではリクエスト１件あたりのレスポンスタイム（処理時間）が増加するため、RPS はユーザー数の増加ほどには増えない。ユーザー数をさらに増加させていくと、サーバーサービスの限界に近づき、RPS は頭打ちになる。

＊１）Servlet：サーバー側 Java 技術の中では CGI の置き換え技術に相当するもの。標準入出力に対する処理がオブジェクト操作として処理可能で、Java の豊富なライブラリや API が利用できる。ユーザーインタフェース処理（HTTP の動的生成など）とビジネスロジックの分離には向いていない。

＊２）Enterprise Java Beans：サーバー側で動作する Java オブジェクトのための仕様。Session Bean と Entity Bean という２種類のアプリケーション記述の標準仕様を中心に、EJB Container（Bean の実行環境）、他の Java 2 Enterprise Edition API と関連する部分の仕様化がなされている。

＊３）Session Bean ： EJB コンポーネントの一種。クライアントからの接続（Session）ごとにインスタンスが用意されるタイプのオブジェクトを記述するための仕様。

＊４）Entity Bean ： EJB コンポーネントの一種。クライアントから１つのインスタンスが共有される形になるタイプのオブジェクトを記述するための仕様。

４．実験結果と分析

4.1 DB 接続プールと実行性能

4.1.1 概要

DB 接続プールとは、JDBC ドライバやアプリケーションサーバーなどがあらかじめ DB 接続を用意（プール）しておき、複数のアプリケーション・インスタンス間で共有・再利用することによって、全体的な DB 接続コストを抑える手法である。

計測対象アプリケーションは Servlet 版を使用し、設定ファイルにより、DB 接続プールを使用するかどうかを切り替えるように実装した。各 Servlet のメソッド内で、DB 接続、SQL 処理、DB 切断が発生する。DB 接続プールは、WebLogic の DB 接続プール機能を使用し、JDBC ドライバは単純な Type４（Thin 型）を使用した。

実際のシステム開発では、標準的に DB 接続プールが使用されるだろうが、キャパシティ・プランニングや DBMS のチューニング、ライセンス・コストの観点から、どの程度の差が出るかを計測しておくことにした。

4.1.2 仮説

DB 接続プールが、パフォーマンス向上策の中で最も効果が高いと一般的に言われている。したがって、計測結果はプールを使用した方が圧倒的に優れているはずである。

4.1.3 計測結果

計測結果を図５に示す。

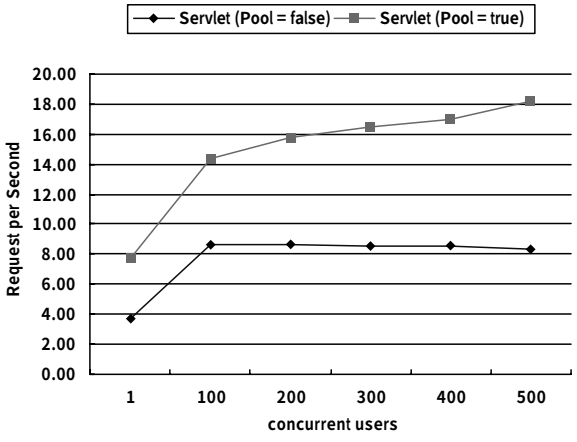

図５　DB 接続プール実行性能の計測結果

4.1.4 分析

仮説通り、プールを使用した実装が優れていた。プールを使用していない実装では、100ユーザー以降、RPS が増加していない。これはサーバーサービスの処理性能が限界に達した状態である。これほど早く頭打ちが発生することは予想していなかった。この結果は、DB 接続がボトルネックになる可能性が最も高いことを示唆している。プールを使用した場合には、個々のリクエストに対する応答時間は長くなりながらも RPS は伸び続けている。

4.2 n 層アーキテクチャと実行性能

4.2.1 概要

サーバー側アプリケーションを構築するための技術は、多数存在する。機能を分割するために組み合わせて使用することが望ましい場合もあれば、そうでない場合もある。今回は、Java Servlet、Enterprise Java Beans の Session Bean、同じく Entity Bean の３つの技術を使用し、組み合わせて多層化した場合の比較を行った。

計測の目的は、実行性能・生産性・保守性のバランスを考慮した Java 技術の組合せを見い出すことである。実行性能の観点からは、Servlet だけの単純な構成が望ましいが、生産性や保守性の観点からは、ユーザーインタフェース処理とビジネスロジック処理を分離実装できる Servlet＋EJB 構成が望ましい。

計測対象アプリケーションは Servlet 版、Servlet＋Session Bean 版、Servlet＋Session Bean＋Entity Bean 版の３つで、全てのパターンで DB 接続プールを使用した。

4.2.2 仮説

Servlet 版が最も高速で、次に Servlet＋Session Bean 版、実行性能が最も低いのは Servlet＋Session Bean＋Entity Bean 版と推測した。Servlet 版はオブジェクト呼出しが発生しない単純な構造をしている。一方、Session Bean を使うとオブジェクト呼出しが発生する上、セッションごとにインスタンス作成、もしくは再利用のオーバーヘッドが発生する。さらに、Entity Bean はインスタンスが共有されるため同時実行性が低いと予想される。

4.2.3 計測結果

計測結果を図６に示す。

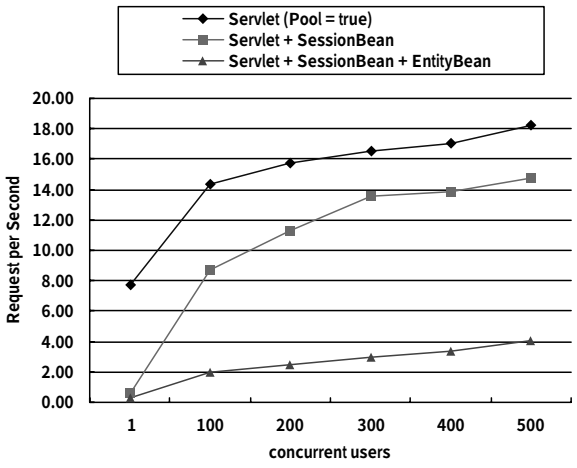

図６　n 層アーキテクチャ実行性能の計測結果

4.2.4 分析

仮説通り、パフォーマンスはServlet版が最も高速で、続いてServlet＋Session Bean版、Servlet＋Session Bean＋Entity Bean版の順となった。

Entity BeanはRDBMSアプリケーションで頻繁に使用するJOIN処理が難しいことから、RDBベースの開発では広く利用されることはないと考えられる。そこで、特に今回は、Servlet版と比較して、Servlet＋Session Beanの実行性能がどれほど低下するかに注目する。

Servlet版はServlet＋Session Bean版と比較して、20%強（300、400、500ユーザーに対して）の性能アドバンテージを持っている。このアドバンテージをどのように評価すべきかは一概には決定できない。しかし、UI層とBL層を分離することによって得られる開発・保守上のメリットは決して小さなものではない。実行性能はマルチプロセッサなどのハードウェア増強や後述のクラスタリング技術を用いて比較的容易に向上できる一方、ソフトウェア工学上、開発生産性・保守性の向上は容易な課題ではない。さらに、今後EJB仕様が普及することにより、再利用可能なコンポーネントのマーケットが立ち上がる可能性もある。そのようなEJBコンポーネントに関する動向として、IBMのサンフランシスコ・プロジェクトによるEJB対応やBEA社によるEC用コンポーネント・ライブラリの開発などがすでにアナウンスされている。

以上の論拠により、この程度の実行性能差であるならば、複雑性を持ったビジネス・アプリケーションに対してはUI層とBL層を分離したServlet＋EJB構成を基本アーキテクチャとして推薦したい。

4.3 クラスタ構成と実行性能

4.3.1 概要

前節までは単一マシンにアプリケーションサーバーを配置してのテストであった。本節はクラスタ構成による実験である。

WebLogicは、Servlet層ではDNS－Round Robinを、EJB層ではネーミングサービスをベースとするクラスタ機構を実装している。EJB層のクラスタリングでは、セグメント内のマルチキャストによりハートビート機構が実現されている。クラスタノードとして起動したサーバーが何らかの理由でダウンした場合、ネーミングサービスが他のサーバーにクライアントを接続し、フォールトトレランスを実現する。

実験では、同一構成のサーバーマシン２台を並列に配置し（図７）、Servlet版とServlet＋Session Bean版の２パターンの実行性能をテストした。使用したテストスイートは前節までと同じである。単一マシン構成と２台構成の実行性能を比較することにより、実際にリニアな性能向上が見られるかを検証する。

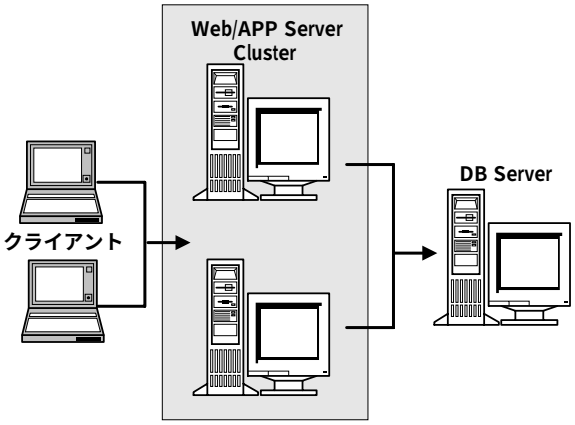

図７　クラスタ構成図

4.3.2 仮説

今回のテスト対象システムでは、セッションレベルのフォールトトレランスを組み込んでいない。セッションレベルのフォールトトレランスを実現する方法は、セッション情報の共有化の手法により幾通りか考えられる。例えば、DB（RDB、ODB）、もしくは共有ファイルシステムへの格納などの方法があるが、それぞれ情報共有化のためのオーバーヘッドは異なる。そのため、今回は、セッションレベルのフォールトトレランス機構は捨象し、純粋に実行性能のみを計測対象とした。したがって、データベースやネットワーク負荷、DNS・NFSサーバーへのアクセスがボトルネックとならない限り、１台構成に比較して、ほぼ倍に近い実行性能が期待される。

4.3.3 計測結果

以下のグラフ（図８）は、DB接続プールを使用したServlet版の計測結果である。下側のラインは１台構成、上側は２台構成のクラスタである。１台構成ではユーザー数を700以上にすると、エラーが発生した。これはLinux

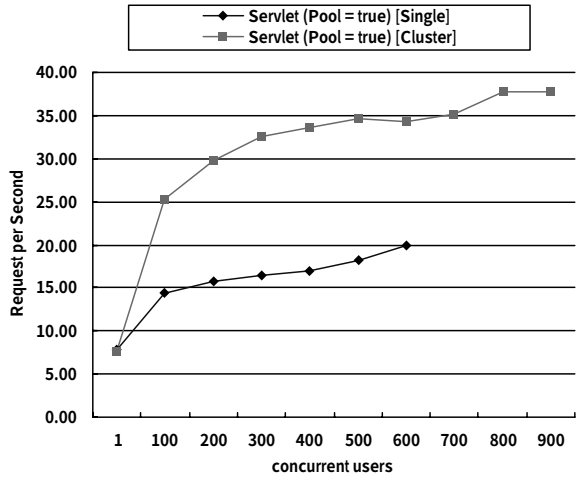

図８　クラスタ構成の実行性能計測結果（Servlet版）

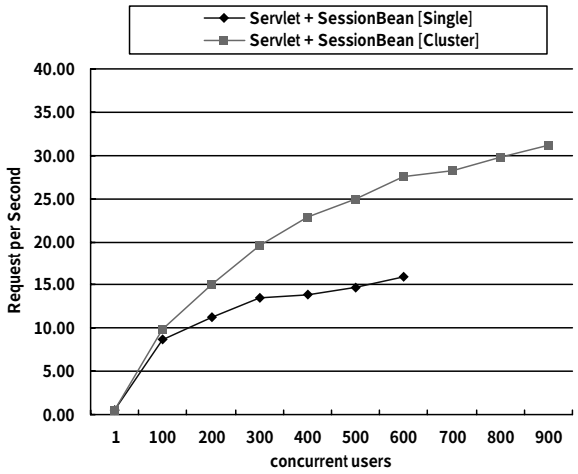

図９　クラスタ構成の実行性能計測結果（Servlet＋Session Bean 版）

の最大オープンファイル数に関する制約（後述）からくるものと推測される。カーネルの再構築により対処できるが、今回の実験では計測をそこでストップした。クラスタ版では900までエラーを発生せず（計測は900まで）、実行性能も漸次向上している。

次に、Servlet＋Session Bean 版の結果を示す（図９）。Servlet 版と同じく、単一マシン構成では700以上のユーザーアクセスに対しエラーが発生する一方、クラスタ・モードでは900ユーザーまで実行性能は向上する。

4.3.4　分析

仮説通り、２つのパターンとも、クラスタ構成の方がほぼ２倍に近い実行性能を実現している。Servlet 版においては明らかであるが、Servlet＋Session Bean 版においても、RPS は倍増している。例えば、600ユーザーのクラスタ・モードでは27RPS 程度の値をマークしている。Servlet 層で DNS－Round Robin によりまず負荷分散されるため、この時に比較すべき対象は同程度の負荷が掛かるシングル・モードの300ユーザーの場合であり、その値は約14RPS である。

なお、クラスタ・モードにおいて計測中の環境におけるネットワーク・トラフィック、DB・NFS サーバーの負荷を調べたが、いずれもボトルネックとはなっていない。

５．高負荷に耐える n 層システムの構築

5.1　ボトルネックの発見と解消

前章では最終的に完成したサーバーサービスに対するロードテストの分析を行った。ここではサーバーサービスの構築で得られた知識を過程に沿って整理する。

5.1.1　構築過程について

今回の実験では、単純な構造のコードから開発を進めた。開発と同時並行的にロードテストを行い、結果に基づき、実行環境とアプリケーションの設定変更、コード修正、ボトルネックの検出と解消というプロセスを踏んだ。

5.1.2　Servlet 版構築時に発生したボトルネック

DB 接続プールの実験で述べたように、最初に発生したボトルネックは DB 接続処理であった。この問題は、DB 接続プールを使用することにより容易に解消した。

5.1.3　Session Bean 版構築時に発生したボトルネック

同時アクセスユーザー数を増加させるとエラーが発生するという問題が生じた。ユーザー数が少ない時には全く問題が発生しないため、単体テスト段階では顕在化しなかった。さらに、一度不安定になるとリブートが必要な状態となった。

Linux では、各プロセスがオープンできるファイル数がカーネルリミットとして決定される。使用していたカーネルでは1024にチューニングされていたが、これを超えるソケットがオープンされることが不安定状態の原因と考えられた。カーネルの再構築も検討したが、Servlet から Session Bean 呼出し時のコードを変更することで問題解決した。今回の構成では、単一マシン（すなわち同一 JVM）上に Servlet と Session Bean 両方が配置されていたにも関わらず、当初ネットワークを介してコールしていた。この呼出し部分のコードを同一 JVM 用に修正することで、解決することができた。

その後、ボトルネックは実行環境に移動した。WebLogic の Web Server 部で待ちが発生し、クライアントのタイムアウトが頻発するようになった。これは、WebLogic のタイムアウト設定を長く取ることにより解消できた。

5.1.4　Entity Bean 版構築時に発生したボトルネック

Entity Bean 自身が極端なボトルネックとなった。

Entity Bean の開発では、設定ファイルに DBMS 中のテーブルとの対応を記述し自動マッピングに任せる手法を採用した。コードはほとんど記述しなかったため、チューニング・ポイントは Entity Bean を呼出す側となる。追加テストとして、呼出し側のコードをチューニングした場合と比較したが、大きな差は出なかった。

また、今回開発したテスト用アプリケーションでは Servlet を３つ作成しているが、全件検索メソッドを呼出

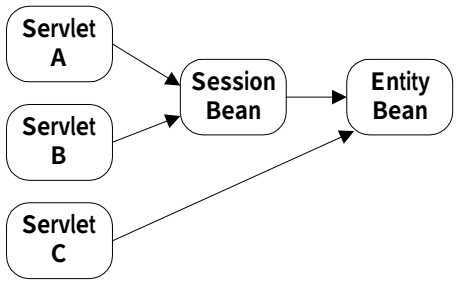

図10　Entity Bean の呼び出し

している Servlet が特に遅かったので、個別検索メソッドをループで回すコードと比較したが、これも大きな差は出なかった。

全件検索を実行している Servlet（図10の C）は Entity Bean を直接呼出しているのに対し、その他の Servlet は Session Bean 経由で Entity Bean を呼出している。

Session Bean は同一接続内であれば再利用されるので、ここがパフォーマンスの差の原因になっているという可能性が考えられたが、これ以降は作業優先度の関係で深く追究できなかった。

残念ながら、今回の実験では Entity Bean の有効な利用方法について明確な結果を出すことができなかった。ODB とのマッピングなども含めて、今後の課題としたい。

5.1.5 構築作業を振り返って

ボトルネックは最初に表面化する部分が問題とは限らない。解消すれば必ず別の個所がボトルネックになる。それが極端かどうか、ハードウェアの増強などで対処できるかどうかを基準に妥当性を判断する必要がある。

また、極端なボトルネックであっても、負荷が軽い状況では表面化しないことがある。単体レベルでの負荷テストクリア基準を設定し、仕様が変更される都度テストを行う必要性を感じた。

今回のような多数のユーザーが同時アクセスを行うアプリケーションでは、少しのコードの違いが運用時に数百倍の差となって顕在化する。そのため、様々な角度からチェックを行う必要性を認識した。特に重要と考えられる点を以下にまとめる。

⑴ 単体レベルでの負荷検査クリア基準を設定し、仕様変更時には必ずテストを実行すること。

⑵ セッションごとのリソース消費量の抑制に留意すること。

⑶ あらかじめフレームワークを決定し、徹底的なエラー処理を実装すること。

5.2 チューニング・ポイント

今回の技術検証実験により、チューニングに関してわれわれが得たノウハウは、次のようなものである。

⑴ DB 接続プール周辺のチューニングを最初に実施する。

⑵ ハードウェアの増強を検討する。

…計測中、ネットワーク負荷に比べ、CPU 負荷が常に非常に高かったことから、今回は行っていないが CPU 増設の効果が大きいと考えられる。

⑶ アプリケーションのフレームワークの設計を、綿密にする。

…開発後にパフォーマンスの観点からコードを広範囲に修正するのはコストが掛かるので、ベースとなるフレームワークを入念に作り込んで品質を確保し、チューニング・ポイントを分離して実装すべきである。また、コード全般に関しては、同時実行性を強く意識して設計・実装する必要がある。具体的には、

・排他制御が必要なリソースへのアクセスを最小限に抑える。

・実行環境の負荷分散が効果的に働くよう、なるべく状態情報を持たない（Stateless）範囲を広げる。

なお、今回は、トランザクションの並列実行性を高めるために、連番を取得する処理を回避することも行っている。

⑷ OS、アプリケーション実行環境、DBMS の各設定をチェックする。

極論すれば、環境設定全体、コード全体がチューニング・ポイントになり得る。しかし、実際には影響や効果の大きな項目に焦点を絞り、ある程度の試行錯誤を覚悟しながらチューニングを実行していくことになるだろう。

環境設定項目には次のような種類がある。

・同時使用可能リソース（connection 数、同時ユーザー数）

・限界リソース（connection 数、同時ユーザー数）

・待ち行列（Timeout 関係）

・キャッシュ

アプリケーション実行環境、アプリケーション、および DBMS にまたがって、これらの項目を整理すべきである。さらに、これら項目間での設定矛盾をチェックする必要がある。例えば、クライアントに近い側のタイムアウトが短い場合、その後ろ側でタイムアウトを延ばしても効果はない。

今回の実験環境を例として、主要なチューニング・ポイントを以下に示す。

5.2.1 OS と Java VM に関するパラメータ

max_files	UNIX における、プロセスがオープンできるファイルディスクリプタ数
JVM ヒープサイズ	初期サイズ(java -ms オプション)、最大サイズ(java -mx オプション)を指定
JVM 非同期 GC	java-noasyncgc オプションを指定

5.2.2 実行環境全体（WebLogic）のチューニング・ポイント

接続タイムアウト	Weblogic.properties 中の weblogic.login.readTimeoutMills を指定
同時実行スレッド数	weblogic.properties 中 の weblogic.system.executeThreadCount を指定
JDBC 接続プールの初期接続数、最大接続数	weblogic.properties 中 の weblogic.jdbc.connectionPool.プール名の設定中の initialCapacity、maxCapacity を指定

5.2.3 アプリケーション(EJB)のチューニング・ポイント

EJB は実行環境全体の設定情報とは別に、Bean ごとの設定情報も持っている。特に関連する設定項目は、以下のようなものである。

キャッシュ、およびフリープールにプールする Bean 数	各 Bean の DeploymentDescriptor.txt 中の maxBeansInCache、maxBeansInFreePool を指定
書き戻し処理の最適化（Entity Bean のみ）数	各 Bean の DeploymentDescriptor.txt 中の isModified を指定し、Bean に適切なメソッドを実装

5.2.4 DBMS のチューニング・ポイント

今回の仕様は DBMS 側から見ると、OLTP に近いものとなっている。そのため、次の観点でチューニングを行った。

⑴ 内部でのバッファなど、リソース割り当ての競合を抑えた。

⑵ １接続当たりのメモリ使用量を抑えた。

⑶ ディスク I／O でボトルネックが発生しないように、負荷を複数デバイスに分散する設定を行った。

⑷ テストデータの量が小さかったため、データを DBMS のキャッシュ上に載せて更新時以外のディスク I／O を抑えた。

⑸ JDBC の PreparedStatement を使った SQL の共通化を図り、SQL の解析負荷を減らした。

６．おわりに

以上、アプリケーションサーバー実験の概要とそこで得られた知見について報告した。実験では、異なる Java 技術を同一マシンに配置した論理的な n 層構造を主題とし、実行性能の比較やクラスタリングのインパクトを評価した。今後は、Java Servlet や EJB を動かすマシンを別々にした物理的な n 層構造とフォールトトレランスを研究課題として、アプリケーションサーバー実験を行っていく予定である。

謝辞

今回の実験では、住商エレクトロニクス・ビジネスソリューション第一事業部の田村俊明氏から実験の全体方向に関する貴重なアドバイスを、また同・刀根誠治氏からは WebLogic と Java に関する詳細な技術支援を戴いた。両氏に心からの感謝を申し上げたい。

＜参考文献＞

１．『CAC インフラ白書99』CAC 技術本部1999

２．Web Site Test Tools and Site Management Tools, by Rick Hower,
http : //www.softwareqatest.com/qatweb1.html

３．Clustering Multiply and Conquer, by Eric Brewer (Inktomi Corp.),
http : //www.data.com/tutorials/clustering.html

４．Dynamic Load Balancing on Web-server Systems, by Valeria Cardellini, Michele Colajanni, and Philip S. Yu, IEEE Internet Computing, Vol. 3, No. 3, May／June 1999

５．TPC BENCHMARK W Public Review Draft Specification Revision D-5. 0, by Transaction Processing Performance Council
http : //www.tpc.org

アーキテクチャ研究グループ　メンバー

CAC 技術研究室室長	稲垣　陽一
CAC 技術研究室	田中　佳美
CAC IT コンサルティング室	金子　嘉夫
IT コンサルタント	八津谷太郎氏