平成17年度 修士論文

Webアプリケーションサーバの性能向上を目指した動作情報の動的解析とパラメータチューニング

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0450031　南野 久美子

指導教員　多田 好克　　教授
星 守　　教授
前川 守　　教授

提出日　平成18年1月31日

目 次

図 目 次

表 目 次

第 1 章

はじめに

近年、パソコンの普及などにより、インターネットの環境は身近なものとなっている。携帯や、PHS からもインターネットに接続が可能な現在、インターネットの利用者数は非常に多い。技術の普及により、静的なコンテンツだけでなく、クライアントから送られてくる情報によって、動的にコンテンツを生成する Web サーバも増えてきている。おのずとインターネット上の Web サーバの利用率も高くなり、Web サーバの稼動時の安定性や処理能力の高さが重要となってきている。

このような背景により、Web サーバの管理者が、より高い処理能力をだすために Web サーバのパラメータチューニング等を行う必要性が生じている。しかし、サーバのパラメータチューニングは作業時間がかかるため、管理者にとって負担が大きい。パラメータの設定を誤れば、現状よりも処理能力がさがったり、サーバがダウンしたりする危険性もある。

そこで本研究は、Web サーバにおけるパラメータチューニングを動的に行うシステムを設計、実装する。本システムを利用することにより、管理者のパフォーマンスチューニングの作業に関する負担を削減し、サーバのパフォーマンスを向上させるのが本研究の目的である。

本論文では、次のような構成となっている。第 2 章では、管理者による Web サーバのチューニングの作業負担を述べ、本研究の目的を述べる。第 3 章では Apache サーバのチューニング手法と、チューニング対象のパラメータのサーバへの影響について説明する。第 4 章では、Web サーバ等のシステムのパフォーマンス改善に

関する関連研究について述べる。第5章では本システムの設計方針と、各パラメータのチューニングに必要となるサーバの動作情報について述べる。第6章でシステムの実装と、パラメータの変更値を求める方法についての予備実験について、および実際のチューニング手法を述べる。第7章では、本システムを導入したサーバに対して、ベンチマークテストによる評価実験を行い、本システムの有効性を検証した。最後に第8章でまとめとする。

第 2 章

背景と目的

2.1 背景

近年、インターネットの中心的サービスである WWW(World Wide Web) は爆発的に発展してきている。それにともない、インターネットの利用者数の増加、Web 利用によるビジネスチャンスの増加など、Web サーバが安定したサービス供給を行う必要性が高まっている。こうしたことから Web サーバの管理者は、Web サーバのパフォーマンスをあげるためのパラメータチューニングを重要視する傾向がある。

Web サーバにおいて、より少ない資源環境でより高速なレスポンスを確保するために、Web サーバの管理者によるパラメータチューニングは欠かせない。しかし、パフォーマンスをあげるためのチューニング作業には、図 2.1 のような行程を繰り返し行う必要がある。サーバの処理能力を現状よりあげる作業をするには、管理者はまず現状のサーバの稼動状態の目安となるデータを取得する。そのデータをもとに分析を行い、サーバ環境のボトルネックを探す。そして、そのボトルネックとなっている部分を緩和させるべく、適切なパラメータを適切な値に変更する。そして、変更後のサーバの状態を監視し、サーバの処理能力が良くなったかなど、変更の妥当性を検証する。

管理者によるこれらの作業は、多くの時間を費やし、管理のコストが増える要因でもある。また、Web サーバへの負荷は常に変化するものであり、高いパフォーマ

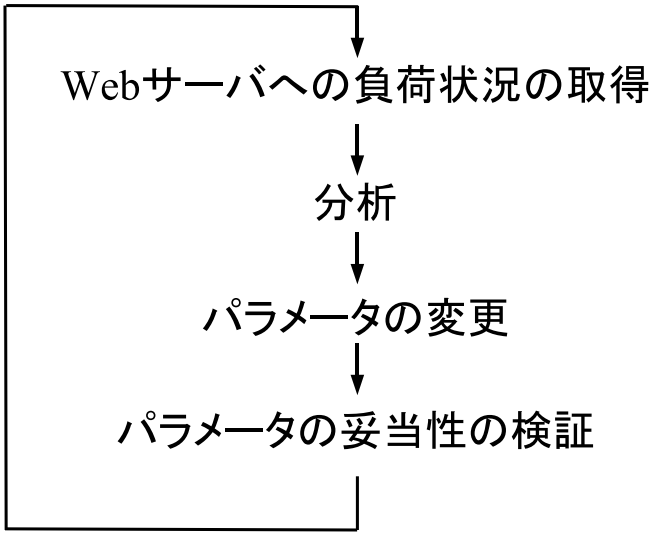


図 2.1: パラメータチューニング作業の流れ

ンスを維持するためには 1 度のチューニング作業で対応できるものではない。パラメータの設定を誤れば、Web サーバのパフォーマンスは現状より低下してしまう。

2.2 目的

本研究は、Web サーバにおける自動的なパラメータチューニングを行うシステムを提案、構築する。前述の背景を踏まえ、本システムを用いることによって、サーバ管理者の作業負担を最小限に抑えたパフォーマンスチューニングを可能にすることが目的である。本研究の提案するシステムは、サーバ管理者の作業負担を減らすために、サーバの負荷状況に応じて動的にパラメータチューニングを行うものである。

第 3 章

Apacheのチューニング

3.1 Apache

本システムでは、チューニング対象の Web サーバを Apache とする。対象サーバに Apache を選んだ理由は次の 2 点である。1 点目は Apache のソースが無償で公開されているので、内部構造を把握しやすいという点。2 点目は、現在実稼動している Web サーバの中で Apache が大きなシェアを占めているため、本システムを実際に使用できる場が多いという点である。Netcraft[1] による調査では、Web サーバの稼動数の 3 分の 2 を Apache が占めている (表 3.1)。

表 3.1: 実稼動サイト数 (2006 年 1 月現在)

開発元	2005 年 12 月稼動数	割合 (%)
Apache	52025380	69.97
Microsoft	15557786	20.92
Sun	1881861	2.53
Zeus	577384	0.78

また、使用する Apache のバージョンは Apache2.0 とする。現在 Apache には

Apache1.3 系と Apache2.0 系がある。今後、2.0 系のサーバが主流になることが考えられ、Apache1.3 系で稼動しているサーバも 2.0 系に移行していくであろう。よって、本研究では Apache2.0 系に対応するシステムを構築する。

以前の Apache1.3 系から Apache2.0 系にバージョンアップしたことで、Apache の内部構造に様々な変更点がある。内部の構造体など大きく仕様が変わってはいるものの、Web サーバとして使用する面での差はほとんどない。Apache2.0 系の変更点および特徴を以下に述べる。

- 内部の API が変更。プラットフォーム非依存な関数群を独立させ、APR(ApachePortableRuntime) として導入されている。プラットフォーム依存の部分は、この APR が全て吸収して動作する仕組みになっている。
- プロセスベースからスレッドベースの稼動方式に変更可能。しかし、スレッドベースは不安定な部分もあることから、Apache2.0 系でもデフォルトではプロセスベースで動作するようになっている。

3.2 Apache のパラメータチューニング

Apache サーバのチューニングには、Apache の設定ファイル (httpd.conf ファイル) に記述されている様々なパラメータの値を変更する方法や、Apache の不要なサービスを停止する方法などがある。特に、設定ファイル内のパラメータでは Apache の細かな動作についての設定が行える。ここでは、Apache サーバのパフォーマンスへの影響が大きいとされる [2] パラメータチューニングについて述べる。以下、Apache サーバに対してどのようなチューニングが行えて、それにはどのようなパラメータを利用するのか、また、パラメータを変更することによるサーバへの影響は何かについてまとめた。

3.2.1 Keep–Alive 接続のチューニング

HTTP[3] では、サーバとクライアント間で一定の時間コネクションを保つ Keep–Alive 接続というものがある。Keep–Alive 接続とは、サーバと 1 クライアント間のコネクションを一定期間保ち続ける接続のことである。Keep–Alive 接続することによって、1 つのコネクション内でクライアントからの複数のリクエストを処理することができる。

Apache には、Keep–Alive 接続の調整を行う「KeepAliveTimeout」というディレクティブパラメータがある。次にこの KeepAliveTimeout の役割について説明する。

通常、プロセスベースで稼動している Apache では、1 つのプロセスが 1 つのリクエストを処理している。よって、1 つのリクエストを処理し終わったら、サーバとクライアントの接続は切断される。しかし、今日の Web 上のコンテンツは複数のファイル (画像ファイルなど) から成り立っている場合が多い。クライアントが、このような複数のファイルからなる Web ページにアクセスした場合、ファイルごとに TCP 接続を開始、切断していては効率も悪く、オーバヘッドも大きくなってしまう。そこで Keep–Alive 接続することにより、サーバとクライアント間のコネクションを保ったまま複数のリクエスト処理を行い、処理能力をあげることができる。

しかし、Keep–Alive 接続によってクライアントとの接続を保ち続ければ、もちろんプロセスは消費される。結果、メモリが消費されることになる。サーバ機は、搭載されているメモリ量によって使用できるメモリの限界というものがある。よって、リクエストを処理するプロセス数にも限界があり、Keep–Alive 接続で無限にクライアントとの接続を保ちつづけるわけにはいかない。そこで KeepAliveTimeout によって、Keep–Alive 接続しつづける時間を決定する。KeepAliveTimeout で指定できるのは、Keep–Alive 接続を行っている 1 つのコネクション内で、次のリクエストを待ち続けていられる最大時間である (図 3.1)。

プロセスは、KeepAliveTimeout で設定された時間内に次のリクエストがこなけれ

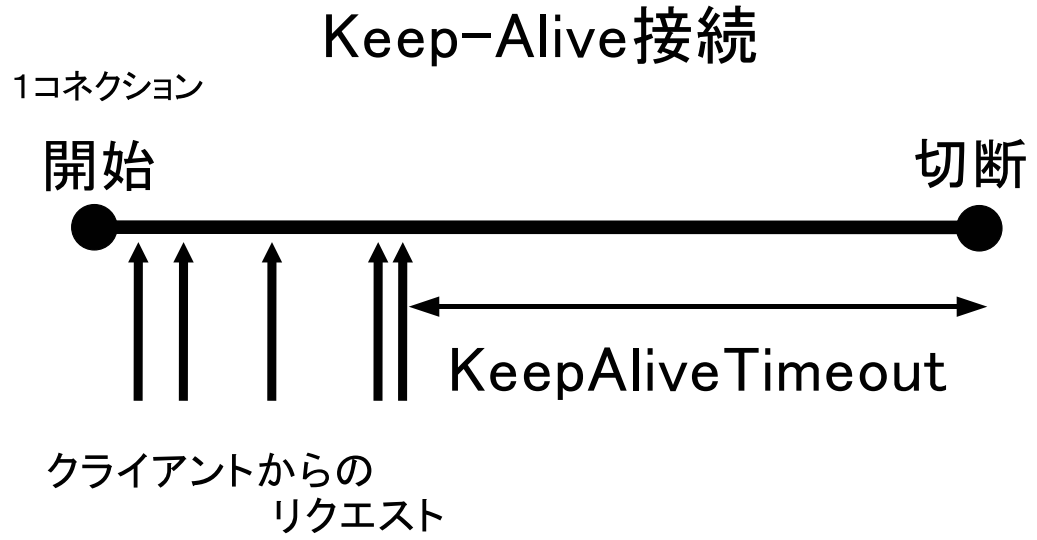


図 3.1: Keep–Alive 接続と KeepAliveTimeout の関係

ば、そのコネクションを切断し、次のリクエストを待つこととなる。この KeepAlive-Timeout の時間は長すぎるとパフォーマンスが落ちることがある。なぜなら、クライアントからリクエストがない状態で接続を保ち続けていれば、そのぶん何も処理をしていないプロセスが存在することなり、リソースは無駄となってしまう。また、リクエストがくるのを待っている待機プロセス数が減少してしまい、その結果、コネクションを確立することができずに待機するリクエストが増えてしまう。これは、サーバのパフォーマンスを下げることにつながる。つまり、サーバとクライアント間で Keep–Alive 接続することにより、効率よくリクエストを処理することができるが、KeepAliveTimeou の設定が適切でない場合、かえってサーバのパフォーマンスを悪くしてしまうと言える。KeepAliveTimeou はデフォルトで 15 秒とされている。サーバの管理者は、KeepAliveTimeou を調整することでサーバとクライアントとのコネクションの状態を変更し、サーバのパフォーマンスを変えることができる。

また、この Keep–Alive 接続を行うかどうかは、Apache の設定ファイル内の「KeepAlive」ディレクティブによって決定される。「KeepAlive」を on に設定すれば、Keep–Alive 接続が有効になり、「KeepAlive」を off とすることで Keep–Alive

接続は無効となる。サーバ側の設定では Keep–Alive 接続を有効にすることが良いとされている [4]。この Keep–Alive 接続は HTTP/1.1 から正式にサポートされている。もし、クライアントが HTTP/1.0 を利用している場合は、コンテンツの容量が先にわかる場合のみ Keep–Alive 接続有効である。つまり、動的なコンテンツを配信する場合には、Keep–Alive 接続は有効ではない。

3.2.2 サーバの同時接続数のチューニング

Apache には、「MaxClients」ディレクティブというパラメータがある。これは、サーバに同時に接続することが可能なクライアント数の上限を調整するパラメータである。

プロセスベースの Apache では Apache 本体を親プロセスとすると、クライアントからのリクエストを処理するのは、親プロセスが生成した子プロセスたちである。よって、親プロセスが生成した子プロセスの数が、同時接続可能なクライアント数を意味することになる。同時接続可能なクライアント数は、多ければ多いほどサーバのパフォーマンスは良いといえる。これは、子プロセスの数が多ければ、それだけ一度に処理できるリクエスト数が増えるからである。

しかし、MaxClients の設定値を高くし、多くの子プロセスが生成されると、サーバのメモリの不足につながってしまうことがある。その結果、スワップイン・スワップアウトが発生してしまう。スワップイン・スワップアウトが頻発すると、サーバの応答速度が遅くなったり、スワップ領域がなくなってしまい、システムがダウンしてしまうことも考えられる。つまり、この MaxClients の設定が低すぎると、生成される子プロセス数が少ないためサーバの処理能力は下がってしまう。反対に設定が高すぎると、サーバのメモリ不足につながり、結果、処理能力を下げてしまうと言える。

サーバの管理者は、サーバに搭載されたメモリ量や Apache の消費メモリ等を考慮したうえで、MaxClients の設定を行う。このように同時接続可能なクライアン

トの数を調整することによってスラッシングの発生を防ぎ、サーバのパフォーマンスを変えることができる。だが、動的なコンテンツを配信するサーバの場合は、子プロセスの使用量が動的に変化するため適切な MaxClients の値を決定するのは難しい。

3.3 その他のチューニング方法

いま述べたパラメータチューニング以外の、代表的な Apache サーバのチューニング手法は次のとおりである。

- 不必要な Apache モジュールの削除

  これよってサーバ内のメモリ等の無駄な資源の消費を無くすことができる。

- DNS の逆引きを中止

  Apache には、リクエスト処理ごとに取るログなどにアクセス元を記録する際や、クライアントの認証の際に、クライアントの IP アドレスから DNS の逆引きを行う設定がある。この DNS の逆引きを中止することにより、ネットワークの負荷を軽減させることができる。

これらの手法は、Apache の設定ファイルで有効にするか無効にするかを変更するだけですむものである。よって、本研究ではこのように 1 度の処理ですむチューニング手法については対象としない。

第 4 章

関連研究

4.1 関連研究

この節では、システムのチューニング手法や、ボトルネックの自動検出など、システムのパフォーマンスを向上させるための既存研究について述べる。

4.1.1 Web システムのパフォーマンス向上に関する研究

KeepAlive 時間の自動調整

杉本 [5] らは、Web サーバの性能パラメータを自動的に調整させることによって Web サーバのパフォーマンスの向上を図っている。杉本らの自動調整機構は、本研究と同様にサーバとクライアントとの通信の状態をモニタリングすることによって、Keep-Alive 接続時間を調整するものである。Linux 上で稼動している Apache サーバをターゲットとし、Linux カーネルと Apache との間にクライアントとの接続を制御する機構を組み込ませる手法をとっている。

杉本らの自動調整機構は Keep-Alive 接続時間の調整のみにとどまっている。また、実装方法により Unix 系の OS 上で稼動しているサーバにしか使用できないため、汎用性は低いと考えられる。

MIMO

Diao[6] らは、CPU とメモリの使用量の変化と、Keep-Alive 接続とサーバへの同時接続数には線形の関係があると主張している。その主張に基づき、CPU とメモリの使用量という 2 つの変化量から、Keep-Alive 接続とサーバへの同時接続数の調整を行うための数学的モデルを作成している。だが、サーバへの負荷状況の変化によって、CPU とメモリの使用状態は変化するために、この数学的モデルには限界があると Diao らは述べている。また、Diao らのシステムを導入するためには、既存の Apache に変更を加える必要が生じる。

パケットのモニタリングによるサーバの挙動観測

中村 [7] らは、実際のサーバとクライアントとのパケットをモニタリングすることによって、Web サーバのコネクションの状態等の挙動を詳細に観測する方法を提案している。実稼動している Web サーバの、より正確なコネクションの情報を取得することは可能であるが、取得情報の解析や得られる情報の意味等には触れられていない。

有向グラフに基づく Web システムのボトルネックの検出

清水 [8] らの研究では、過去の事例を蓄積した知識ベースの推論アルゴリズムによって Web サーバのボトルネックを求めるシステムを提案、実装している。これは、サーバ管理者にとってシステム内部のボトルネックになっている部分を特定する作業負担は削減されるが、ボトルネックを改善する作業の負担は減らない。

4.1.2 OS レベルでの動的なシステムチューニング機構

DTrace

Sun Microsystems の Solaris10 に搭載された機能の 1 つである DTrace[9] は、稼動中のシステムの状態を把握し、パフォーマンスのチューニングを行うツールである。この DTrace は、ソフトウェアを実行中に分析し、どのプログラムによってど

のプロセスをリクエストしているか見つけ出すことが可能である。DTrace には、アプリケーションやシステムの変更、再起動が不要といった特徴がある。Solaris10 上で稼動している Web サーバは、DTrace の機能を利用して間接的にパフォーマンスを向上できる可能性はある。DTrace が作動すのは Solaris10 のみなので、汎用性が低い。

4.2 システムの動作情報取得に関する既存ツール

この節では、Web サーバや OS の動作情報の取得に関連したシステムについて述べる。

4.2.1 Web サーバの動作監視ツール

mod_status

Apache には、Apache の動作情報を監視することができる mod_status というモジュールがある。mod_status を利用すると、稼動中の Apache サーバに関する動作情報を得ることができる。mod_status から得られる動作情報は以下のようなものである。

- サーバの連続稼働時間
- クライアントからの合計アクセス数
- 合計転送量
- サーバ全体の CPU 使用量
- 1 秒あたりのリクエスト数、1 秒あたりの送られたバイト数、リクエストあたりのバイト数の平均
- プロセスごとの状況

- プロセスごとの CPU 使用量
- リクエストをなげたクライアントの IP アドレス

この mod_status は、通常 Apache のインストール時に静的に組み込まれるので、httpd.conf ファイルの設定を変更するだけで利用することが可能である。mod_status を利用して Apache の動作情報を取得するためには、サーバが稼動しているローカルマシンからブラウザ等で
http://サーバのホスト名もしくは IP アドレス/server-status/
にアクセスする。すると mod_status は、Apache の動作情報の統計を HTML 形式のテキストで返してくる。

mod_status を利用して得られる動作情報は、http://サーバのホスト名/server-status/ にアクセスした瞬間的な情報でしかないため、サーバに対する負荷の詳細な分析はできない。しかし、生成されている子プロセス数や、プロセスの状況、プロセスごとの CPU 使用量などは Apache のチューニングの際に役立つと思われる。

4.2.2 Unix システム系のシステムモニタリングツール

Unix 系の OS には、システムの動作情報を把握する際に利用できる、システムパフォーマンス測定コマンドがある。それらのコマンドの簡単な説明を以下に述べる。

vmstat 各プロセスキューの状態や、メモリや CPU の使用状態の統計をかえす。

ps 実行中のプロセスごとの CPU 時間や、そのプロセスの状態をモニタする。

top システム全体のプロセスを監視する。

第 5 章

設計

5.1 設計方針

本研究で構築するシステムは、Web サーバの管理者の作業負担をできるだけ抑えたパフォーマンスチューニングを可能にするものである。以下に設計方針を示す。

- Web サーバの動作情報を自動的に収集・解析し、適切と思われるパラメータ値に動的に変更

  これにより、サーバの管理者手動によるパラメータチューニングを行わずに Web サーバのパフォーマンスを向上させる。パラメータの変更を動的に有効にできるということは、サーバを再起動、再構築しなくてすみ、管理者の作業負担は減る。

- 既存の OS や Apache には手を加えずに本システムの利用が可能

  Web サーバは 24 時間常ににサービスを提供していることが好ましいとされる中で、本システムを導入する際の手間は、できるだけおさえるようにする。

- 静的コンテンツだけではなく、動的コンテンツ配信も行う Web アプリケーションサーバに有効なシステム

  昨今の Web サーバには、HTML 文書のような静的コンテンツ配信のサーバや、クライアントから送られてくる情報によって動的にコンテンツを生成して配信する動的コンテンツ配信サーバなどが混在する。以前は静的コンテ

ンツ配信のサーバが主流であったが、今は Perl や PHP といった技術を用いた動的コンテンツ配信のサーバの需要も高くなっている。静的・動的コンテンツの 2 つのサービスを 1 つのサーバで行わなければならないという状況もある。

本システムが静的・動的コンテンツ配信の両サービスに有効なチューニングを行うにあたり、サーバ、クライアント間の Keep–Alive 接続と、サーバへの同時接続数に関するパラメータチューニングを行う。サーバの管理者がサーバのサービス形態によって、Keep–Alive 接続のチューニングを行うのか、それとも、同時接続数に関するチューニングを行うのか選択できるように、この 2 つのチューニング機構をそれぞれ独立させる。主に、動的コンテンツ配信のサーバに対しては、子プロセスの消費メモリが多いため、サーバへの同時接続数のチューニングを行う。静的コンテンツ配信のサーバに対しては、Keep–Alive 接続のチューニングを行うという方針でシステムを設計する。

本システムのチューニング機構は大きくわけて以下のような 3 つのフェーズから成る。

1. サーバの動作情報の取得部
2. 取得した動作情報の解析部
3. 解析結果をパラメータに反映させる調整部

これらのフェーズを、チューニング対象のパラメータごとに用意した。図 5.1 のように、チューニング機構はお互いに独立な立場で、チューニングを行う。

この節では、2 種類のパラメータチューニングを行うにあたり、どのようなシステムの動作情報に着目し、取得していくか述べる。

今述べた 2 つの動作情報は、稼動中のサーバから動的に取得するようにする。これは、稼動中のサーバの動作情報をリアルタイムに把握することで、その状況にあったチューニングを行えるようにするためである。

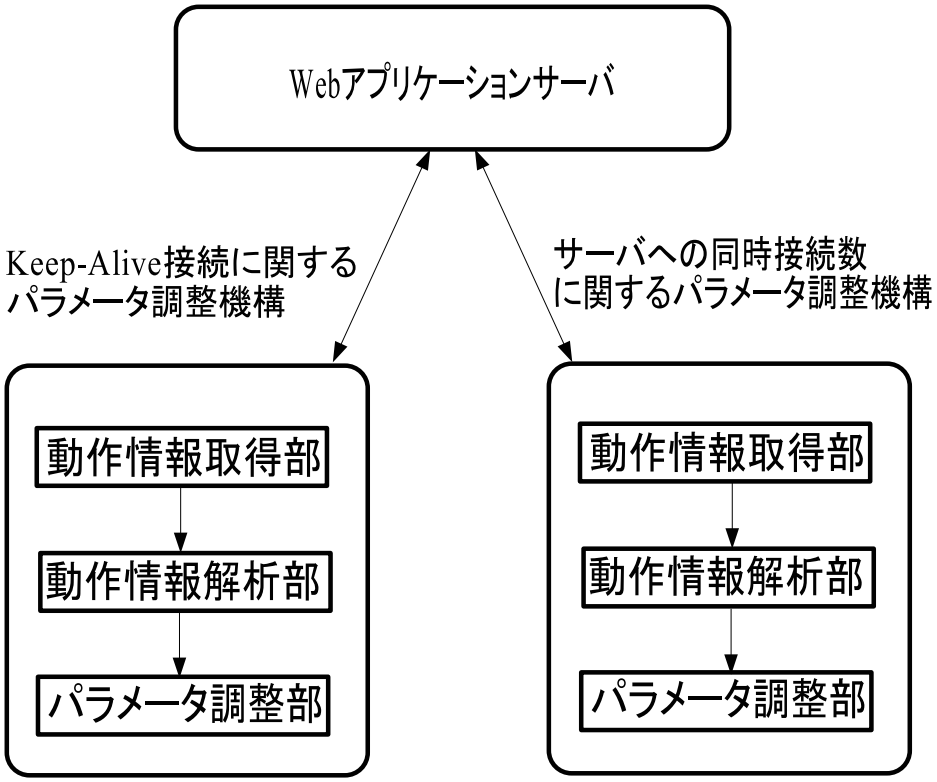


図 5.1: 本システムの構成

5.2 Keep–Alive接続に関するパラメータチューニング機構

5.2.1 動作情報の取得部

Keep–Alive 接続中、しばらくの間クライアントからのリクエストがこない時間のことを思考時間 (think time) と呼ぶ (図 5.2)。

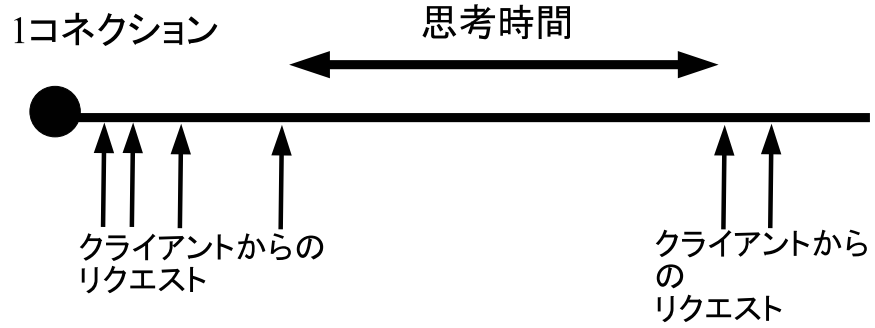


図 5.2: Keep–Alive 接続中の思考時間

たとえば、ユーザがある Web ページにアクセスし、そのページをブラウザ上で閲覧したとする。閲覧し終わった後に、そのページから他のページにジャンプしようとリンク先をクリックする。このクリックして、次のリクエストをサーバに送るまでの時間が思考時間である。1 つの Keep–Alive 接続中にこの思考時間が多くあるということは、サーバの子プロセスが何も処理を行わずに無駄にコネクションを張り続けているということである。つまり、サーバ資源の無駄な消費でしかない。そこで、本システムでは 1 つの Keep–Alive 接続中に発生するクライアントの思考時間をできるだけ短くし、サーバ資源の無駄な消費を抑えることにより、サーバ全体の処理能力をあげるためのパラメータチューニングを行う。

クライアントの思考時間をできるだけ少なくするためには、思考時間に入ったと思われるところでクライアントとの接続を切断するのが良い。そのためには、まず Keep–Alive 接続中にユーザの思考時間に入ったことを判断するための、サーバ

の動作情報を得ることが必要となる。本システムでは、思考時間突入時間の判断材料となるサーバの動作情報を、サーバとクライアントのリクエストのやりとりから得ることにする。このサーバとクライアントのリクエストのやりとりは一定ではなく、ネットワークの動作状況などにより常に変化するため、動作情報は常に取得し続けることとする。

5.2.2 動作情報の解析部

解析部では、サーバとクライアントのリクエストのやりとりから得た動作情報を基に、Keep–Alive 接続中の平均的な思考時間突入時間を算出する。平均的な思考時間突入時間を求め、その情報をもとに Keep–Alive 接続を切断することによって、Keep–Alive 接続中に生じる思考時間を減らす。また、解析部の平均的な思考時間突入時間の算出を常に行っていると、オーバーヘッドが少なからずとも生じてしまう。よって、解析は一定期間ごと行うものとし、サーバへの負荷を軽減させるようにする。

5.2.3 パラメータ調整部

1 コネクション内で、解析部が算出した思考時間突入時間を越えてもクライアントからの次のリクエストが来ない場合、その Keep–Alive 接続を切断するように Keep–Alive 接続に関するパラメータを調整する。この調整部は、一定期間ごとに作動する動作情報の解析部にあわせて作動する。つまり、動作情報の解析部が作動し、思考時間突入時間を算出するたびにパラメータの調整を行うものとする。

5.3 同時接続数に関するパラメータチューニング機構

5.3.1 動作情報の取得部

Apache はクライアントからのリクエストの状況によって、子プロセスを生成している。3.2.2 節では、生成される子プロセスが少なすぎたり、1 つのリクエスト処理で多くのメモリを消費するようなプロセスを多く作りすぎたりするとサーバの処理能力が悪くなることを述べた。

この状況を踏まえ、子プロセスのリクエストの処理等によるメモリ消費量を、サーバへの同時接続数を調整する動作情報として利用する。これは、全子プロセスのメモリ消費量を動的に得ることによって、そのときの Apache 全体のメモリ消費量を知ることができるからである。Apache 全体のメモリ消費量を動的に把握することによって、サーバがメモリ不足をおこさない範囲内で、サーバの同時接続数を調整することができる。

この動作情報の取得部は、一定期間ごとに作動し、取得部が作動時したときの全子プロセスのメモリ情報を取得する。

5.3.2 動作情報の解析部

サーバの全子プロセスのメモリ消費量をもとに、最適なサーバへの同時接続数の最大値を算出する。サーバへの同時接続数の最適値は、サーバが使用できるメモリ量と、全ての子プロセス数が消費するメモリ量とが近くなるように調整する。サーバへの同時接続数の分析は、一定期間ごとに作動する動作情報の取得部にあわせて作動するものとする。

5.3.3 パラメータ調整部

サーバへの同時接続数の調整部は、クライアントと確立するコネクション数が、解析部が算出したサーバへの同時接続数の上限値以上にならないように調整する。クライアントからリクエストがくるのを待っている状態のプロセスのことを、ここでは待機プロセスと呼ぶ。この待機プロセスは、リクエストの処理が終わったプロセス、もしくは新しく生成されたプロセスである。調整部は、クライアントと確立するコネクション数を変更させるために、この待機プロセスの数を調整する。

第 6 章

実装

6.1 実装概要

本研究のチューニングの対象 Web サーバは、Apache2.0 である。Apache2.0 にはプロセスベースとスレッドベースの 2 通りの稼動パターンがある。プロセスベースとスレッドベースの違いによるパフォーマンスの違いはない [10]。よって、本研究ではデフォルトのプロセスベースで稼動する Apache サーバを対象とする。本システムは、Apache2.0 のモジュールとして実装し、約 700 行の C のプログラムで構成されている。モジュールとして実装することにより、既存システムに変更を加えずに本システムを導入することが可能である。このモジュールは、静的・動的コンテンツ両方に有効なチューニングを行うために、5 章で述べた 2 つのパラメータ調整を行う。Keep-Alive 接続に関するパラメータは KeepAliveTimeout、サーバへの同時接続数に関するパラメータは MaxClients とする。この 2 つのチューニングを行うモジュールは設計方針に従い、完全に分離している。Apache サーバの動作情報を利用し、2 つのパラメータを動的にチューニングするシステムを構築した。

6.2 Apache2.0 モジュール

Apache は機能ごとに分割されたモジュールで構成される。Web サーバとして必要最低限な基本的な機能を http_core モジュールとしてまとめ、その他の機能は取

り外し可能なモジュールとして実装している。よって、独自の機能をモジュールとして組み込むことが可能である。そこで、本システムは図 6.1 のように、2 つの Apache モジュールとして実装する。

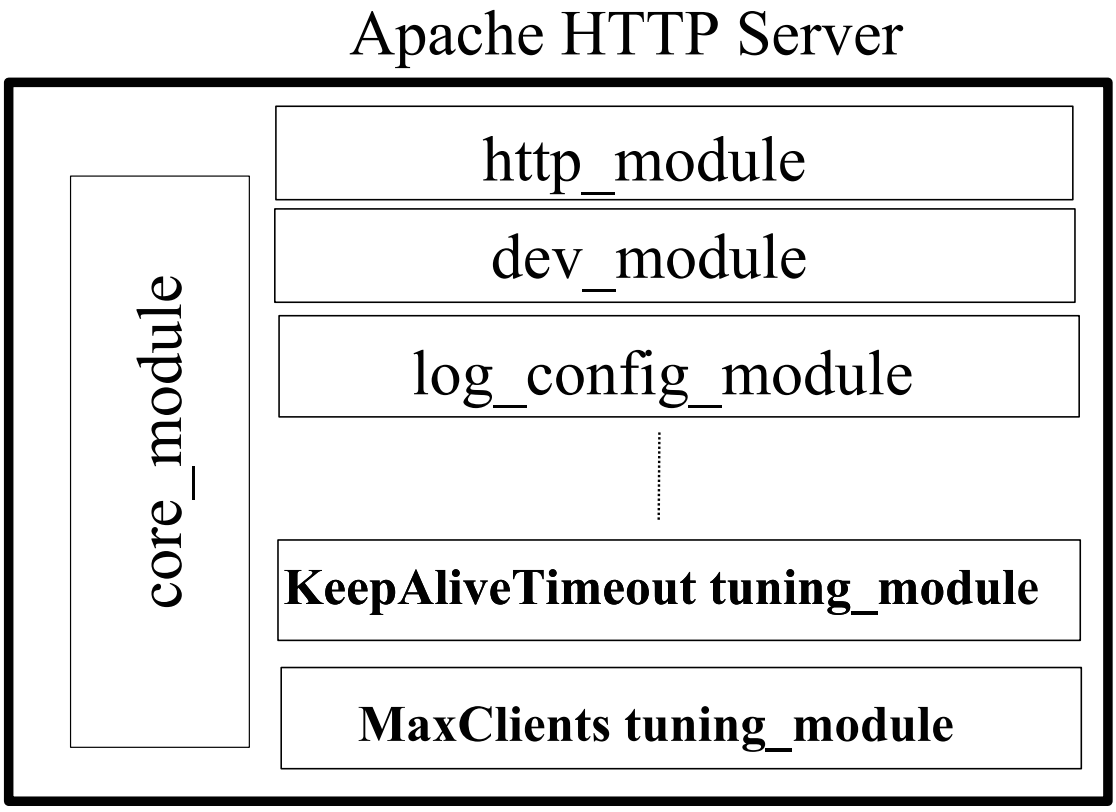


図 6.1: モジュールのアーキテクチャ

モジュールとして実装することの利点を以下に述べる。

利点

- モジュールは Apache 内部で動作するため、Apache の内部情報を比較的容易に参照・変更することが可能である。
- Apache サーバの動作をカスタマイズする際に用いられる他の手法に比べ、Apache モジュールの実行速度は高速である。小山 [11] によって報告されている Apache モジュールとその他の既存のカスタマイズ手法との実行速度の差を表 6.1 に記す[1] 。この実行速度の差は、Apache モジュー

[1] 実験は、現在の時刻を決まった形式に整形する処理を 50 回繰り返して HTML を送信するアプ

ルが他の手法と違い、プロセスの生成やプロセス間通信を行う必要がないために生じるものである。

- API が整っているので、プログラマの負荷は軽減される。また、API は Apache が稼動している OS の差異も吸収できるようになっている。そのため、OS に依存しないモジュールの作成が可能である。
- Apache モジュールのリソース管理は Apache 本体が行うため、モジュールによるメモリリーク等の危険性は減る。

表 6.1: モジュールとその他の手法における実行速度の比較

対象	処理件数/sec	転送レート
CGI(Perl)	11.74	24.50
CGI(C)	48.90	102.47
Java Servlet(Tomcat+mod_jk+Apache)	84.49	178.38
Java Servlet(Tomcat)	104.90	219.14
PHP	102.93	219.28
mod_perl(Apache::Registry)	108.16	229.11
Apache module	311.73	652.85

リケーションを実行したもの。それぞれ 10 クライアントから同時に 1000 回リクエストを発行するテストを行い apache_bench で計測。それぞれの環境で 10 回行った結果の平均値を記載。

6.2.1 モジュールの動的な組み込み

前節で、Apache 本体である http_core モジュール以外のモジュールは取り外しが可能であることを述べた。モジュールを Apache に組み込む方法には、静的な組み込み手法と動的な組み込みの 2 通り手法がある。静的な組み込みも、動的な組み込みもモジュール開発上の差異は生じない。ただし、静的な組み込みは、モジュールの変更時やアンロードのたびに Apache をビルドする必要があり、管理者の運用時の負担が生じてしまう。よって、本研究のモジュール組み込みには、動的な組み込み手法を用いる。

以下にに動的な組み込み手法について簡単な説明を述べる。

動的な組み込み手法とは、httpd バイナリとは別に動的共有オブジェクト (DSO: Dynamic Shared Object) としてコンパイルし、ダイナミックリンクさせて組み込む方法である。DSO をサポートしているのは、mod_so.c というモジュールである。ただし、この mod_so.c モジュールは、httpd バイナリに静的に組み込まれている必要がある。mod_so.c モジュールが組み込まれていれば、管理者が Apache の設定ファイルを書き換えるだけで、Apache の起動や再起動時に任意のモジュールを動的にロード・アンロードすることができる。

本システムではモジュールの組み込みには、DSO による動的な組み込み手法を用いるため、本研究ではチューニングの対象となる Apache サーバには、すでに mod_so.c モジュールが httpd バイナリに静的に組み込まれているという前提にもとづくものとする。

6.2.2 内部フェーズへのフック

本システムの Apache モジュールは、稼動中のサーバの動作情報の動的に取得・解析を行う必要がある。Apache モジュールは、このような要求に対応するため、Apache 内部の処理フェーズにフックすることができる。本システムでは、この処

理フェーズへのフックを利用する。稼動中のサーバの処理にフックすることで、動的にサーバの動作情報を取得したり、サーバのパラメータの変更を行ったりすることが可能となる。

次に、Apache の処理フェーズのフックについて述べる。親プロセスから生成された子プロセスは、クライアントからのリクエスがくるのを特定の port(通常は 80 番 port) を listen する形で待機している。リクエストがきたら、リクエスト処理のループが開始される。リクエストの処理は、

- リクエスト解析
- URI 変換
- ヘッダ解析
- アクセス制御/認証
- MIME チェック
- コンテンツ出力
- ログ出力
- 終了処理

といった一連の流れになっている。Apache モジュールは、これらの処理ごとに介入できる (図 6.2)。

6.3 Keep-Alive 接続のチューニング

6.3.1 KeepAliveTimeout

5.2.1 節で述べたように、Keep-Alive 接続に関するパラメータのチューニングでは、Keep-Alive 接続中に生じる思考時間を少なくなるように調節し、サーバの処理

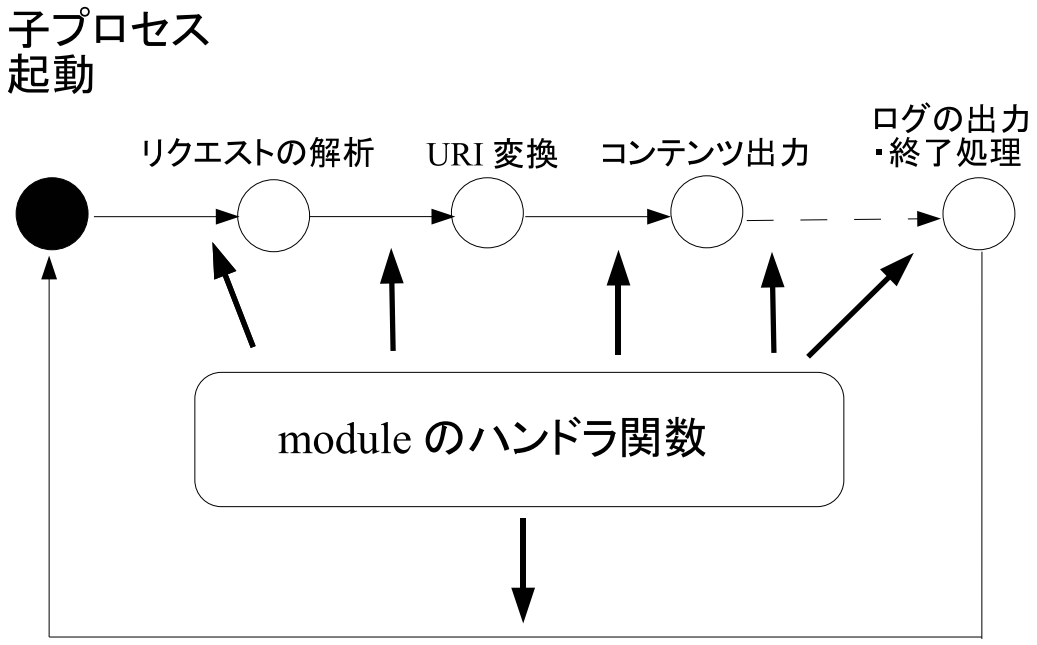


図 6.2: 内部フェーズへのフック

能力をあげることが目的である。これを実現するために、Keep-Alive 接続を切断する時間を決定している KeepAliveTimeout を調整する。この KeepAliveTimeout を調整し思考時間を少なくする方法としては、以下の 2 つの手法が考えられる。

1. 1 コンテンツが複数のファイルからなる場合、すべてのファイルの転送が終了したところで Keep-Alive 接続を切断する。

2. クライアントからのリクエストが到着する間隔を計測し、リクエストが到着する間隔が長くなったところをクライアントが思考時間にはいったところと判断し、Keep-Alive 接続を切断する。

1 の手法では、1 コンテンツを転送し終わるまでの平均ファイル転送速度を求め、それを KeepAliveTimeout に適応するという方法が考えられる。これは、1 コンテンツをなす全てのファイルをクライアントに転送し終わった時間で接続を切るということである。

しかし、この手法は適切でない。なぜなら、クライアントがサーバにリクエストをなげる環境 (ブラウザや HTML パーサ) によって、1 コンテンツ内のファイルすべてにリクエストを発行する方法が異なるからである。1 コンテンツ内のすべてのファイルを取得するのに、サーバとのコネクションを 1 つしかはろうとしない場合もあれば、コネクションを複数はって処理を行おうとする場合もある。例えば、1 コンテンツが 5 つのファイルからなる場合、コネクションを一度に 5 つはって処理するのと、コネクションを 1 つしかはらずに処理をするのでは、クライアントの思考時間に突入する時間が異なってきてしまう。よって、この手法は適切ではないと考える。

そこで、本システムでは 2 の手法を用いる。KeepAliveTimeout のチューニングに利用するサーバの負荷情報には、クライアントからのリクエスト間隔を使用する。リクエスト間隔とは、1 コネクション内でのクライアントからのリクエストがきてから、次のリクエストがくるまでの時間のことを言う。この 1 コネクション内でのリクエスト間隔を計測し、リクエスト間隔がある一定時間以上長くなったところを、クライアントが思考時間にはいったと判断する。この手法では、1 つ 1 つのクライアントの動作情報にもとづく情報を使用するため、1 の手法の問題点で挙げた「クライアントの環境による、リクエストを発行する動作の差異」を考慮する必要はない。

6.3.2 クライアントからのリクエスト間隔の取得

1 コネクション内での、クライアントからのリクエストと次にくるリクエストまでの間隔を計測すると述べたが、1 つ注意しなければならない点がある。Keep-Alive 接続中では、次のリクエストがくるまでクライアントとの接続を維持するパラメータが KeepAliveTimeout であることはすでに説明した。しかし Apache では、リクエストが到着した後は、クライアントとの接続を維持するパラメータは、KeepAliveTimeout ではなく TimeOut が用いられる仕組みになっている。TimeOut

とは、各イベントについて、リクエストを失敗させるまでにサーバが待つ時間を設定しているパラメータである。つまりサーバは、リクエストの処理にどんなに時間がかかろうとも、TimeOut の時間内はクライアントとコネクションをはり続けるということである。Apache では TimeOut 適用後、TimeOut の時間内にリクエストの処理が終了する、もしくは、TimeOut の時間内にリクエストの処理が終了せず、に失敗したら、クライアントとの接続を維持するパラメータには KeepAliveTimeout が再度適用される。よって、Keep–Alive 接続中では、KeepAliveTimeout と、TimeOut が交互に適用されることとなる。

このことから、KeepAliveTimeout のチューニングに利用するリクエスト間隔を、1 つのリクエストと次にくるリクエストまでの時間としてしまうのには問題がある。なぜなら、リクエストとリクエストの間隔の中に、今述べた TimeOut 時間内のリクエストの処理時間が含まれてしまうからである。TimeOut の設定はデフォルトで 30 秒とされているため、最大で 30 秒のリクエストの処理時間がリクエスト間隔の中にふくまれてしまう。予備実験の計測では、大きな画像ファイル (数 Kbyte〜数 Mbyte) の転送 (＝リクエストの処理) に数秒から数 10 秒かることもあった。この状況では、リクエスト間隔は数秒以上の値で記録されてしまう。実際は、サーバとクライアントのファイルの転送中であるにもかかわらず、本システム側では思考時間にはいったと判断されかねない。このリクエストの処理時間を含むことによる、リクエスト間隔の誤差をなくす必要がある。よって本システムでは、図 6.3 のように、1 つのリクエストの処理終了時間と、次にくるリクエストまでの時間をリクエスト間隔として計測する。これにより、計測するリクエスト間隔には、TimeOut 時間内で接続されている時間が含まれずにすむ。

次に、リクエスト間隔の計測ハンドラについて述べる。リクエスト間隔の計測ハンドラは、Keep-Alive 接続中のすべてのコネクションを監視する。リクエスト間隔を計測するのに必要な情報は、現リクエストを取得した時刻と、同じコネクション内での 1 つ前のリクエストの処理が終了した時刻である。

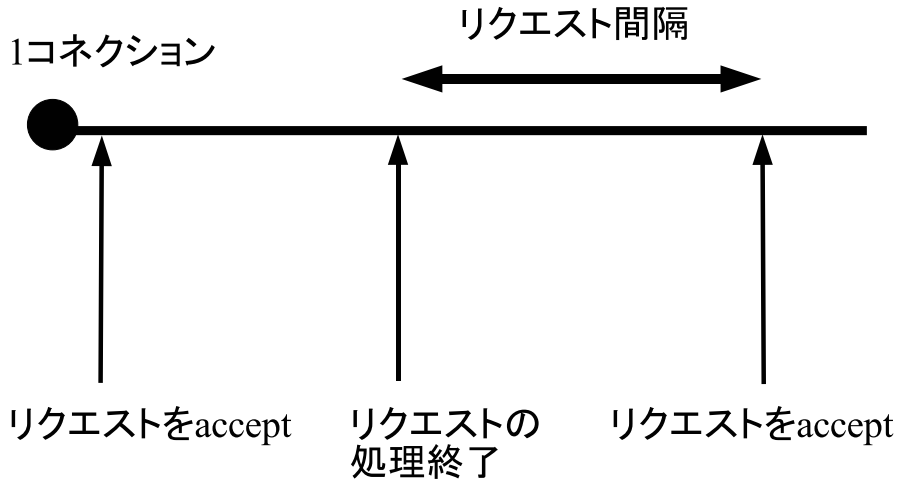


図 6.3: 取得するリクエスト間隔

現リクエストを取得した時刻の取得

まず、現リクエストを取得した時刻を得る方法であるが、これには Apache 内部の request_rec 構造体に格納されている request_time フィールドの値を利用する。この request_rec 構造体とは、リクエストごとに保持している構造体であり、request_time フィールドには、そのリクエストが取得された時刻がμ秒単位で格納されている。request_time フィールドを用いる方法以外にも、より正確なリクエスト取得時刻を求める方法としてリクエストを取得して accept(取得に成功した状態) し終わった直後のフェーズにフックをかけて、その時刻を用いる方法がある。実は、request_time フィールドに時刻がセットされるのは、リクエストを accept し終わった後のフェーズよりも 1 つあとのフェーズのため、後者の手法の方がより正確なリクエスト取得時刻を得ることができる。しかし、両者から得られる時刻の差は数μ秒とわずかなため、本システムでは request_time フィールドの値を利用することとした。

1 つ前のリクエストの処理の終了時刻の取得

1 つ前のリクエストの処理が終了した時刻を得る手法について述べる。子プロセスは、リクエストの処理を終了した次に、処理に関するログに書き込むフェーズに

移る。計測ハンドラは、このログ書き込みのフェーズにフックし、ログ書き込みのフェーズに移行した時刻を timeofday() で取得する。この時刻を、リクエストを処理し終わった時刻として記録する。このように、リクエスト間隔を求める際には記録された 1 つ前のリクエストの処理終了時刻を用いる。

リクエスト間隔を記録するタイミング

リクエスト間隔を記録するタイミングは、リクエストの処理終了時刻を求めるのと同じく、ログ書き込みフェーズにフックをかけた時とした。リクエスト間隔取得ハンドラとフックの様子を図 6.4 に示す。このリクエスト間隔の計測ハンドラは、全てのコネクション内で作動し記録を行う。

以上のようにして求められた、現リクエストが取得された時刻と 1 つ前に取得されたリクエストの処理終了時刻との差からリクエスト間隔を記録する。今回、本システムでは 100m 秒単位で記録することとする。

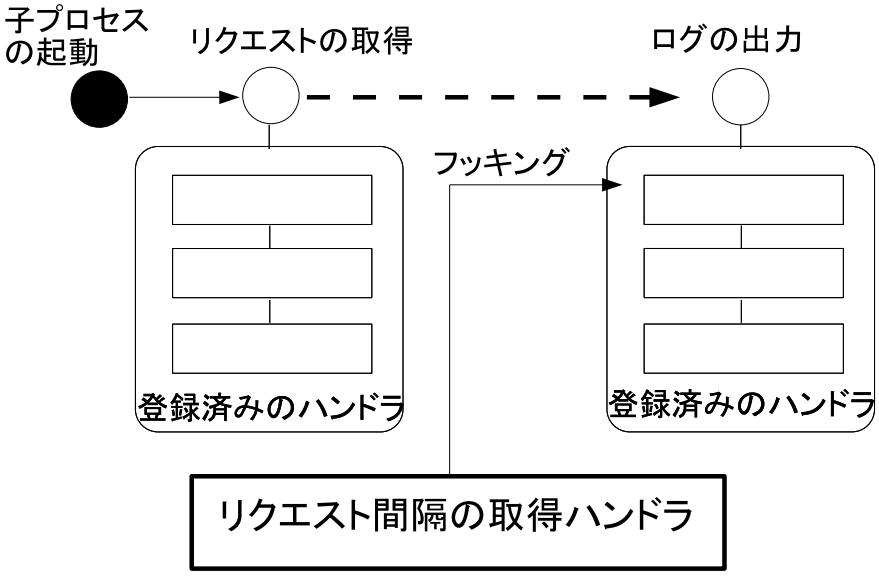


図 6.4: リクエスト間隔計測ハンドラ

6.3.3 KeepAliveTimeout の最適値の算出

解析ハンドラでは、取得したリクエスト間隔の分布を用いて KeepAliveTimeout の最適値を算出する。KeepAliveTimeout の最適値の算出方法に以下の方針をとる。

- 取得したリクエスト間隔の分布から、リクエスト間隔の平均 μ と標準偏差 σ をもとめる。
- σ に定数 α 分だけ重み付けをし、

$$\mu + \alpha\ \sigma = KeepAliveTimeout \tag{6.1}$$

を最適値として算出する。

このリクエスト間隔の平均 μ、標準偏差 σ と重み付け量定数 α の関係は図 6.5 のようになる。

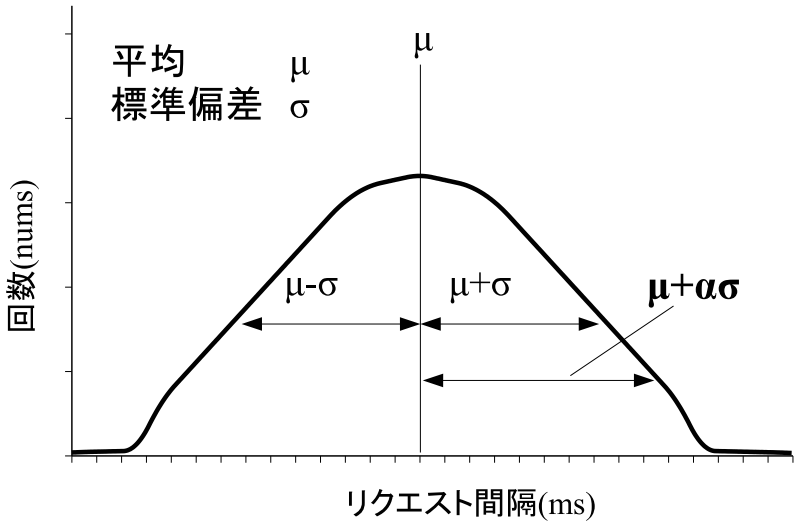


図 6.5: リクエスト間隔の分布と重み付け α の関係

表 6.2: 実験用サーバマシンのスペック

[htbp]

OS	Vine3.2(Linux2.4)
CPU	Intel Pentium 4 1700MHz
メモリ	256MB
NIC	Intel 82558/8/9 Pro/100 Ethernet

6.3.4 KeepAliveTimeout の値と、サーバのスループットとの関係調査

重み付け定数 α と、リクエスト間隔の平均 μ と標準偏差 σ の求め方を決定するにあたって、KeepAliveTimeout とスループットの関係を、実験により調査した。KeepAliveTimeout の違いによるスループットの変化を実際に測定、解析することにより、KeepAliveTimeout の調整に用いる重み付け定数 α、リクエスト間隔の平均 μ と標準偏差 σ の求め方を定義する。

実験環境

実験に用いる観測対象のサーバマシンのスペックは表 6.2 に、観測側のクライアントマシンのスペックは表 6.3 に示す。図 6.6 のようにサーバマシンとクライアントマシンは、閉じたネットワーク内で 1 台の 100Base-T のスイッチングハブによって接続している。観測対象の Web サーバには Apache2.0.55 を使用した。

実験に用いる負荷

サーバのスループットを計測するにあたって、ベンチマークソフトを使用した。今回、ベンチマークソフトには Microsoft Web Application Stress Tool[12] を用いた。Microsoft Web Application Stress Tool を使用した理由は以下のとおりである。

表 6.3: 実験用クライアントマシンのスペック

OS	Windows XP Professional Version 2002
CPU	Intel Pentium M 1100MHz
メモリ	256MB
NIC	Realtek RTL8139/810X Family PCI Fast Ethernet

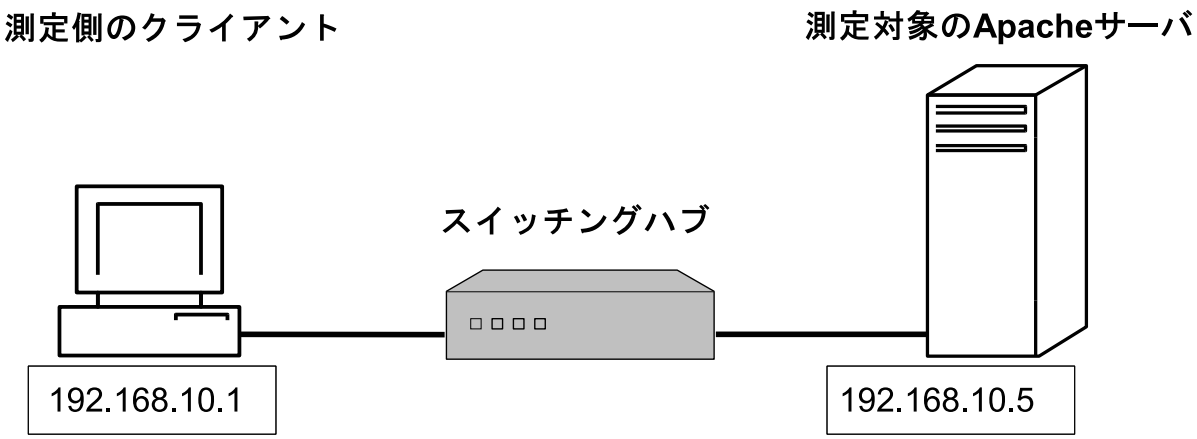


図 6.6: 実験環境

- Keep-Alive 接続を有効にした状態で負荷テストが行える。
- 1 クライアントのリクエストのシナリオで、複数の URL に対してリクエストを投げることができる。
- 1 クライアントのリクエストのシナリオを、1 コネクションで処理するため、Keep-Alive 接続中のリクエスト間隔が採取しやすい。

次に、負荷に用いたリクエストのシナリオの詳細を、以下に述べる。

- 1 つの HTML ファイルと、その HTML ファイルにリンクがはられている 6 つのファイル、計 7 つのファイルをクライアントがサーバにリクエストする。
- リクエストされるファイルの内訳
  - 1 つの HTML ファイル
  - 1 つの JPG ファイル
  - ファイル名の異なる 2 つの BMP 画像ファイル (容量はすべて同じ)
  - ファイル名の異なる 3 つの GIF 画像ファイル (容量はすべて同じ)
- ファイルごとの容量は表 6.4 に示す。

今回の実験では、サーバのスループットをリクエストレート (Requests per Second) として計測する。サーバの KeepAliveTimeout 以外の各種設定は、何も手を加えず、デフォルトのままのものを使用した。

以上のリクエストのシナリオで、クライアントの同時接続数を 200 としたテストを 3 分間行う。KeepAliveTimeout の設定値を 100m 秒ごと変更し、各設定で 3 回づつテストを行い、その平均値で評価する。KeepAliveTimeout の設定値を 100m 秒ごと変更するにあたって、サーバのパラメータ設定を行う現状の httpd.conf ファイルからは、KeepAliveTimeout の値は秒単位でしか設定できない。そこで今回は、

表 6.4: リクエストするファイルの種類と個数

ファイルの種類	ファイルの容量 (Byte)	個数
HTML 文書	1282	1
GIF	2326	3
BMP	31498	2
JPG	674	1

100m 秒単位で KeepAliveTimeout の値が設定できる実験用プログラムを Apache モジュールで作成し、本実験に用いた。

ネットワークの帯域制御

本実験では、サーバとクライアント間のネットワークトラフィックの状況が異なる場合における KeepAliveTimeout とスループットの関係も調査した。実世界のネットワーク上では、一時的にバーストが生じたり、サーバへのアクセス数が増減したりすることによって、サーバとクライアント間のネットワークトラフィックが変化することが多い。このようにネットワークトラフィックが変化することによって、サーバとクライアントとの通信の挙動が異なってくることが予想できる。通信状態、つまりリクエストのやりとりが異なれば、Keep-Alive 接続中のリクエスト間隔も異なってくるであろう。そこで本実験では、サーバとクライアント間のネットワークの帯域を制御し、異なるネットワークトラフィック状況下での KeepAliveTimeout とスループットの関係を調査する。

本実験のネットワークの帯域制限には、iproute2+tc の手法を用いた。iproute2 は、Linux カーネル内から IP 接続の制御を行うフレームワークである。この iproute2 で、帯域制御関連の設定を行うコマンド tc を用いて帯域を制御する。制御の機構

にはHTB(Hierarchical Token Buket) の機構を使用した。以上の手法をサーバ機のカーネルで用いることにより、サーバの任意のポートに対する帯域を制御することが可能となる。実験では、http の通信で使用される 80 番ポートの帯域に制限をかける。

実験にあたり、このiproute2+tc を使用して帯域制限をかけた状況での、実際の帯域を調べる。サーバ側でiproute2+tc の手法を使用し、帯域を 100Mbps、75Mbps、50Mbps、10Mbps に制御した条件での、実際のネットワークの転送速度を調べた。その結果を表 6.5 に示す。

この調査は、サーバ側がiproute2+tc で帯域制限している環境で、ftp コマンドを使用してネットワークの転送速度を計測したものである。測定の概要は以下のとおりである。

- ftp でファイル等のデータのやりとりに使用される 20 番ポートの帯域を制限。
- サーバからクライアントへ 150Mbytes のファイルを転送させる。
- 帯域制限なし、100Mbps、75Mbps、50Mbps、10Mbps の帯域制限をかけた状態で各 3 回づつ計測。その平均値で評価する。

表 6.5 の結果より、iproute2+tc によるネットワークの帯域制限では、実際のネットワークの状況がおおむねiproute2+tc で設定した値に近い帯域に制限されていることがわかる。

表 6.5: 実験に用いた帯域制限と、その転送速度

帯域制限 (Mbps)	帯域制限なし	100	75	50	10
測定された転送速度 (Mbps)	91.83	90.75	73.14	48.83	9.72

本実験では、この 100Mbps、75Mbps、50Mbps、10Mbps と制限をかけた状態でそれぞれに上で述べた負荷テストを行った。

実験結果

実験より求められた KeepAliveTimeout とスループットの関係を、帯域制限ごとに示したものが図 6.7〜図 6.10 である。図 6.7 の 100Mbps の帯域結果では 4 秒から 8 秒の間、図 6.8 の 75Mbps の帯域結果では 200m 秒から 500m 秒の間、図 6.9 の 50Mbps の帯域結果では 100m 秒から 300m 秒の間、図 6.10 の 10Mbps の帯域結果では 100m 秒から 1 秒の間がおおむねスループットが良いということが見てわかる。

全ての帯域において、デフォルトの 15 秒より短い KeepAliveTimeout 値にすることで、スループットは上がっていく傾向がある。これは、1 つの Keep–Alive 接続中のクライアントの思考時間が減ることによって、子プロセスの処理する割合が増えていっているからであると考えられる。しかし、図 6.10 の 10Mbps の帯域をのぞいて、ある一定値までスループットが上がった後は、それ以上 KeepAliveTimeout の値を短くしてもスループットが下がってきている。これは、一定の時間よりも KeepAliveTimeout の時間を短くしてしまうと、3.2.1 節で述べたように、1 クライアントから連続してくるリクエストに対して毎回コネクションの確立・切断を繰り返してしまうこととなり、サーバへの負荷がかかってしまっているからである。その結果、サーバのスループットがさがってしまっている。

6.3.5 KeepAliveTimeout の値とリクエスト間隔との関係調査

次に、任意の KeepAliveTimeout 値に設定された状態で負荷テストを行い、そのときのリクエスト間隔の分布を求めた。これにより、6.3.4 節の実験によって得られたスループットが良いときのリクエスト間隔と、スループットが良くないときのリクエスト間隔とに差があるのかどうかを調査する。

実験環境

6.3.4 節の調査実験と同様の環境で実験を行った。

実験に用いる負荷

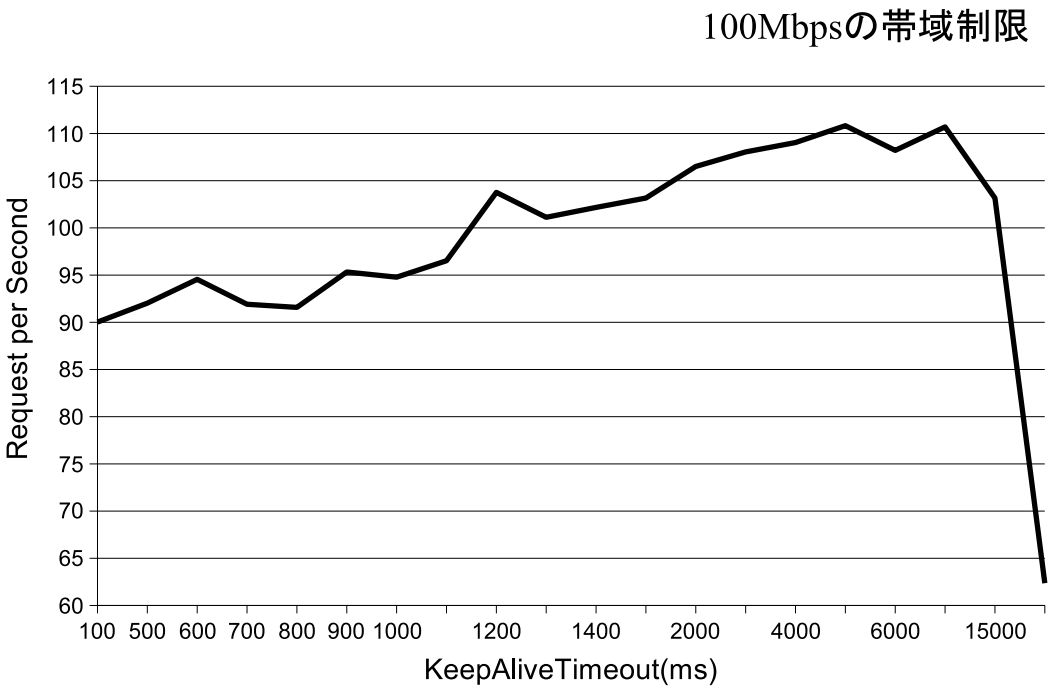


図 6.7: 100Mbps 帯域制限での KeepAliveTimeout とスループットの関係

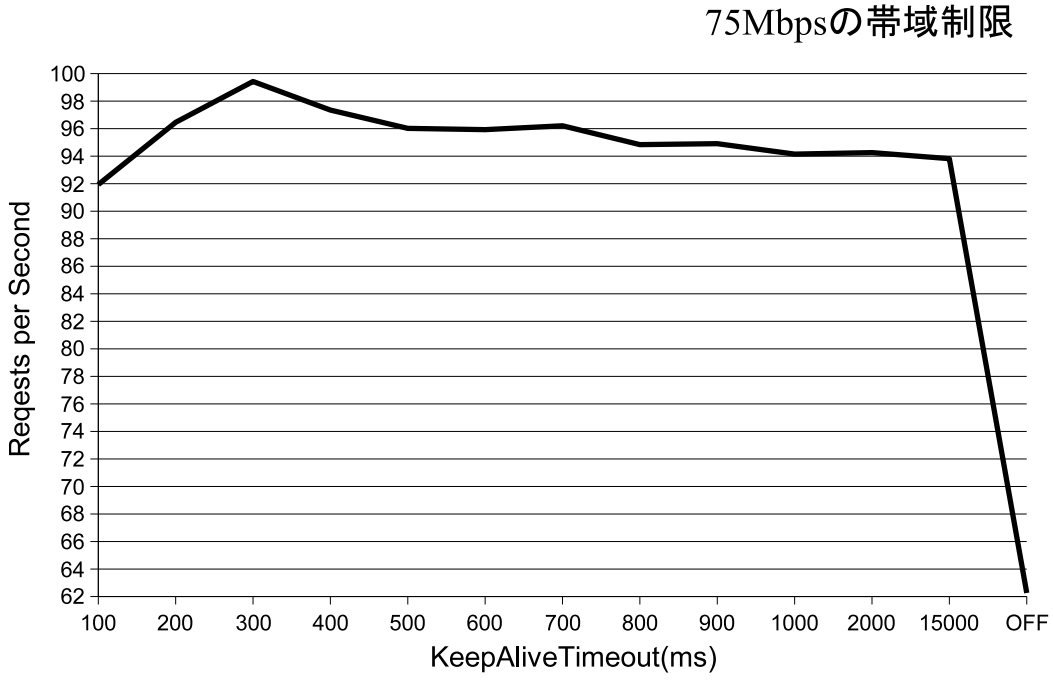


図 6.8: 75Mbps 帯域制限での KeepAliveTimeout とスループットの関係

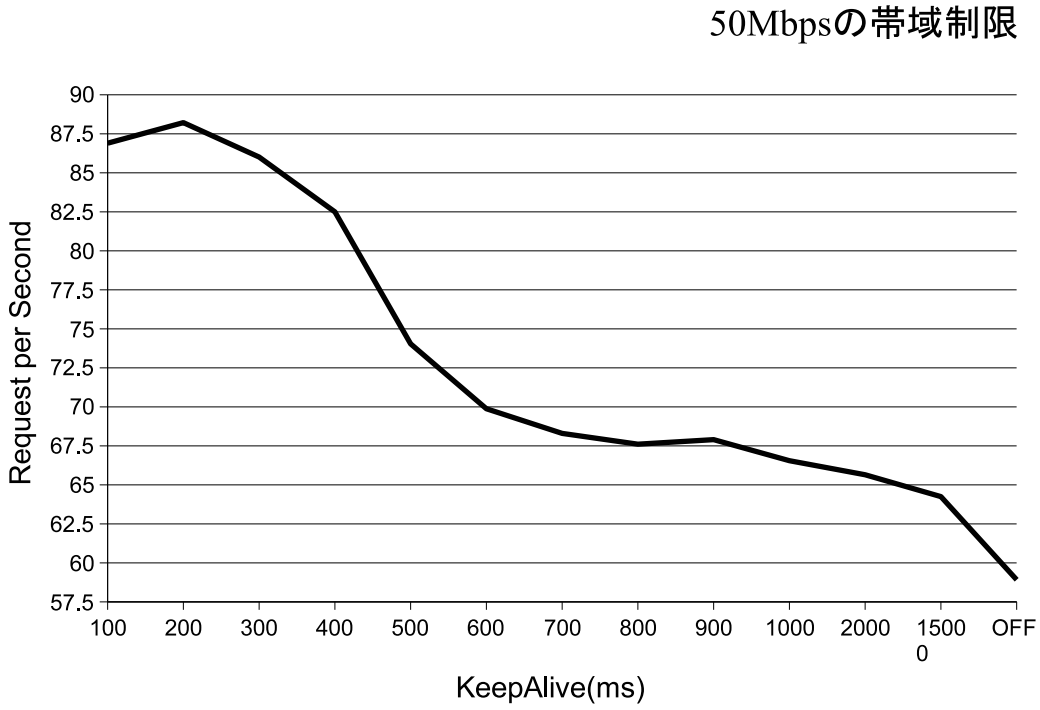


図 6.9: 50Mbps 帯域制限での KeepAliveTimeout とスループットの関係

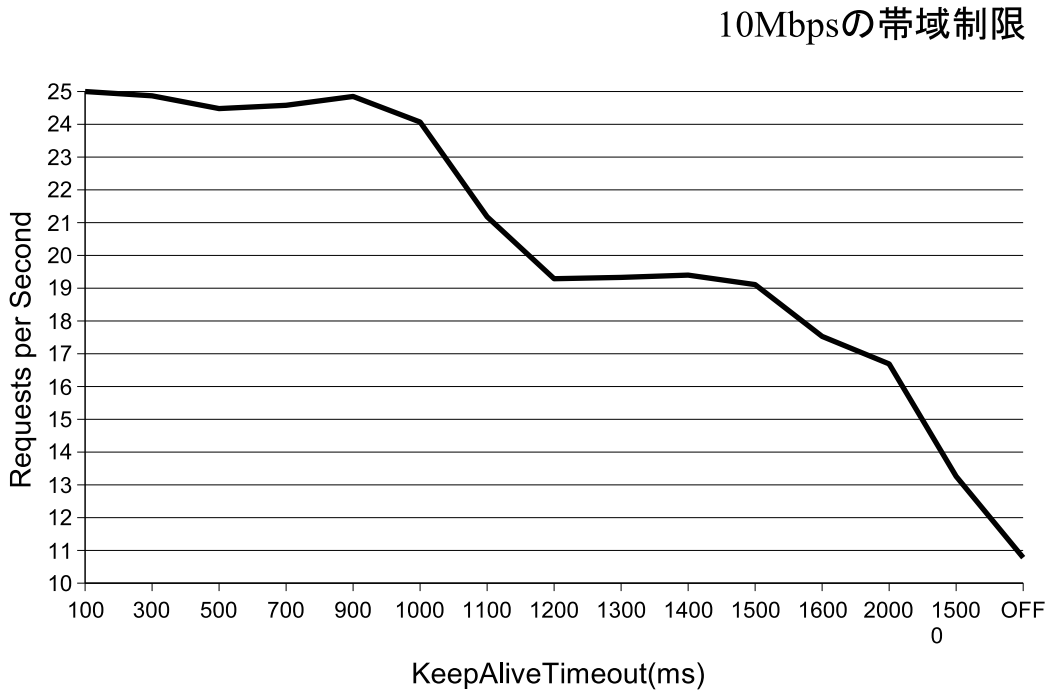


図 6.10: 10Mbps 帯域制限での KeepAliveTimeout とスループットの関係

KeepAliveTimeout の値とサーバのスループットの関係の調査実験と同様のリクエストシナリオでベンチマークテストを行った。ただし、今回は標本数であるリクエスト間隔の数が重要となる。6.3.4 節の実験では、ベンチマークを走らせる時間を 3 分と一律にしたが、今回はベンチマークの実験中に得られたリクエスト間隔の標本数が十分と思われる値になるまでベンチマークテストを走らせる。本実験では、十分な標本数を 20000 とし、Keep-Alive 接続で得られたリクエスト間隔の標本数が 20000 になるまでテストを行った。実験には、6.3.4 節の実験と同様にサーバとクライアント間のネットワークに帯域制限をかけ、100Mbps、75Mbps、50Mbps、10Mbps の帯域制限ごとに実験を行う。実験は、KeepAliveTimeout 値の変更単位と同様に 100m 秒単位で行う。

設定する KeepAliveTimeout 値は、

- KeepAliveTimeout のデフォルト値の 15 秒
- 6.3.4 の実験でスループットが一番高く、最適値と思われる KeepAliveTimeout の値
- 最適値よりも低い KeepAliveTimeout の値

の 3 種類とする。

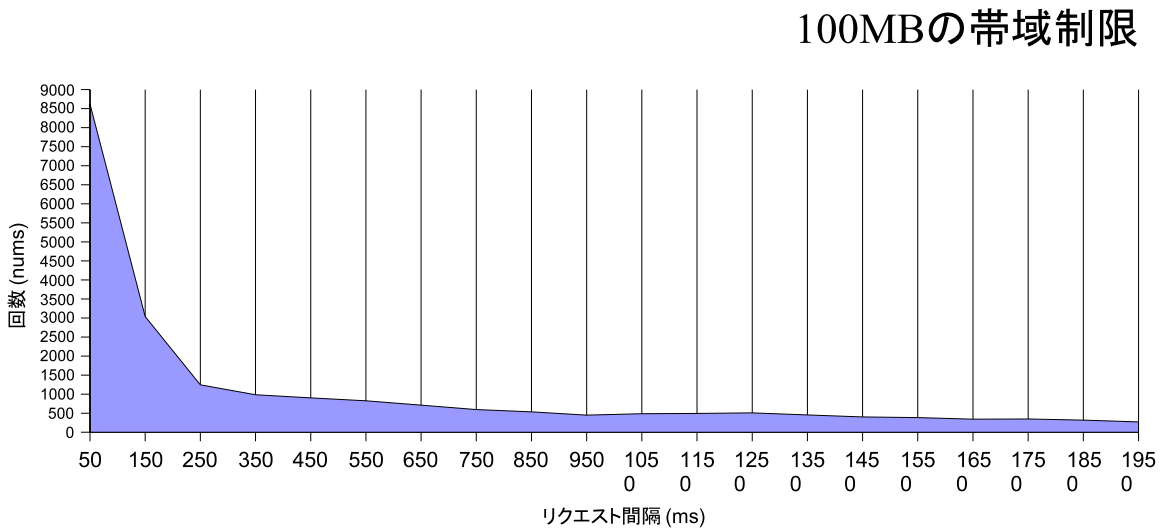
これにより、なにもチューニング等を行っていないデフォルト値でのリクエスト間隔、スループットが一番良いときのリクエスト間隔、KeepAliveTimeout の値が最適値よりも低く設定されている状況のリクエスト間隔を調査する。

表 6.6 で、実験で用いる 3 種類の KeepAliveTimeout 値を帯域制限ごとに示す。

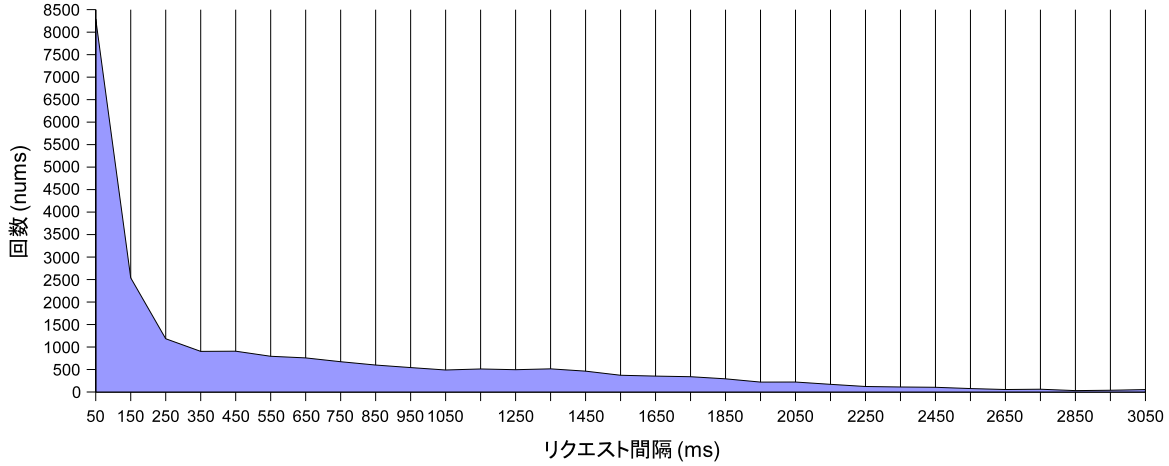
実験結果

実験より求められた KeepAliveTimeout とリクエスト間隔の関係を、帯域制限ごとに示したものが図 6.11〜図 6.14 である。

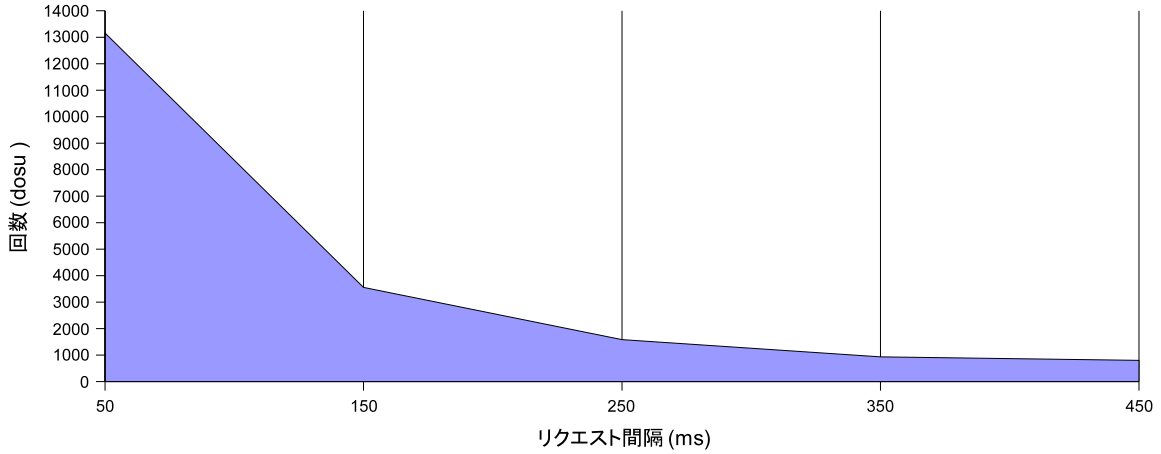
この実験より、以下のような特徴がわかる。



KeepAliveTimeout=15s(デフォルト)設定でのリクエスト間隔

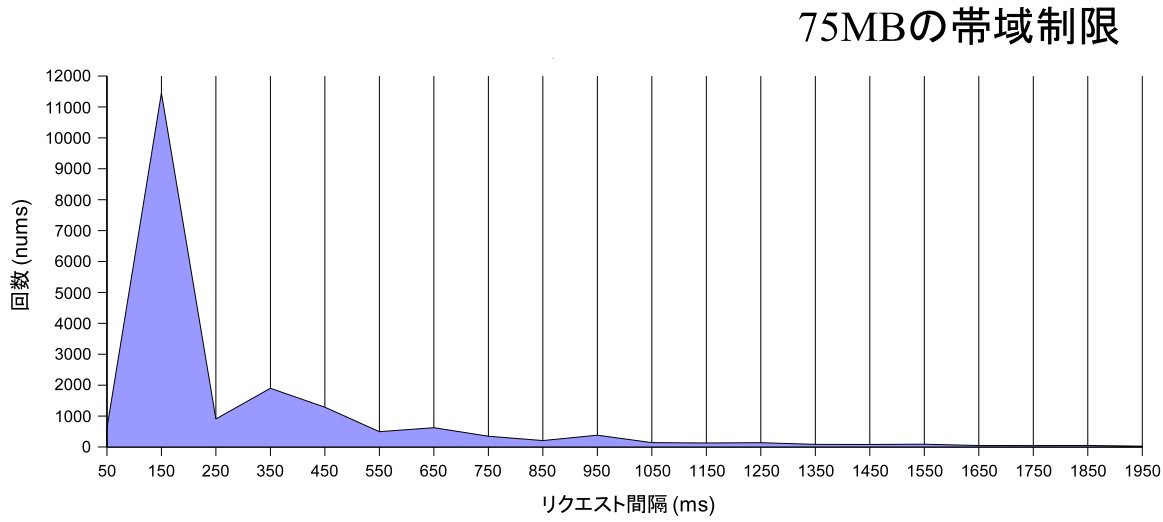


KeepAliveTimeout=5s(最適値)設定でのリクエスト間隔

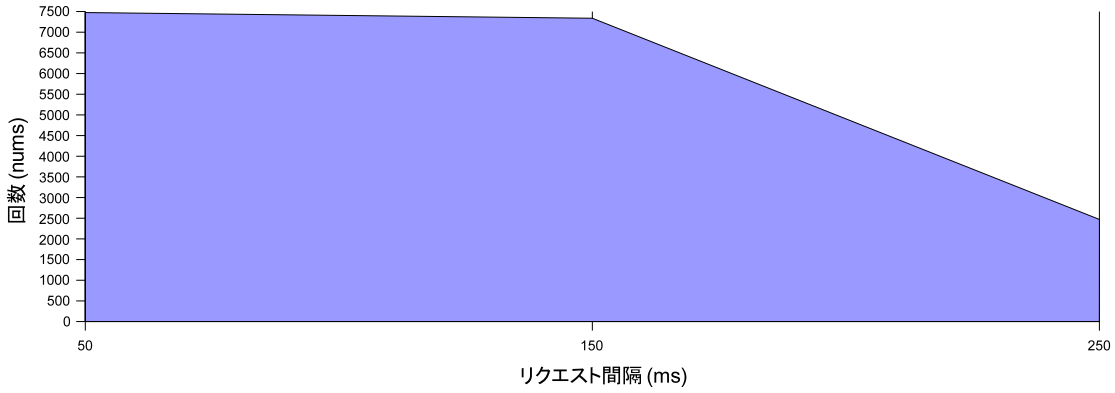


KeepAliveTimeout=500ms(最適値以下)設定でのリクエスト間隔

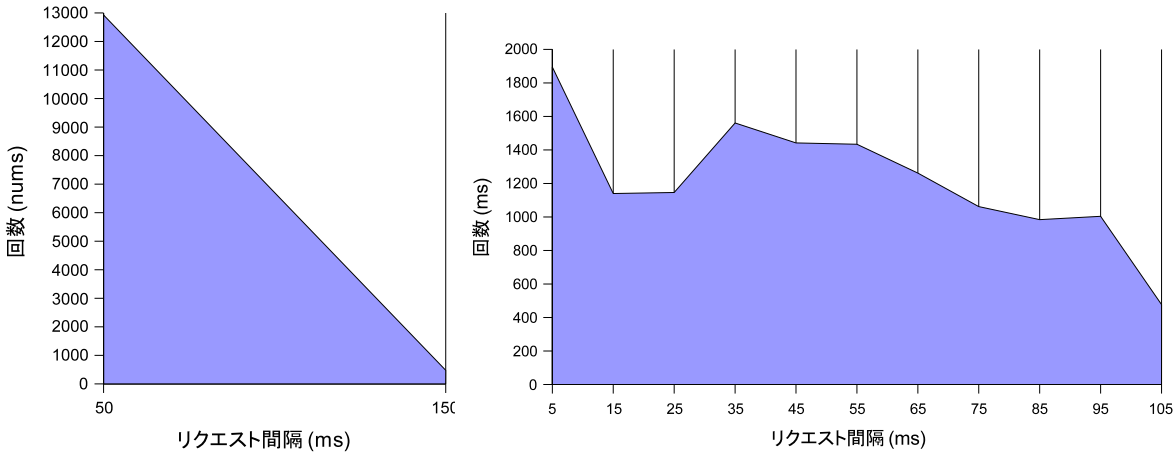
図 6.11: 100Mbps 帯域制限での KeepAliveTimeo とリクエスト間隔の関係



KeepAliveTimeout=15s(デフォルト)設定でのリクエスト間隔



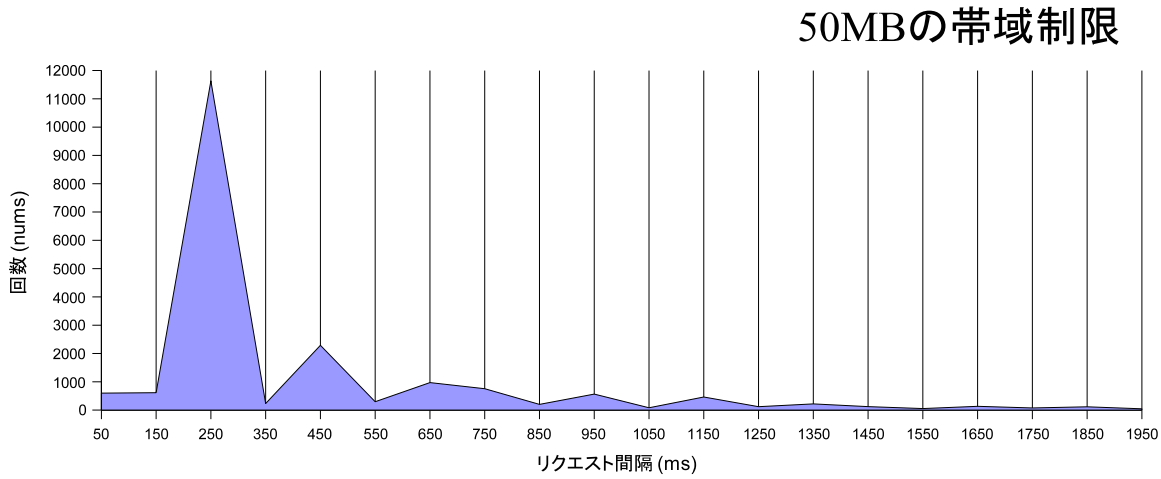
KeepAliveTimeout=300ms(最適値)設定でのリクエスト間隔



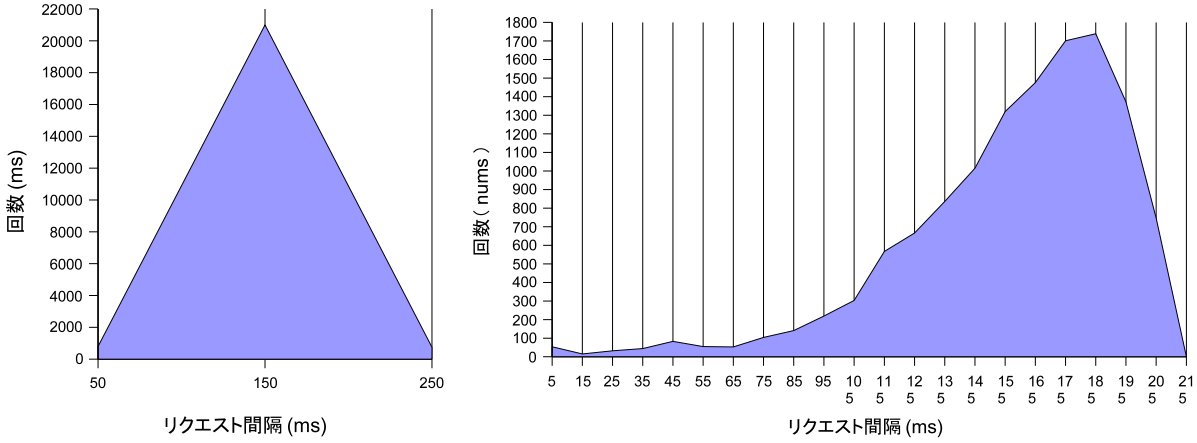
100ms間隔で計測　　10ms間隔で計測

KeepAliveTimeout=100ms(最適値以下)設定でのリクエスト間隔

図 6.12: 75Mbps 帯域制限での KeepAliveTimeo とリクエスト間隔の関係

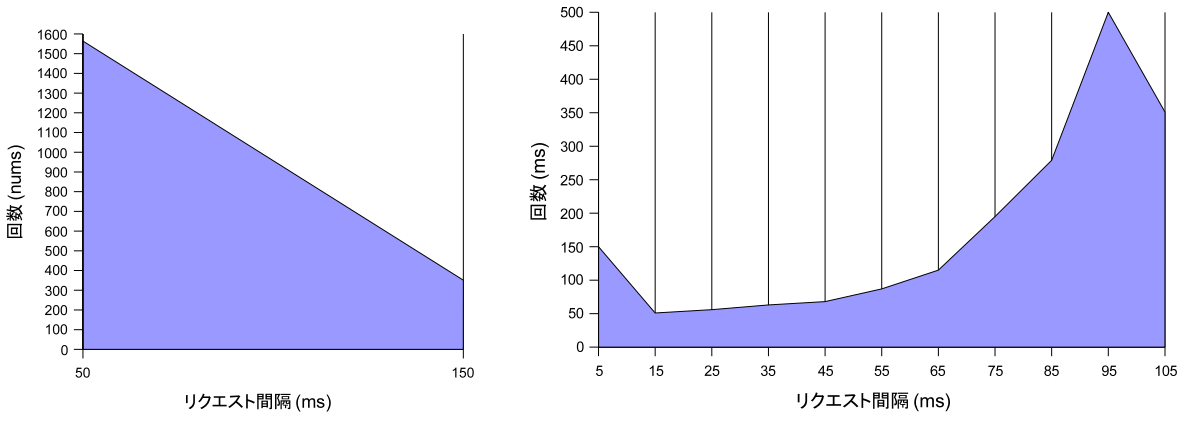


KeepAliveTimeout=15s(デフォルト)設定でのリクエスト間隔



100ms間隔で計測　　10ms間隔で計測

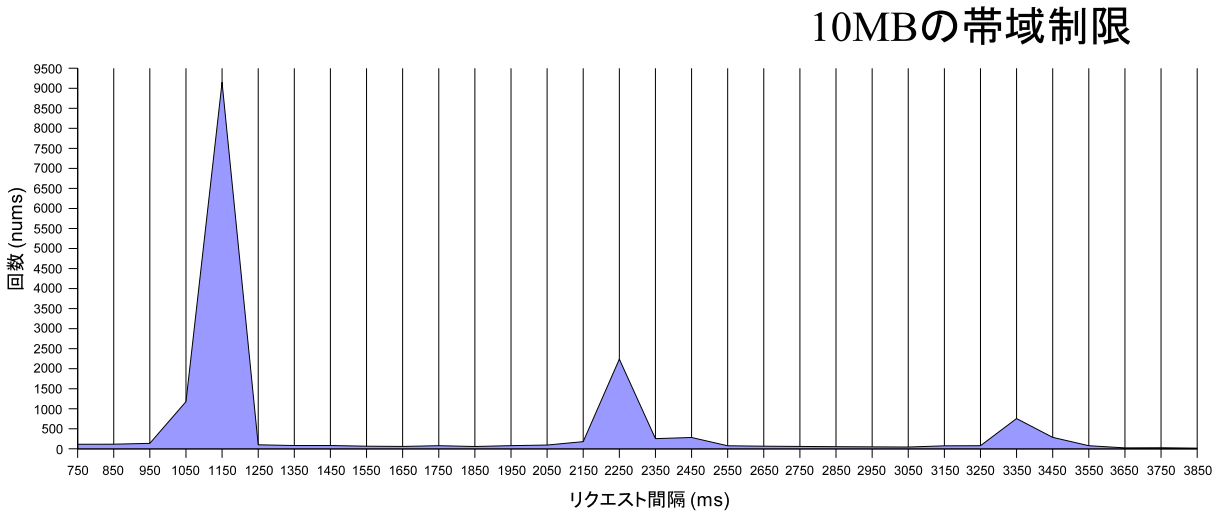
KeepAliveTimeout=200ms(最適値)設定でのリクエスト間隔



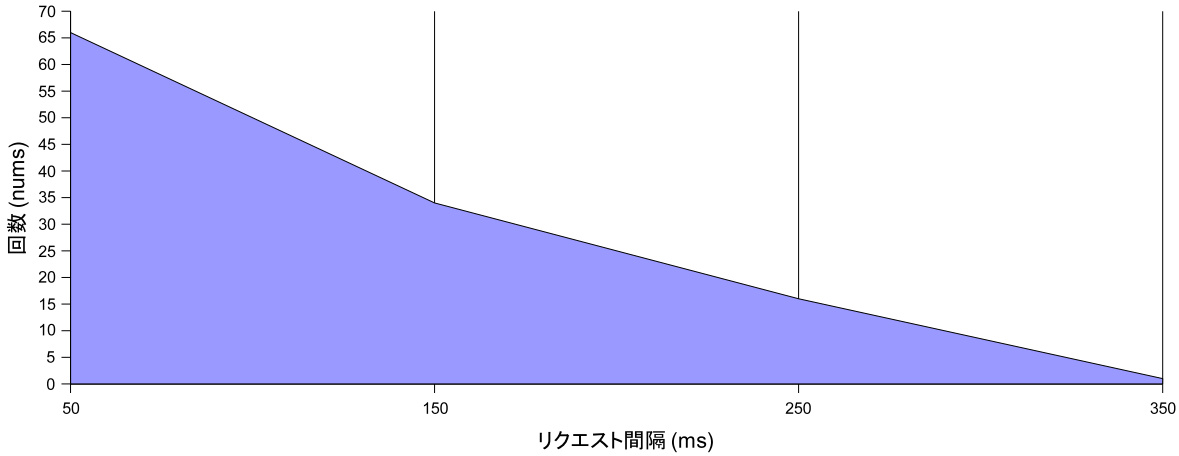
100ms間隔で計測　　10ms間隔で計測

KeepAliveTimeout=100ms(最適値以下)設定でのリクエスト間隔

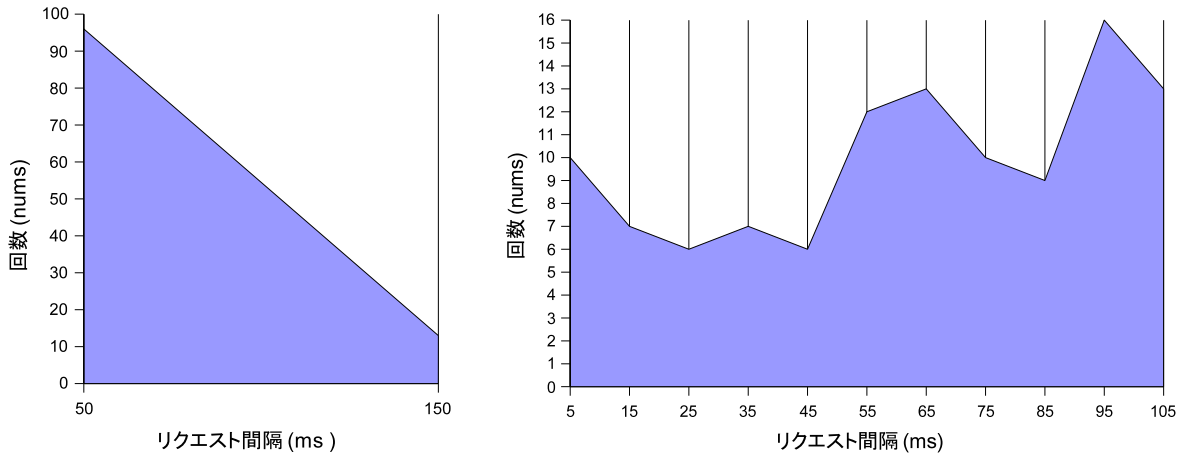
図 6.13: 50Mbps 帯域制限での KeepAliveTimeo とリクエスト間隔の関係



KeepAliveTimeout=15s(デフォルト)設定でのリクエスト間隔



KeepAliveTimeout=300ms(最適値)設定でのリクエスト間隔



100ms間隔で計測　　10ms間隔で計測

KeepAliveTimeout=100ms(最適値以下)設定でのリクエスト間隔

図 6.14: 10Mbps 帯域制限での KeepAliveTimeo とリクエスト間隔の関係

表 6.6: 実験で用いる帯域制限ごとの 3 種類の KeepAliveTimeout 値

帯域制限	デフォルト値	最適値	最適値以下
100Mbps	15 秒	1200m 秒	500m 秒
75Mbps	15 秒	300m 秒	100m 秒
50Mbps	15 秒	200m 秒	100m 秒
10Mbps	15 秒	300m 秒	100m 秒

- 図 6.13 の 1 番上のグラフのように、リクエスト間隔の分布に複数もしくは単峰の山ができる状況と、図 6.11 の 3 つの表のようにリクエスト間隔の回数の最頻値が 0〜100m 秒付近となり、以降、分布が右肩下がりしている状況 (単峰の山が右に大きく歪んでいる状況とも言える) の 2 パターンに分類することができる。

- 図 6.11〜6.14 すべてにおいて、最適値以下に KeepAliveTimeout 設定されている場合は、リクエスト間隔の最頻値が 0〜100m 秒付近となり、分布が右肩下がりしている状況である。

- 図 6.11 をのぞき、図 6.12、図 6.13、図 6.14 の一番上の表をみると、KeepAlive-Timeout の設定が最適値より長い場合は、リクエスト間隔の分布に複数の山ができている状態である。

今回の実験と、6.3.4 の実験結果より得られた KeepAliveTimeout とスループットの関係と照らし合わせることによって次のことが言える。

1. リクエスト間隔の分布に複数もしくは単峰の山ができる状況は、設定されている KeepAliveTimeout が十分長いと判断できる。つまり現状の KeepAlive-

Timeout よりも短い値にすることによって、スループットがあがる可能性が大きい。

2. リクエスト間隔の分布において度数の最頻値が 0〜100m 秒付近であるような単純下方している状況は、設定されている KeepAliveTimeout が短い、もしくは最適値付近と判断できる。よって現状の KeepAliveTimeout よりも長くすることによって、スループットがあがる可能性が少なからずともある。

6.3.6 KeepAliveTimeout の算出方法

6.3.4 節と 6.3.5 節の実験により、KeepAliveTimeout、リクエスト間隔、スループットの関係を知ることができた。これらの実験結果をもとに、6.1 式における重み付け量定数 α を決定した。重み付け量定数 α の求め方は以下のとおりである。

1. 6.3.4 節の実験から得られた図 6.7〜図 6.10 をもとに、各帯域別に 1 番よいスループットをだしているであろう KeepAliveTimeout(以降この部分での説明の間は BKAT:BestKeepAliveTimeout とする) を決める。

2. 各帯域の BKAT 値におけるリクエスト間隔の分布を得る。これは 6.3.5 節の実験よりすでに得られている。

3. BKAT 値におけるリクエスト間隔より、リクエスト間隔の平均値 μ と標準偏差 σ を帯域別にもとめる。

4. 各帯域ごとに、BKAT ＝ μ ＋ α σ を解き、4 種類の α を得る。

5. 4 種類の α の平均をとり、それを重み付け量定数 α として用いる。

今回、本システムでは、以上の方式より求めた α =7 を重み付け量定数として使用する。リクエスト間隔の平均と、標準偏差は、チューニング機構の取得ハンドラが取得したリクエストの分布をもとに、毎回計算してもとめる。

また、6.3.4 節と 6.3.5 節の実験により、リクエスト間隔の分布を見ることによって、現在設定されている KeepAliveTimeout よりも長く設定するのが良いのか、もしくは短く設定するのが良いのかの判断をつけられることも分かった。この結果より、本研究では以下のポリシーを提案する。KeepAliveTimeout のチューニング機構には、このポリシーを採用する。

1. リクエスト間隔の分布に複数もしくは単峰の山があらわれていたら、KeepAliveTimeout を現在の設定より短くなるように調整する。
2. リクエスト間隔の分布において度数の最頻値が 0〜100m 秒付近であるような単純下方している状況では、KeepAliveTimeout を現在の設定より長くなるように調整する。

2 の調整も行うことによって、誤って KeepAliveTimeout の最適値より短い値に設定してしまった場合にも修正がきくようになる。

KeepAliveTimeout を現状より短くする

KeepAliveTimeout を現状より短くする場合は、リクエスト間隔の分布から得られる平均と、標準偏差が小さければよい。よって、得られたリクエスト間隔の分布が 1 のポリシーにあてはまった場合、本システムでは次のようにしてリクエスト間隔の平均と標準偏差を求める。

- 取得ハンドラが得た全リクエスト間隔数を母集団とした場合、標本数を母集団の 70%とする。
- その標本数の平均と標準偏差を 6.1 式に用いる。

KeepAliveTimeout を現状より長くする

KeepAliveTimeout を現状より長くする場合は、標準偏差の重み付け量定数 α が 7 と十分に大きいため、取得ハンドラが得た全リクエスト間隔数の母集団をすべて標本数とすることで、現状の KeepAliveTimeout より長くなる。

よって、得られたリクエスト間隔の分布が 2 のポリシーにあてはまった場合、本システムでは次のようにしてリクエスト間隔の平均と標準偏差を求める。

- 取得ハンドラが得た全リクエスト間隔数を母集団とした場合、標本数を母集団の 100%とする。
- その標本数の平均と標準偏差を 6.1 式に用いる。

KeepAliveTimeout の設定範囲

今回、本システムでは、KeepAliveTimeout の最適値を算出を 100m 秒単位で行う。よって、算出された KeepAliveTimeout の最適値が 100m 秒以下であった場合は、自動的に 100m 秒に設定する。また、算出された KeepAliveTimeout の最適値が無限大に長くなってしまう可能性もあるので、動的に設定する KeepAliveTimeout の値の上限をもうけた。今回は、デフォルトの 15 秒を上限とする。

KeepAliveTimeout の最適値を算出するタイミング

KeepAliveTimeout の最適値を算出する解析ハンドラは一定期間ごとに作動する。本システムでは解析ハンドラが作動する指標として、一定のリクエストが到着したら解析を行うという方針をとった。これは、解析に必要なリクエスト間隔数が十分な量だけ取得できないといった状況を防ぐためである。

KeepAliveTimeout の分布の初期化

本システムでは、KeepAliveTimeout の最適値を算出し、KeepAliveTimeout の値を変更した際に、今まで記録してきたリクエスト間隔の分布数をすべてリセットし、新たな KeepAliveTimeout の値でのリクエスト間隔のみを記録していく。古い

リクエスト間隔の情報は考慮せず、最も新しい KeepAliveTimeout 値の環境下でのリクエスト間隔のみの情報で次の解析を行うためである。

以上の方針で、本システムは KeepAliveTimeout の最適値を算出する。

6.3.7 KeepAliveTimeout の自動調整

KeepAliveTimeout の調整ハンドラでは、分析ハンドラが算出した KeepAliveTimeout の最適値を Apache に適用する。

調整ハンドラが算出した KeepAliveTimeout の値は、本システムが独自に用意したスコアボードに格納しておく。新しい KeepAliveTimeout の値といったデータは、プロセス間で共有しなければならない。よって本システムでは、共有メモリを利用したプロセス間通信を行い、このスコアボードの情報を共有する。

Apache は起動した際に、設定ファイルの KeepAliveTimeout の値を内部に読み込み、子プロセスが各自持っている server_rec 構造体の keep_alive_timeout フィールドにセットする。server_rec 構造体とは、子プロセスごとの様々な情報を保持している構造体である。調整ハンドラでは、この子プロセスごとの keep_alive_timeout フィールドに、スコアボード上の KeepAliveTimeout の値をセットする。これにより、その子プロセスの KeepAliveTimeout が新しい値に設定し直される。

子プロセスの keep_alive_timeout フィールドの変更のタイミング

子プロセスは生成された時点では、設定ファイルに記述されている KeepAliveTimeout を、各自の keep_alive_timeout フィールドにセットしている。したがって、子プロセスがクライアントとコネクションをはってリクエストの処理を開始する前に、スコアボード上にある本システムが算出した KeepAliveTimeout の値を keep_alive_timeout フィールドにセットする必要がある。本システムでは、子プロセスがクライアントとコネクションをはる処理のフェーズにフックをかけ、子プロセスの keep_alive_timeout フィールドにスコアボード上の KeepAliveTimeout の値

をセットする。

調整ハンドラの有効範囲

本システムにおいて、1 つの子プロセスの KeepAliveTimeout の値を変更するタイミングは、クライアントとコネクションをはる時の 1 回だけである。よって、一度クライアントとコネクションをはり、処理を行いはじめた子プロセスに対しては、そのコネクションが切れるまで KeepAliveTimeout の変更は行えない。したがって、スコアボード上の KeepAliveTimeout が変更された後、その値を反映できるのは、スコアボード上の KeepAliveTimeout が変更された後に確立されたコネクションからである (図 6.15)。

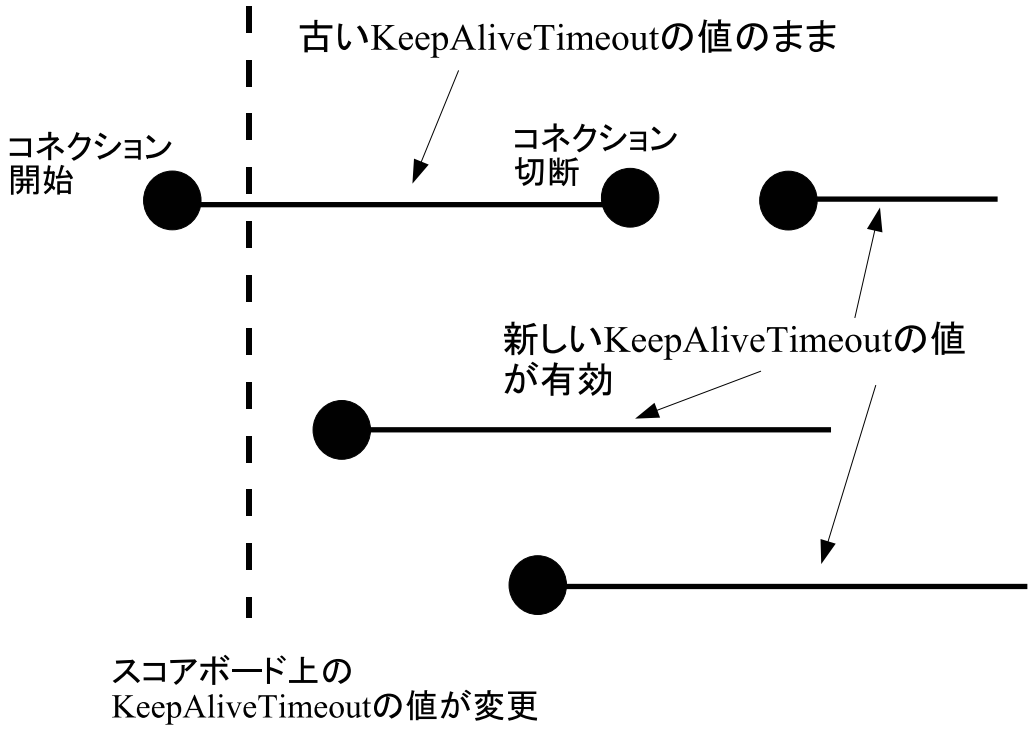


図 6.15: 調整ハンドラの有効範囲

6.4 同時接続数のチューニング

6.4.1 MaxClients

5.3.1 節で述べたようにサーバへの同時接続数に関するチューニングでは、多くの子プロセスの処理によってサーバがメモリ不足をおこさないようにサーバへの同時接続数を調整することが目的である。そのために、サーバへの同時接続数の最大値を決定している MaxClients のチューニングを行う。この MaxClients のチューニングには、Apache の子プロセスの消費メモリ量を取得し、Apache 全体のメモリ消費量を把握する。

6.4.2 子プロセスのメモリ消費量取得

ここでは、子プロセスのメモリ消費量の取得ハンドラについて述べる。本チューニング機構では、プロセスのメモリ情報を得るために、proc ファイルシステムを利用する。これにより、そのときに生成されているプロセスメモリ使用量を把握することができる。メモリ消費量の取得は、Apache の全ての子プロセスを対象とする。

proc ファイルシステムを利用して動的に子プロセスのメモリ状態を把握するためには、生成されている Apache の子プロセス数と、そのプロセス ID を動的に知る必要がある。この条件を満たすために、本システムでは、Apache サーバが内部に保持している scoreboard を利用する。この scoreboard とは、Apache の全プロセスに関する情報を格納しているものである。scoreboard は、scoreboard.h のヘッダファイルを取り込むことで、どのプロセスからも情報の参照・変更を行うことができる。本システムでは、この scoreboard に格納されている次の情報を使用する。

- 現時点で生成されている子プロセスの数
- 生成されている子プロセスのプロセス ID

この 2 つの情報と、proc ファイルシステムを用いて、生成されている Apache の子プロセスの全消費メモリ量を取得する。また、scoreboard に格納されている親プロセス ID より、親プロセスの消費メモリ量も取得する。

メモリ消費量取得のタイミング

本システムでは、一定時間ごとにこの取得ハンドラが動作するように実装した。これは、module による Apache の内部処理以外に、常に proc を利用してファイル読み込みを行っていると、オーバヘッドが大きくなってしまうことが予想されるためである。

6.4.3 Apache 全体の消費メモリの算出

本システムでは、MaxClients の最適値を

$$\textit{Apache}\text{が使用可能なメモリ量} \qquad (6.2)$$
$$\text{子プロセスのメモリ消費量} \times \textit{MaxClients} + \text{親プロセスのメモリ消費量}$$

とする。子プロセスのメモリ消費量の取得ハンドラで取得した、子プロセスの消費メモリ量と親プロセスのメモリ消費量の総和を求める。

Apache が使用可能なメモリ量

本システムは、Apache が使用可能なメモリ量を動的取得する実装にはなっていない。本システムのモジュール対応の設定ファイル経由で、Apache で使用できるメモリの最大量を KByte 単位で指定する方式を取っている。

MaxClients の最適値を算出するタイミング

MaxClients の最適値を算出する解析ハンドラは、メモリ消費量取得ハンドラに連動して作動する。

MaxClietnts の最大値

現在の Apache の仕様では、Apache の内部で MaxClietnts の最大値は 256 とされている。これ以上の値を設定ファイルで指定しても、実際には 256 個以上の子プロセスは生成されない。よって、解析ハンドラが MaxClients の最適値を 256 以上と算出した場合は、自動的に 256 と修正する。

6.4.4 MaxClients の自動調整

MaxClients の調整ハンドラでは、解析ハンドラが算出した MaxClients の最適値を Apache に適用する。解析ハンドラが算出した MaxClients の最適値は、KeepAlive-Timeout のときと同様に、本システムが独自に用意したスコアボードに格納し、全てのプロセスで情報を共有する。

Apache の子プロセス管理

Apache の子プロセスの管理は、prefork という独立したモジュールが行っている。この prefork は、子プロセス数が少なければ、fork() して子プロセスを生成したり、クライアントとのコネクションを確立させるのを待っている待機プロセスが多ければ、プロセスを kill() して停止したりする。Apache の起動時に、prefork は Apache の設定ファイルに書かれている MaxClients などの子プロセスに関する設定を読み取り、prefork の内部変数に格納する。prefork はその格納した情報をもとにプロセスの管理を行うが、他のモジュールからそれらの情報を動的に変更することができない。

よって本システムでは、算出した MaxClients の最適値以上のプロセスがクライアントとコネクションをはらないように、kill() システムコールによってプロセスを停止させて、プロセス数の調整を行う。prefork は常駐してプロセスの管理を行っているため、本調整機構も常に作動させる。

子プロセスの停止

子プロセスは生成されると、0 から順に Apache 内での識別 ID がふられる。本システムはこの識別 ID を利用し、MaxClients の最適値以上の識別 ID をもつ子プロセスを停止させる。

しかし、MaxClients の最適値以上の識別 ID をもつ子プロセスを全て強制的に停止させるのは危険である。なぜなら、MaxClients の最適値以上の識別 ID をもつ子プロセスの中に、すでクライアントとコネクションをはり、リクエストの処理を開始しているかもしれない。そのようなプロセスを停止させてしまっては、クライアントとのデータのやり取りも停止してしまう。

そこで Apache 内部の scoreboard に格納されている各子プロセスの状態を参照し、その子プロセスを停止させるかどうか判定する。本システムでは、子プロセスが次の状態である場合のみ、その子プロセスを停止させる。

- 子プロセスの各種初期化処理中
- クライアントとのコネクションをはるのを待っている状態

本調整ハンドラは、prefork の動作には介入しない。よって、調整ハンドラが子プロセスを停止させたら、prefork は子プロセスが少ないと判断し、新たに子プロセスを生成するかもしれない。しかし、調整ハンドラはつねに作動しているため、上記の子プロセスを停止する動作を繰り返す。よって、prefork が fork() して子プロセスを生成、調整ハンドラが kill() して子プロセスを停止といった動作が繰り返されることになる (図 6.16)。

6.5 チューニングのログ

本システムによるチューニングを行った際、Web 管理者は、システムがパラメータをどの値にチューニングして変更したかを知りたいと思うこともある。よって、

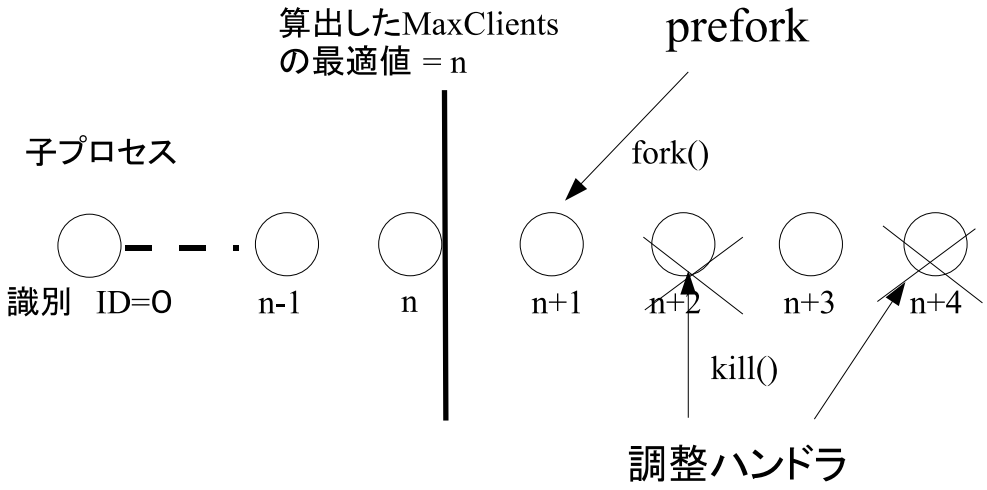


図 6.16: MaxClients のチューニングの様子

本システムは、チューニングを行った結果をログフィルに書き出す。2 つのパラメータチューニングのログは図 6.17、図 6.18 のようになる。

```
2006 01 16 15:14(+340) from 4700(ms) to 3100(ms)
2006 01 16 15:20(+343) from 3100(ms) to 2800(ms)
2006 01 16 15:14(+354) from 2800(ms) to 2500(ms)
```

図 6.17: KeepAliveTimeout のチューニングのログ

```
2006 01 16 20:07(+20) from 39 to 43    Daemons 24
2006 01 16 20:27(+20) from 43 to 50    Daemons 22
2006 01 16 20:48(+20) from 50 to 23    Daemons 20
```

図 6.18: MaxClients のチューニングのログ

ログの最初に出力されているのは、チューニングが行われた日付けと時刻である。

その次に記録されているのは、前回のチューニングからどれくらいの時間が経過したか、秒単位で記録されている。そして、変更前のパラメータ値と、変更後のパラメータ値が記録されている。MaxClients のログ最後に記録されている Daemons とは、その時に生成されている子プロセスの識別 ID の最大数である。

第 7 章

評価実験と考察

7.1 KeepAliveTimeout のチューニングの評価実験

本システムの、Keep–Alive 接続に対するチューニング機構が、サーバの負荷に応じて動的に最適な KeepAliveTimeout 値を算出し、自動的にパラメータの値を変更できることを示すための評価実験を行った。

ここでは 2 通りの実験を行った。1 つは、6.3.4 節の実験で用いたようなサーバへの負荷が一定である場合に、本システムの Keep–Alive 接続に対するチューニング機構がどのような自動設定を行っていくかを評価する実験である。もう 1 つは、サーバへの負荷が変化した場合におけるチューニング機構の自動設定を評価する実験である。

7.1.1 サーバへの負荷が一定

実験環境

6.3.4 節の実験と同様の環境で実験を行った。

実験に用いる負荷

6.3.4 節の実験と同様のリクエストのシナリオを用いた。今回は、本チューニング機構が自動設定する KeepAliveTimeout の値が、一定の値に落ちつくまでベンチ

マークテストを行った。KeepAliveTimeout のチューニングを行うタイミングは、サーバにリクエストが 5000 回到着するごとに行うものとする。

実験は、6.3.4 節の実験と同様に 100Mbps、75Mbps、50Mbps、10Mbps の 4 種類の帯域別にそれぞれ行い、負荷がことなる場合でも本システムが有効であるかどうか評価する。

7.1.2 実験結果

サーバとクライアント間のネットワークの帯域が 100Mbps、75Mbps、50Mbps、10Mbps のときの実験結果を述べる。各実験の評価は、Apache の KeepAliveTimeout がデフォルトの 15 秒で稼動している場合と、本システムが作動しているときの平均スループットの比較によって行う。

100Mbps の帯域制限時

負荷実験の時間は 5 分である。結果を表 7.1 に示す。

表 7.1: 負荷が一定時の結果 (100Mbps の帯域制限)

KeepAliveTimeout=15 秒	103(Request/秒)
本システム作動時	111(Request/秒)

本チューニング機構が行ったチューニングの様子を表 7.2 に示す。

表 7.1 の結果より、本システムを稼動することによって、スループットは 1.07 %ほどあがった。スループットがあまりあがらなかったのは、図 6.7 を見るとおり、デフォルトの 15 秒と最適値の 8 秒のスループットの差があまりなく、デフォルトの 15 秒でも十分よいスループットがでる状況であったためである。表 7.2 の結果より、本システムのチューニングによる KeepAliveTimeout の値は、おおよそ

6600m 秒前後に収まっていっているのがわかる。図 6.7 のスループットと比べて、KeepAliveTimeout の値が 6600 前後でサーバが稼動するのは良いと言える。

表 7.2: 動的変更による KeepAliveTimeout の変化 (100Mbps の帯域制限)

チューニング回数	前回の変更からの経過時間 (秒)	変更前 (m 秒)	変更後 (m 秒)
1	44	15000	5700
2	45	5700	6700
3	45	6700	6600
4	45	6600	6700
5	45	6700	6600
6	45	6600	6700

75Mbps の帯域制限時

負荷実験の時間は 6 分である。結果を表 7.3 に示す。

表 7.3: 負荷が一定時の結果 (75Mbps の帯域制限)

KeepAliveTimeout=15 秒	94(Request/秒)
本システム作動時	97(Request/秒)

本チューニング機構が行ったチューニングの様子を表 7.4 に示す。

表 7.3 の結果では、本システムを稼動ではスループットは 1.03 %しかあがらなかった。しかし、表 7.4 の結果では、本システムのチューニングによる KeepAliveTimeout

の値は、おおよそ 400m 秒前後に収まっていっているのがわかる。図 6.8 のスループットと比べて、KeepAliveTimeout の値が 400 前後でサーバが稼動するのは良いと言える。

表 7.4: 動的変更による KeepAliveTimeout の変化 (75Mbps の帯域制限)

チューニング回数	前回の変更からの経過時間 (秒)	変更前 (m 秒)	変更後 (m 秒)
1	53	15000	900
2	54	900	400
3	53	400	300
4	53	300	400
5	61	400	400
6	53	400	400
7	53	400	400

50Mbps の帯域制限時

負荷実験の時間は 12 分である。結果を表 7.5 に示す。

表 7.5: 負荷が一定時の結果 (50Mbps の帯域制限)

KeepAliveTimeout=15 秒	13(Request/秒)
本システム作動時	15(Request/秒)

本チューニング機構が行ったチューニングの様子を表 7.6 に示す。

表 7.5 の結果より、本システムを稼動することによって、スループットは 1.15 %ほどあがった。表 7.6 の結果より、本システムのチューニングによる KeepAliveTimeout の値は、おおよそ 200m 秒前後に収まっていっているのがわかる。図 6.9 のスループットと比べて、KeepAliveTimeout の値が 200 前後でサーバが稼動するのは良いと言える。

表 7.6: 動的変更による KeepAliveTimeout の変化 (50Mbps の帯域制限)

チューニング回数	前回の変更からの経過時間 (秒)	変更前 (m 秒)	変更後 (m 秒)
1	150	15000	1900
2	132	1900	200
3	91	200	200
4	100	200	200
5	105	200	200
6	82	200	200

10Mbps の帯域制限時

負荷実験の時間は 40 分である。結果を表 7.7 に示す。

本チューニング機構が行ったチューニングの様子を、表 7.8 に示す。

表 7.7 の結果より、本システムを稼動することによって、スループットは 1.24 %ほどあがった。表 7.8 の結果より、本システムのチューニングによる KeepAliveTimeout の値は、おおよそ 2500m 秒前後に収まっていっているのがわかる。図 6.10 のスルー

表 7.7: 負荷が一定時の結果 (10Mbps の帯域制限)

KeepAliveTimeout=15 秒	65(Request/秒)
本システム作動時	81(Request/秒)

プットと比べて、KeepAliveTimeout の値が 2500 前後でサーバが稼動しているのは、デフォルトの 15 秒より良いスループットがでる範囲ではあるが、現状より KeepAliveTimeout の値を小さくすればもっとスループットはあがる状況である。つまり、最適な値を算出しているチューニングではない。

表 7.8: 動的変更による KeepAliveTimeout の変化 (10Mbps の帯域制限)

チューニング回数	前回の変更からの経過時間 (秒)	変更前 (m 秒)	変更後 (m 秒)
1	345	15000	4700
2	340	4700	3100
3	343	3100	2800
4	354	2800	2500
5	334	2500	2900
6	355	2900	2800
7	340	2800	2900

7.1.3 サーバへの負荷が変動

ここでは、サーバへの負荷が変動する場合における本システムの有効性についての評価実験を行う。

実験環境

本実験には、負荷テストソフトと、サーバのスループットを一定時間ごとにモニタリングし、スループットを測定するベンチマークソフトの 2 種類を同時に使用した。負荷を変化させてサーバのスループットを測定する方法は以下のとおりである。

- スループット測定用のベンチマークソフトは、一定のリクエストをサーバに投げながらスループットを測定し続ける。
- 一定期間だけ負荷テストソフトでサーバに大量のリクエストをなげ負荷を上げる。

今回、スループットの測定には httperf を使用した。この httperf は、測定対象のマシンのスループットを 5 秒ごとに表示することができる。また、一定期間だけサーバに負荷をかける負荷テストソフトには Microsoft Web Application Stress Tool を使用した。実験環境を図 7.1 に示す。

実験は、6.3.4 の実験と同様に 100Mbps、75Mbps、50Mbps、10Mbps の 4 種類の帯域別にそれぞれ行い、負荷がことなる場合でも本システムが有効であるかどうか評価する。

実験に用いる負荷

サーバのスループットを測定する httperf の負荷は以下のとおりである。

- 1 秒間に生成するコネクション数を、100Mbps では 700、75Mbps では 500、50Mbps では 400、10Mbps では 300 とする。各コネクション数は、その帯域の httperf のテストにおいてサーバに対する負荷が大きすぎない値である。

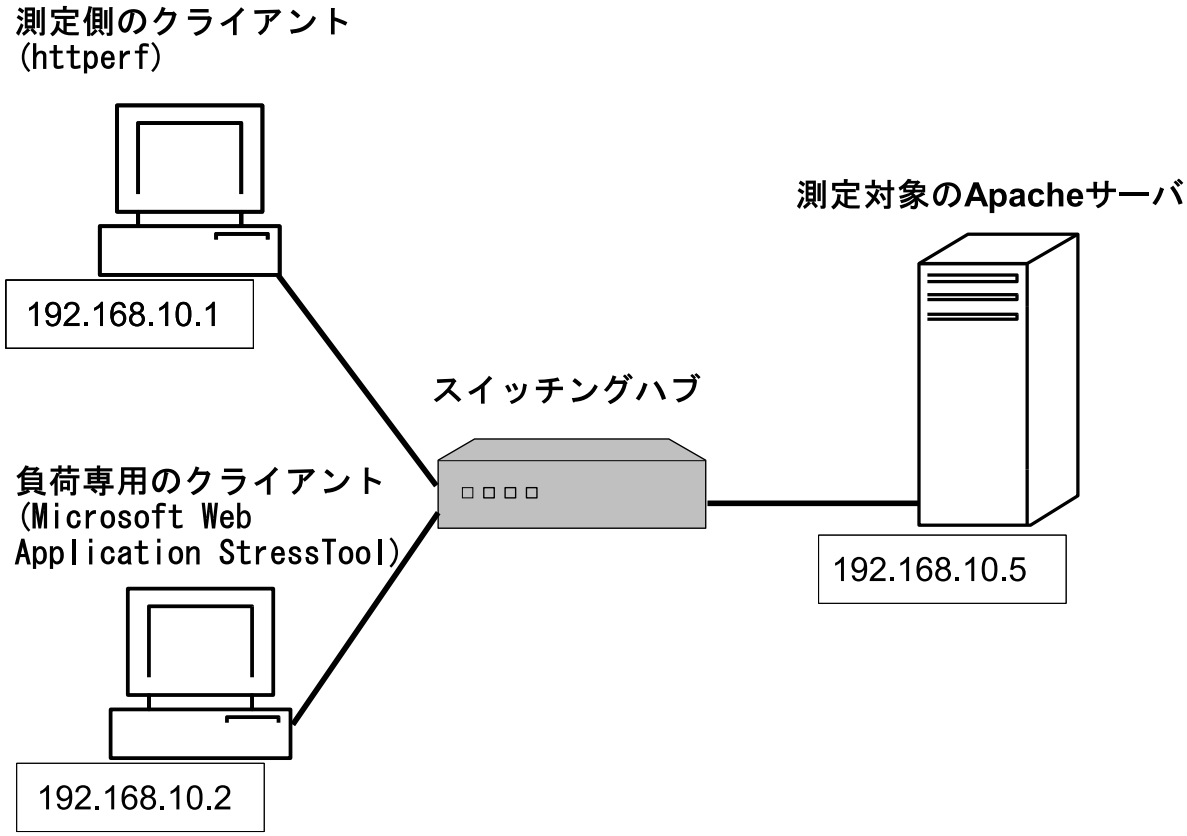


図 7.1: 負荷を変化させる実験の環境

- Keep–Alive 接続している 1 つのコネクション内で 3 つのリクエストを処理する。

実験では、httperf でサーバのスループットを測定し、途中で Microsoft Web Application Stress Tool からリクエストを 1 分間送り負荷をかける。Microsoft Web Application Stress Tool でかける負荷は 6.3.4 の実験と同様のリクエストのシナリオである。

7.1.4 実験結果

サーバとクライアント間のネットワークの帯域を 100Mbps、75Mbps、50Mbps、10Mbps と変えて測定したときの各実験結果は図 7.2〜図 7.5 のようになった。各図の下にに示してある矢印は、本チューニング機構が作動し、KeepAliveTimeout の値をチューニングした時間である。

図 7.2〜図 7.5 の結果に見られるように、Microsoft Web Application Stress Tool による 1 分間の負荷がかかっている間は、全ての環境においてスループットは極端にさがっている。本システムの稼動中も、未稼働時同様にスループットはさがっているが、未稼働時よりも高いスループットを保っているのが確認できる。

7.1.5 2 つの実験結果についての考察

本システムによる KeepAliveTimeout の値のチューニングの評価実験を、負荷が一定の場合と、変動する場合の 2 通り行った。結果より、本システムを使用することによってデフォルトの設定でサーバを稼動させた場合よりも良いスループットが出ていることがわかる。ただし、7.1.1 節の実験結果では、本システム稼動によりスループットはあがったが、大幅なスループットの改善にはいたっていない。これは、本システムのモジュールの動作によるオーバーヘッドが関係していると考えられる。

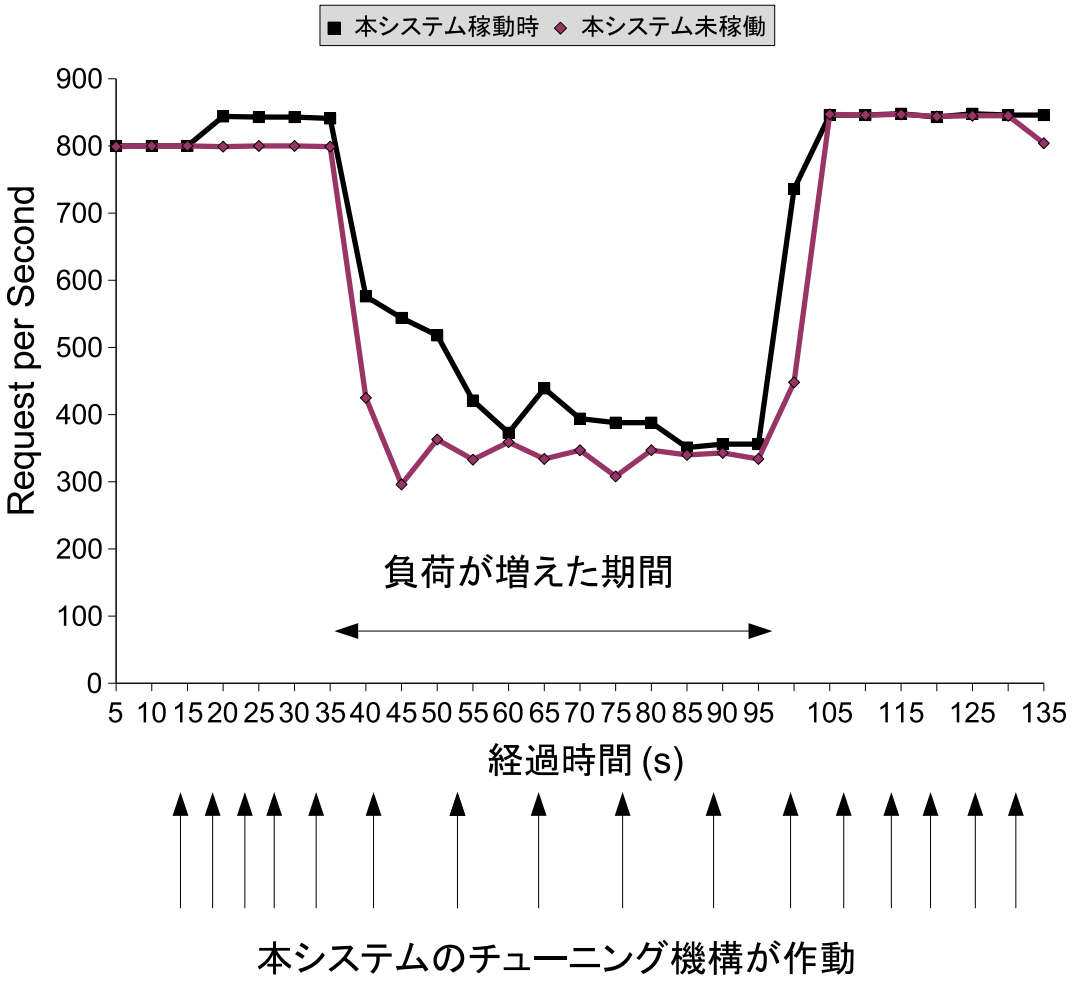


図 7.2: 負荷を変動させた場合の結果 (100Mbps の帯域制限)

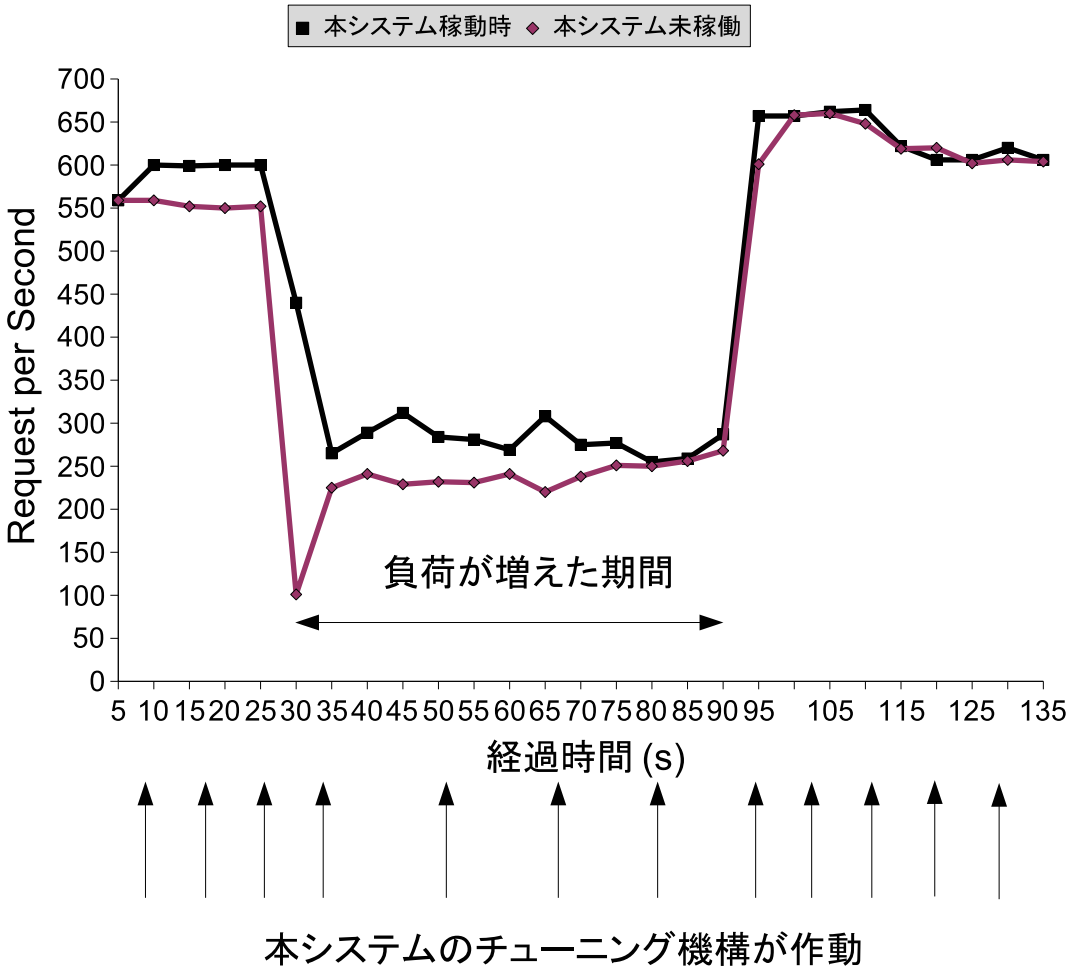


図 7.3: 負荷を変動させた場合の結果 (75Mbps の帯域制限)

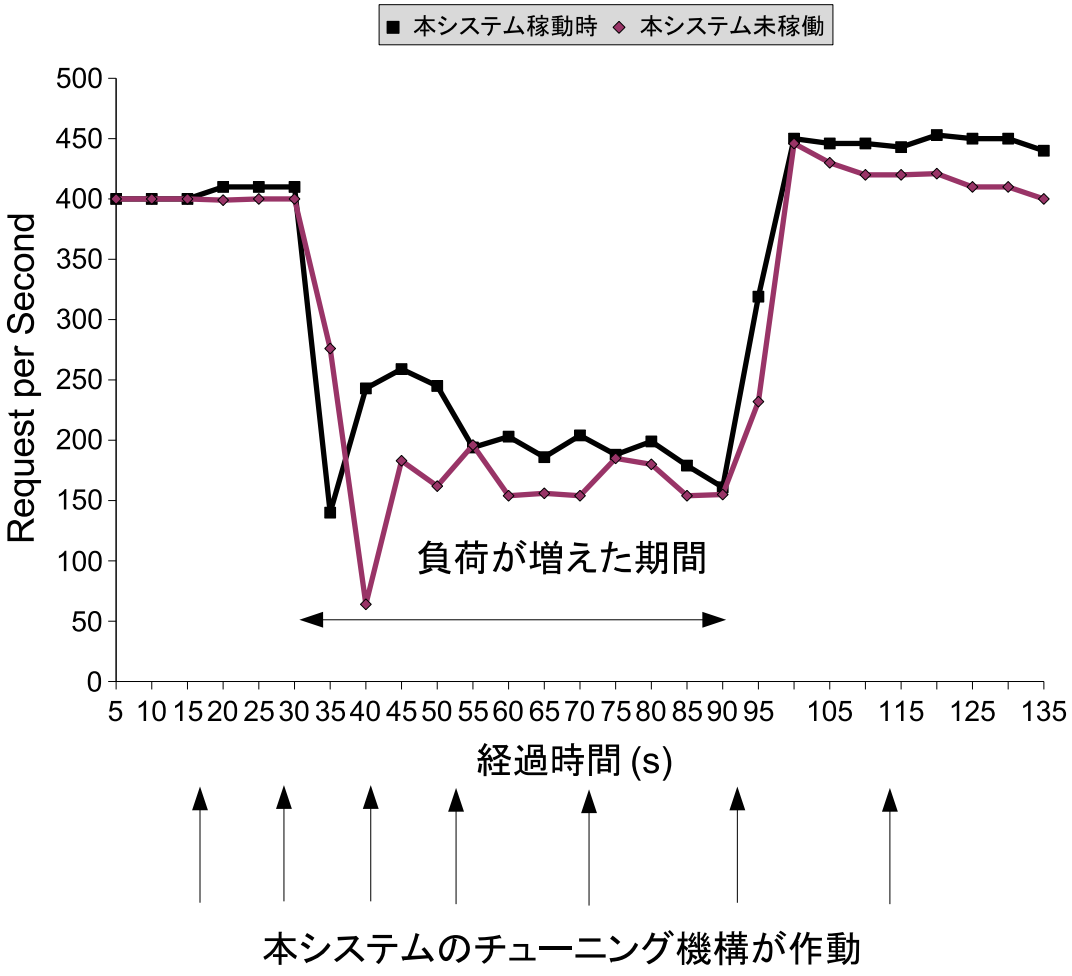


図 7.4: 負荷を変動させた場合の結果 (50Mbps の帯域制限)

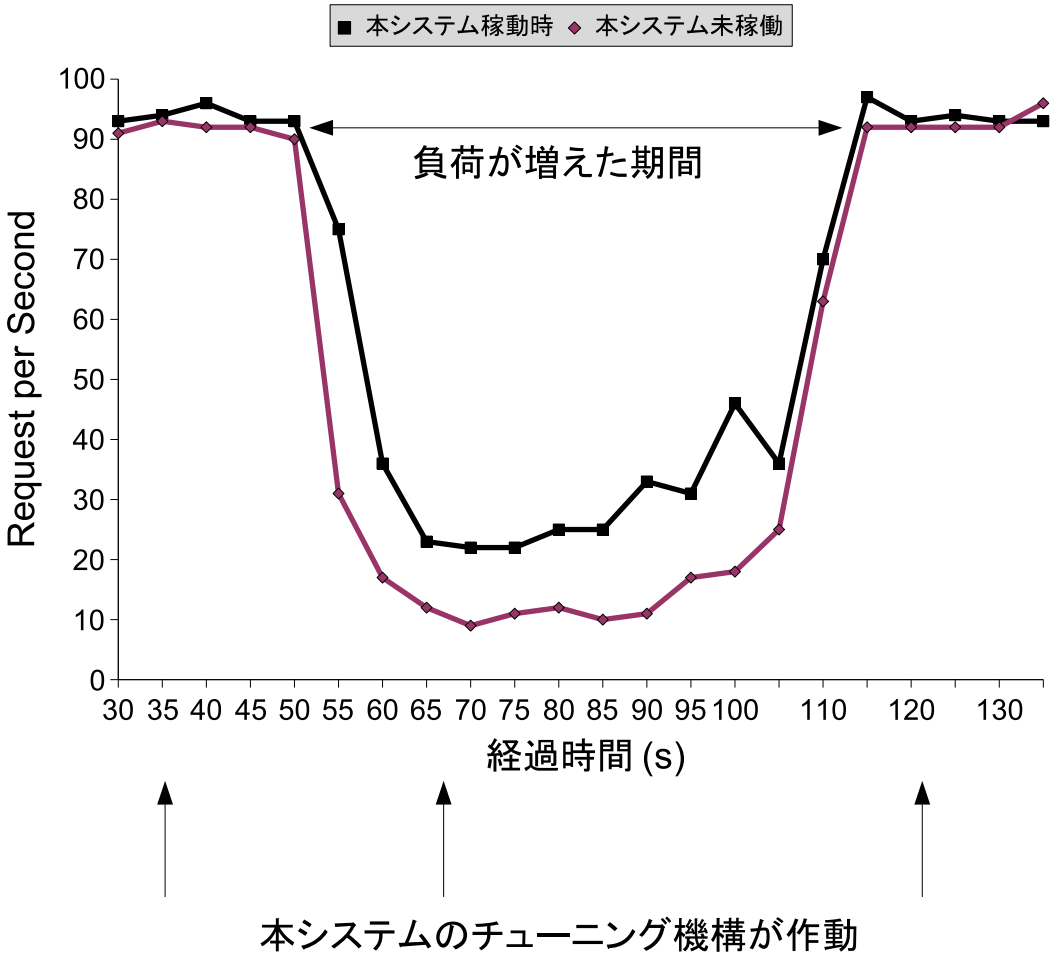


図 7.5: 負荷を変動させた場合の結果 (10Mbps の帯域制限)

しかし、7.1.3 節の実験のようなサーバへの負荷が変化する状況では、急激なサーバへの負荷がかかった際に、本システム稼動時と未稼動時ではスループットのさがり具合に大きな差がみうけられる。よって、本システムはサーバへの負荷が変化するような、実世界により近いサーバ環境において効果を発揮するものだと考えられる。

7.2 同時接続数のチューニングの評価実験

本システムの、MaxClients に対するチューニング機構が、サーバへの負荷に応じて適切な MaxClients の上限値を算出して、動的に子プロセス数を調節し、サーバへの負荷を緩和しているかどうかを示すための評価実験を行った。

7.2.1 動的コンテンツ配信のサーバ

実験は、子プロセス数の消費メモリが大きい動的コンテンツ配信のサービスを行うサーバに対して行った。

実験環境

6.3.4 節の実験と同様の環境で実験を行った。

実験に用いる負荷

実験では、Apache に mod_perl モジュールを Apache に動的に組み込んだ。動的なコンテンツ生成の処理は、この mod_perl モジュールが担当する。

本実験では、クライアントからの要求に応じて、動的に画像を生成し配信する CGI プログラムを用いた。負荷テストツールには、Microsoft Web Application Stress Tool を使用した。Microsoft Web Application Stress Tool からは、CGI プログラムに対して 1 クライアントが 1 つのリクエストをなげる。本実験では、KeepAliveTimeout の実験で用いていたサーバとクライアント間のネットワークの帯域制限は行わない。

Microsoft Web Application Stress Tool から同時接続 200 の設定でサーバに負荷を 2 分間かけ続け、サーバ全体のメモリ消費と、クライアント (Microsoft Web Application Stress Tool) 側でのサーバのスループットを評価する。

実験結果

今回の実験では、本システムが 20 秒ごとにチューニングを行う。管理者が設定する Apache 全体の使用メモリ量を 150MB、Apache の設定ファイルに記述する MaxClients の設定値をデフォルトの 150 とした。

負荷テスト中のサーバの消費メモリを表 7.9 に、発生したスワップイン・スワップアウトの数を表 7.10 に示す。

表 7.9: 本システム稼動時と未稼働時のメモリ消費量

	搭載メモリ量	未稼動時	本システム稼働時
メモリ (MB)	250	244	187
スワップメモリ (MB)	510	151	126
合計使用メモリ		395	313

表 7.10: 10 秒間に発生したスワップイン、スワップアウトの数

	未稼動時	本システム稼働時
スワップイン	7634	4344
スワップアウト	14627	9974

また、Microsoft Web Application Stress Tool で計測されたサーバのスループットを表 7.11 に示す。本システムが行った MaxClients の調整の様子を表 7.12 に示す。

表 7.11: 本システムの稼動時と未稼働時のサーバのスループットの違い

	未稼働時	本システム稼動時
スループット (Request/秒)	2.38	1.21

表 7.12: 動的変更による MaxClients の変化 (抜粋)

前回の変更からの経過時間 (秒)	変更前 (nums)	変更後 (nums)
20	80	30
20	30	26
20	26	70
20	70	50

7.2.2 実験結果についての考察

動的コンテンツを要求する負荷テストを行った結果、Apache の子プロセスをデフォルトの 150 個生成させて処理をさせると、サーバ内のメモリを大量に消費していることが表 7.9 からもわかる。表 7.10 にあるように、大量のスラッシングが発生し、実際はサーバの挙動は不安定なものとなっていた。本システムを稼動させて、サーバへの同時接続数を調整しながら処理をさせると、システムを動かしていない場合と比べて、2 割ほど消費メモリ (スワップ領域も含めて) は減っている。よって、サーバのメモリ不足を多少は解消していると言える。

しかし、表 7.11 にあるように、クライアント側から見た場合のサーバのスループットは落ちている。これは、本システムが Apache の子プロセス数を調整し減らしたためによるものである。トレードオフの関係にあるサーバのシステムの安定性と、クライアントからみたサーバのスループットのどちらに優先度をつけるかは、サーバの管理者次第である。しかし、サーバをダウンさせるような状況を避けたいのであったら、クライアントからみたスループットを多少なりとも下げて、システムの安定性を取るべきである。

7.3 本システムの今後の課題

本システムは、Apache の KeepAliveTimeout と MaxClients のパラメータを動的にチューニングする 2 つの Apache モジュールである。静的なコンテンツ配信部に関しては KeepAliveTimeout のチューニングを行い、動的なコンテンツ配信部には MaxClients のチューニングを行った。各パラメータのチューニングの評価実験では、デフォルトの設定値で Apache サーバを稼動させた時よりも、本システム稼動時はサーバのスループットは概ね向上している。しかし、パラメータの調整方法には改善の余地があると思われる。

現在は、1 つのパラメータの調整に用いる動作情報が 1 つの、1 入力 1 出力の

チューニングを行っているが、今後は、2つのモジュールがお互いに連携しあった、2つのパラメータ間の影響を考慮したチューニングも検討すべきである。

第 8 章

おわりに

本研究では、静的・動的コンテンツ配信を行う Web アプリケーションサーバにおける動的なチューニングシステムを設計し、Apache モジュールとして実装した。チューニング対象は Apache2.0 系のサーバとし、チューニングを行うパラメータは、クライアントとサーバ間のコネクションに関する KeepAliveTimeout と、サーバのプロセスに関する MaxClients の 2 つである。この 2 つのパラメータの動的チューニングの評価実験を行い、本システムの有効性を確認した。

本システムは、サーバの稼動時の動作情報をもとに各パラメータの最適値を算出して、チューニングを行うものである。KeepAliveTimeout のチューニングの動作情報には、サーバとクライアントの 1 コネション内におけるリクエストの様子を使用した。MaxClients のチューニングの動作情報には、プロセスの消費メモリを使用した。本研究の KeepAliveTimeout の最適値の算出法は、予備実験をもとに考案したものである。

評価実験より、本システムを用いたサーバのパラメータチューニングが有効であるといえる。よって、本システムの利用により、サーバ管理者のパフォーマンスチューニングの作業負担を削減することができる。

本システムを、サーバ内で常に稼動させてパラメータのチューニングを行わせる状況を仮定している。だが他にも、新たにサーバを導入させたときに、一時的にサーバ内で稼動させて本システムのチューニングのログから、そのサーバの状況にあったパラメータの最適値を求めるといった利用法もある。

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。

指導教員の多田好克先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。また、ご多忙中にもかかわらず本論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。また、本研究室の佐藤喬助手には、研究方針や研究内容に関して多くのご指導をいただきました。ここに厚く御礼申し上げます。

そして、本研究が行なえたことは、研究方針や方法論について議論をし、共に研究生活をおくってきた多田研、そして村山研の学生諸氏おかげでもあります。最後に、これらの皆さんに感謝いたします。

参考文献

[1] Netcraft, : “Netcraft”, http://news.netcraft.com/.

[2] Laurie, B. and Laurie, P.: “Apache ハンドブック”, オライリー・ジャパン (2003).

[3] Berners-Lee, T., Fielding, R. and H.Frystyk, : “Hypertext transfer protocol http/1.0”, in *RFC 1945 MIT/LCS,UC Irvine* (1996).

[4] Apache, : “Apache HTTP サーバ バージョン 2.0 ドキュメント”, http://httpd.apache.org/docs/2.0/.

[5] 杉木章義, 河野健二：“性能パラメータの自動調整による web サーバの性能向上”, 情報処理学会 OS 研究会報告 2005-OS-99, pp. 29–36 (2005).

[6] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S. and Tilbury, D. M.: “Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server”, in *IEEE/IFIP Network Operations and Management Symposium*, No. 8, pp. 219–234 (2002).

[7] 中村豊, 知念賢一, 山口英, 砂原秀樹：“パケットモニタによる www サーバ挙動観測方法の提案”, インターネットコンファレンス’98 論文集, pp. 59–66 (1998).

[8] 清水淳也, 山根敏志, 加藤整, 中村理之：“有向グラフの昇順探索に基づく web システムのボトルネック検出法”, in *IBM PROVISION*, No. 44, pp. 84–89 (2005).

[9] Sun Microsystems, : “BigAdmin: DTrace”, http://www.sun.com/bigadmin/content/dtrace/.

[10] Hut, Y., Nandat, A. and Yangt, Q.: “Measurement, analysis and performance improvement of the apache web server”, in *Proc 18th IEEE int’l Performance Computing and Cmmunications Confrence*, pp. 261–267 (1999).

[11] 小山浩之：“Web エンジニアのための Apache モジュール プログラミングガイド”, 株式会社技術評論社 (2003).

[12] Microsoft, : “MS Web Application Stress ツール”, `http://www.microsoft.com/`.

[13] L.Urner, D.: “Pinpointing system performance issues”, in *USENIX:1997 LISA XI October 26-31*, pp. 141–154 (1997).