リクエスト間隔を考慮したウェブサーバの keep-alive 時間の自動設定

Automatic tuning of the keep-alive parameter of web servers based on request intervals

杉木 章義 † 河野 健二 †† 岩崎 英哉 †††

Akiyoshi SUGIKI Kenji KONO Hideya IWASAKI

† 電気通信大学大学院電気通信学研究科情報工学専攻

Course in Computer Science and Information Mathematics,

Graduate School of Electro-Communications, The University of Electro-Communications

sugiki@zeus.cs.uec.ac.jp

†† 慶應義塾大学理工学部情報工学科

Department of Information and Computer Science, Keio University

kono@ics.keio.ac.jp

††† 電気通信大学情報工学科

Department of Computer Science, The University of Electro-Communications

iwasaki@cs.uec.ac.jp

インターネットサーバの手動による性能パラメータ調整は，多くの経験や時間を必要とし，管理コストの増大を招くことが知られている．ウェブサーバの主要な性能パラメータである keep-alive 時間は，適切に設定しない場合，サーバのスループットや応答性を低下させることがある．本論文では，ウェブサーバの keep-alive 時間の自動設定機構を提案する．本機構は管理者の介入を必要とせず，手動設定で求めた値に近い keep-alive 時間に自動設定する．本機構はリクエスト間隔を監視しながら，データを送受信していない接続を切断し，データを頻繁に送受信している多くのクライアントからの接続を保つように keep-alive 時間を設定する．本機構を Apache ウェブサーバを対象としたライブラリとして実装し，実験を行った．その結果，異なる 2 つの負荷に対して，それぞれ keep-alive 時間を自動的に設定し，サーバの性能を適切に維持することを確認した．

1 はじめに

インターネットサーバにおいて目標とする性能を得るためには，サーバの性能パラメータを適切に設定することが必要である．ハードウェアやソフトウェアが適切であっても，性能パラメータの値が適切でなければ大幅な性能低下を招く．

現状の性能パラメータ設定は管理者によって試行錯誤を繰り返しながら行われている．適切なパラメータ値を得るためには，負荷を予測し，予測に基づいた負荷をサーバに与え，何度もテストを繰り返す必要がある．これは多くの時間を要するのと同時に，予測した負荷が実際の負荷と異なっていた場合に，目標とする性能が得られないことがある．また，このようなテストを行うことなく管理者の経験に基づいてパラメータを設定することも多い．この場合，サーバの環境に合わせて性能パラメータ値を求めているわけでないため，環境によっては期待した性能が得られない場合がある．手動による性能パラメータ設定は多大な経験や時間を必要とし，管理コストの増大を招くことが知られている [1]，また，設定誤りも招きやすく，サーバの主要な障害の原因となっている [2, 3, 4]．以上から，自動的なパラメータ設定を可能とする技術が求められている [5]．

本論文では，従来は手動により設定されていたウェブサーバの keep-alive 時間を自動設定する機構を提案する．本機構は管理者の介入を必要とせず，サーバの環境に応じて，試行錯誤を繰り返して求めた値に近い keep-alive 時間に自動設定する．サーバの稼働中に，クライアントからの負荷を監視しながら設定するため，事前に負荷を予測する必要もテストを繰り返す必要もない．本機構はリクエスト間隔を監

表 1: keep-alive 時間の違いによる性能の変化

keep-alive 時間	400 ミリ秒	1 秒	2 秒	15 秒 (デフォルト)
スループット [MBytes/sec]	18.79	17.29	11.68	7.51
平均応答時間 [ms]	880.9	955.9	882.3	1016.4

視しながら，データを送受信していない接続を切断し，データを頻繁に送受信している多くのクライアントからの接続を保つように keep-alive 時間を設定する．

本機構は既存のサーバソフトウェアやオペレーティングシステム (OS) を変更することなく利用可能である．本機構は Linux 上で動作する Apache ウェブサーバ [6] 用の外部ライブラリとして実装しており，ライブラリを環境変数に指定して Apache を再起動するだけでよい．このライブラリは実行時に Apache と Linux 間に介在し，Apache がシステムコールを呼び出すたびに，その引数を書き換え，keep-alive 時間の設定を行う．

本機構を用いて実験を行ったところ，2 つの異なる負荷に対して，それぞれ keep-alive 時間を適切に自動設定することができた．標準的なファイル分布を想定した負荷では，400 ミリ秒に手動設定することで，良好なスループットを維持することができる．本機構では，これに近い 450 ミリ秒に設定することができた．画像の多いサイトを想定した負荷では，800 ミリ秒に手動設定することで応答時間の増加を抑えることができる．本機構により，これに近い 850 ミリ秒に設定することができた．

以下，2 章では，keep-alive 時間の分析を行う．3 章では，keep-alive 時間の自動設定機構を示す．4 章で実装について示し，5 章では実験とその結果を示す．6 章で関連研究について述べ，7 章でまとめを示す．

2 ウェブサーバの keep-alive 時間

2.1 keep-alive 時間が性能に与える影響

クライアントとウェブサーバ間では，HTTP の規約 [7] により，ひとつの TCP 接続で複数の要求の送信やファイルの受信を行うことができる．これは **keep-alive 接続**と呼ばれ，ネットワークの使用率や応答時間の向上に効果的であることが知られている [8]．

keep-alive 時間とは，クライアントとサーバ間の接続のタイムアウトを決めるパラメータである．keep-alive 接続中にデータを送受信しない期間が keep-alive 時間を超えると，接続は切断される．

大規模なウェブサイトでは，経験的に keep-alive 時間を 1 秒や 2 秒に設定するのがよいとされている．しかし，keep-alive 時間の設定をより緻密に行うとさらに高い性能が得られることがある．

keep-alive 時間による性能への影響を確認するため，Apache ウェブサーバに対し，クライアントの振る舞いをエミュレートするベンチマークを用いて実験を行った（実験環境の詳細は 5.1 節を参照）．表 1 に，さまざまな keep-alive 時間でのスループットと平均応答時間の関係を示す．これはサーバに接続を試みるクライアント数を 1200 とした場合の結果である．

表 1 を見ると，keep-alive 時間はサーバの性能に大きな影響を与えている．keep-alive 時間を 1 秒，2 秒にすると，デフォルトの 15 秒に比べ，スループットや応答時間が向上している．keep-alive 時間を 400 ミリ秒に設定すると，1，2 秒の場合に比べスループットがさらに，それぞれ 27.9%，60.9%向上している．応答時間もそれぞれ 7.8%，0.2%向上している．以上から，keep-alive 時間の適切な設定が必要である．

2.2 keep-alive 時間が性能に影響を与える要因

keep-alive 時間がウェブサーバの性能に影響を与える要因は，データを送受信しない接続の維持を続けると TCP 接続の使用率が低下するためである．我々の実験でも，Internet Explorer，Firefox，Opera などの主要なウェブブラウザでは応答性を向上するため，サーバが許可するかぎり接続を維持することが確認されており，データを送受信しない場合にも接続は維持されたままとなる．

図 1 に示すように，クライアントは接続中，データの送受信を常に行っているわけではなく，データを送受信する期間とデータを送受信しない期間がある [9, 10]．データを頻繁に送受信する期間は，同一ウェブページを構成する画像や動画などのファイルを続けて取得する期間である．一般にウェブページは複数のファイルで構成されるため，リクエストの

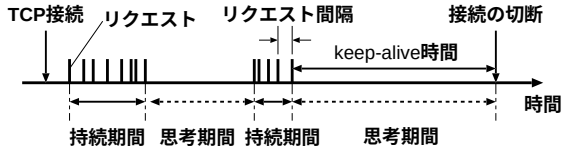

図 1: クライアントからのリクエストの流れ

送信が連続する．この期間を**持続期間**と呼ぶ．一方，接続を使用しない期間は，ウェブブラウザを操作する人が次のページへのリンクをクリックするまでの期間であり，**思考期間**と呼ぶ．

サーバがリクエストを受信する間隔を**リクエスト間隔**と呼ぶ（図 1 参照）．リクエスト間隔が keep-alive 時間を超えると，接続は切断される．そのため，keep-alive 時間が大き過ぎれば，思考期間内の接続も維持されたままとなり，サーバに対する TCP 接続の使用率が低下する．逆に，keep-alive 時間が小さ過ぎれば，持続期間内に接続が切断され，スループットと応答性が低下する．keep-alive 時間を適切に設定すれば，持続期間中は接続を維持し，思考期間に入ると接続を切断することができ，サーバの性能を適切に維持することができる．

3 keep-alive 時間の自動設定機構

本章では，リクエスト間隔を用いた keep-alive 時間の自動設定機構を提案する．まず，3.1 節でリクエスト間隔の分析を行う．3.2 節でリクエスト間隔をもとにした自動設定機構を示す．

3.1 リクエスト間隔の分析

本機構は keep-alive 時間を適切に設定することで，持続期間中は TCP 接続を維持し，思考期間に入った場合，直ちに接続を切断することを目指す．リクエスト間隔を分析し，リクエスト間隔がどの程度大きくなったら切断するかを考える．そのため実際のリクエスト間隔のヒストグラムを作成して調査した．

図 2 は著者らの研究室のウェブサーバで，1 週間リクエスト間隔を測定した結果である．6868 個のリクエストがあり，その間隔のヒストグラムを示したのが図 2 である．学外からのアクセスが全体の 91.2%であった．

図 2 を見ると，リクエスト間隔が 500 ミリ秒以内にリクエストの大きな分布があることが分かる．持続期間内のリクエスト間隔は短く，全リクエストの

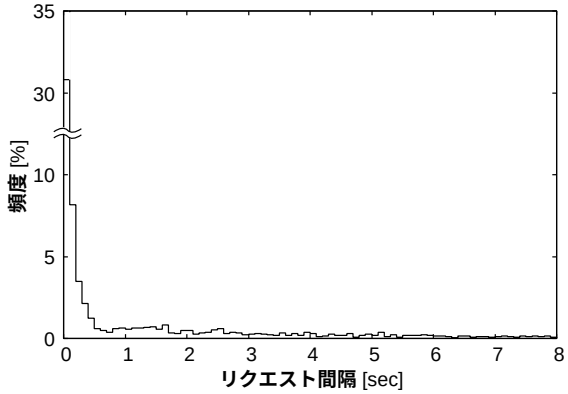

図 2: 実運用ウェブサーバでのリクエスト間隔の分布

大部分を占めるためである．リクエスト間隔が 500 ミリ秒以降，リクエスト間隔の頻度が大きく落ち込んだ後，一様な分布が見られる．これは思考期間内のリクエスト間隔である．このような分布が得られるのは，人間の思考時間が入るためリクエスト間隔が大きく，ユーザがリンク先をクリックするまでの時間が大きく異なるためである．

短いリクエスト間隔に全リクエストの大部分が集中するという図 2 の概形は一般的に現れる．本機構では，リクエスト間隔の頻度が急激に減少する場所を見つけ，そこを keep-alive 時間として設定する．この時間を**最大持続間隔**と呼ぶ．図 2 では，最大持続間隔は 500 ミリ秒である．こうすることで，持続期間内の多くのクライアントの接続は維持され，思考期間に入った場合に接続を切断できる．

思考期間に比べ，クライアント間の RTT（round-trip time）によるリクエスト間隔の揺らぎは小さい．これを確認するために実際に測定した結果が図 3 である．ここではウェブサーバに対して 100KB のファイル 10 個を keep-alive 接続を利用して 100 回取得した結果である．横軸は左から順にホップ数 2 の研究室内の場合，ホップ数 15，19 の他大学から取得した場合，最後はホップ数 17 の家庭用の光ファイバー回線から取得した場合であることを示している．縦軸はリクエスト間隔の平均と 95%信頼区間を示している．どの場合も，ほとんどのリクエスト間隔が 100 ミリ秒以内に収まっている．これは文献 [11, 12] のほとんどの RTT は 500 ミリ秒以内に収まるという結果と一致する．

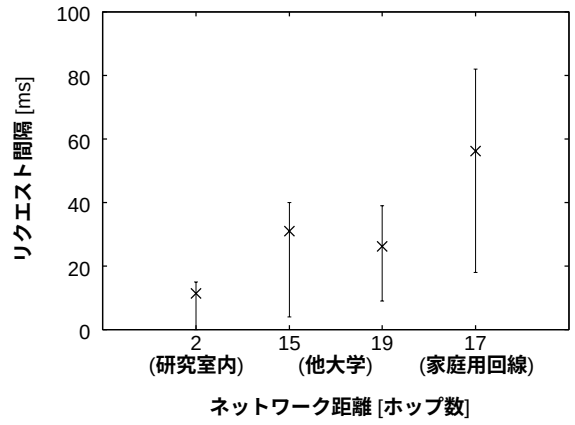

図 3: リクエスト間隔の RTT によるゆらぎ

3.2 設定方式

提案する keep-alive 時間設定機構では，3.1 節の手法を実現する．本機構では，サーバ全体で単一の keep-alive 時間を設定する．サーバに接続するクライアント全体で一つのヒストグラムを作成し，keep-alive 時間の設定に使用する．

図 4 に設定方式全体の疑似コードを示す．関数 control() は，リクエスト間隔を測定する度に呼び出す．control() では，まず (1) 全てのリクエスト間隔を記録用配列 *arr* に記録し（8–11 行），(2) リクエスト間隔のヒストグラム *hist* を作成する（12–14 行）．*hist* は配列であり，リクエスト間隔 t を添字に与えると，そのリクエスト間隔での頻度を返す．(3) 作成したヒストグラムをもとに，最大持続間隔 t_{max} を見つけ（15–28 行），(4)keep-alive 時間 t_{ka} を更新する（29–30 行）．以上により，keep-alive 時間の設定を実現する．

なお，keep-alive 時間の設定は定期的に行うこともできる．これは，ウェブサーバに対する負荷の変化に応じて，適切な keep-alive 時間も変化するからである．

以下，疑似コードの詳細な説明を行う．簡単のため，最大持続間隔が現在の keep-alive 時間以下の場合（3.2.1 節），最大持続間隔が現在の keep-alive 時間を超える場合（3.2.2 節）に分けて説明する．最後に，3.2.3 節で keep-alive 時間の更新方法を説明する．

```
 1: function control();
 2: // arr は記録用配列（大きさ N）
 3: // index は arr の添字（初期値 0）
 4: // hist はヒストグラム用配列（大きさ T）
 5: // t_ka は現在の keep-alive 時間
 6: // t_max は最大持続間隔
 7: begin
 8:   // (1) リクエスト間隔の記録
 9:   arr[index] := 前回のリクエストからの経過時間;
10:   index := index + 1;
11:   if index < N then return;
12:   // (2) ヒストグラムの計算
13:   arr からヒストグラム hist を作成;
14:   index := 0;
15:   // (3) 最大持続間隔の計算 （3.2.1 節）
16:   for t := 0 to T − 1 do begin
17:     ratio := hist[t]/∑_{j≤t} hist[j]; // 式 (1)
18:     // 最大持続間隔 > keep-alive 時間（3.2.2 節）
19:     if t ≥ t_ka then begin
20:       err := (ratio − target)/target; // 式 (2)
21:       err に応じて最大持続間隔 t_max を大きく;
22:       break
23:     end;
24:     if ratio < target then begin
25:       t_max := t; // 最大持続間隔の更新
26:       break
27:     end
28:   end;
29:   // (4)keep-alive 時間の更新 （3.2.3 節）
30:   t_ka := α · t_ka + (1 − α) · t_max // 式 (3)
31: end.
```

図 4: 本調整機構の擬似コード

3.2.1 最大持続間隔の計算 (最大持続間隔 ≤keep-alive 時間)

最大持続間隔 t_{max} を求める過程を図 5 を用いて説明する．最大持続間隔となるリクエスト間隔を求めるため，リクエスト間隔 0 から順に大きい方へ，ヒストグラム *hist* を調べていく．そして，リクエスト間隔の頻度が十分小さくなった地点を最大持続間隔とする．

直感的に，リクエスト間隔 t の頻度 $hist[t]$ が，ある閾値を下回ったところを最大持続間隔とするのが自然である．しかし，その判定法では，頻度が比較的小さいリクエスト間隔の山の立上りに，最大持続間隔が誤って設定されることがある．

そのため，式 (1) の比率 *ratio* の値が閾値 *target* より小さくなった地点を最大持続間隔とする．比率 *ratio* は，新しく調べるリクエスト間隔 t の頻度 $hist[t]$ が，これまでの累積の頻度 $\sum_{j\leq t} hist[j]$ に占める割合である（図 4 の 17 行目）．

$$ratio := hist[t]/\sum_{j\leq t} hist[j] \qquad (1)$$

比率 *ratio* を用いて判定すると，山の立上りではこ

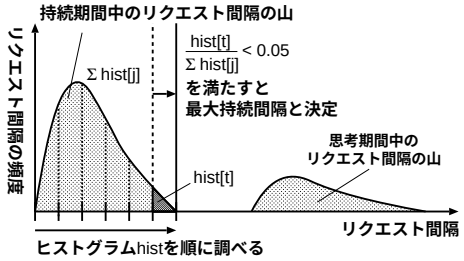

図 5: 最大持続間隔の計算

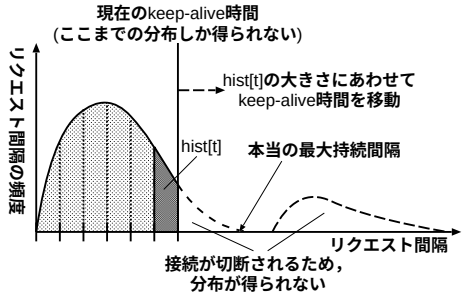

図 6: 最大持続間隔の計算 (keep-alive 時間より大きい場合)

の比率が相対的に大きくなるため，最大持続間隔が誤って設定されるのを避けることができる．判定のための閾値 *target* は現在は 0.05 としており，比較的良好な結果が得られている．また，*target* は負荷によらない値であるため容易に決めることができる．$hist[i]$ が最大持続間隔を含むならば，*ratio* は持続期間内のリクエスト間隔全体に $hist[i]$ が占める比率となるからである．

3.2.2 最大持続間隔の計算 (最大持続間隔 >keep-alive 時間)

最大持続間隔が現在の keep-alive 時間より大きい場合，図 6 に示す通り，リクエスト間隔が keep-alive 時間を超える接続は切断されるため，keep-alive 時間を超えるリクエスト間隔の分布が得られない．

本機構では，keep-alive 時間の直前のリクエスト間隔 t の比率 *ratio* と，最大持続間隔を判定する閾値 *target* とのずれに応じて最大持続間隔 t_{max} を現在の keep-alive 時間より大きくする．このずれは式 (2) により計算する（図 4 の 20 行目）．

$$err := (ratio - target)/target \tag{2}$$

keep-alive 時間直前のリクエスト間隔 t に，誤差 *err* に比例した値を加えたものを最大持続間隔 t_{max} とすることで，本当の最大持続間隔に近づける．なお，最大持続間隔を大きく調整し過ぎた場合でも，3.2.1 節の設定方式により修正されるため問題はない．

3.2.3 keep-alive 時間の更新

最後に，3.2.2 節までの設定方式により計算した最大持続間隔をもとに，keep-alive 時間を更新する．

本機構では，リクエストのバーストを考慮した keep-alive 時間の更新を行う．インターネット・サーバでは，頻繁にリクエストのバーストが観測される [13, 14, 15]．一時的なバーストによって，リクエスト間隔が大きく変化し，最大持続間隔が誤った値に設定される可能性がある．そのため本機構では，3.2.2 節までの方式で求めた最大持続間隔をただちに keep-alive 時間とするのではなく，過去の keep-alive 時間を考慮して keep-alive 時間を更新する．過去の keep-alive 時間を含めることで，一時的なバーストにより最大持続間隔が大きく変化しても，その影響を抑えることができる．実現には指数移動平均法による式 (3) を用いる（図 4 の 30 行目）．

$$t_{ka} := \alpha \cdot t_{ka} + (1 - \alpha) \cdot t_{max} \tag{3}$$

t_{max} は図 4 の擬似コードにより計算した最大持続間隔である．t_{ka} は式 (3) で計算した一つ前の時刻の keep-alive 時間であり，これを用いて新しい keep-alive 時間を計算する．α は重みであり，現在は $\alpha = 0.9$ としている．

ウェブサーバの起動時には，keep-alive 時間は任意の値（例えばデフォルト値）でよい．その後，式 (3) により徐々に適切な keep-alive 時間へと近付けていく．

4 実装

3 章で述べた手法を，Linux 上で動作する Apache 2.0.49 に実装を行った．

Apache ではクライアントからの接続ごとにプロセスを割り当て，各プロセスは I/O 入出力を監視するシステムコール `poll()` により，クライアントからのリクエストの送信を監視する．クライアントからリクエストが送信されると，その処理を行い，再び `poll()` により次のリクエストを待つ．keep-alive 時間は `poll()` の引数であるタイムアウト時間で指定されており，`poll()` がタイムアウトすると，接続を

表 2: 実験で用いる負荷

負荷	説明	ファイル・サイズ分布 ≤1KB	≤10KB	≤100KB	≤1MB	総ファイル・サイズ
(a) SPECWeb 標準	SPECWeb99 標準の負荷	35%	50%	14%	1%	3.6GB
(b) ファイル大	SPECWeb99 標準を平均ファイル・サイズが大きくなるように修正したもの	1%	14%	50%	35%	3.6GB

切断する．

本機構に必要な keep-alive 時間の設定は，この `poll()` の引数を書き換えることで実現できる．タイムアウト時間を 3 章で述べた手法で計算した値に設定することで，keep-alive 時間を調整する．keep-alive 時間の計算に必要なリクエスト間隔の測定は，`poll()` の前後の時間差を測定することで実現する．測定には `gettimeofday()` を用いているが，50 ミリ秒の単位で keep-alive 時間の設定を行うため，タイマの精度は問題とならない．

本機構は環境変数 `LD_PRELOAD` の機能を利用し，実行時に OS とサーバ間に挿入するライブラリとして実装されている．ユーザが作成したライブラリを環境変数 `LD_PRELOAD` に指定しておくと，OS やアプリケーションを改変することなしに，標準ライブラリ関数（とそれに対応するシステムコール）への呼出しをフックし，置き換えることができる．本機構では `poll()` の呼出しをフックし，タイムアウト時間の書き換えを行っている．そのため，本実装は Apache と Linux を変更することなしに，利用可能である．

5 実験

本機構が，試行錯誤を繰り返して求めた値に近い keep-alive 時間に適切に設定することを示すため実験を行った．異なる 2 つの負荷を与えて，本機構がそれぞれ適切に keep-alive 時間を設定し，サーバの性能を維持することを確認した．

5.1 実験環境

サーバ計算機は，CPU が Pentium4 2.8GHz，主記憶 512MB，SCSI 接続，7200 回転の HDD，33MHz，32 ビット PCI バスの PC を用いた．クライアント計算機は，サーバ計算機と同じ構成のものを 16 台用いた．サーバ計算機とクライアント計算機は，1000 Base-T で 1 台のスイッチに接続されている．

ウェブサーバは Apache 2.0.49 を用いた．Apache のコンパイルではデフォルトの設定であるプロセスのみを使用する `prefork MPM` を用いた．最大同時接続数を決める `MaxClients` パラメータの値は 512 とした．他の全てのパラメータは Apache の設定ファイルのデフォルト値を用いた．OS は Linux 2.4.20 を用いており，カーネルのパラメータは全て変更していない．

5.2 実験に用いる負荷

負荷生成は SPECWeb99[16] に思考時間を導入したものを用いた．SPECWeb99 はウェブサーバを対象とした標準的なベンチマークである．しかし，サーバの性能測定を目的とするため，思考期間に相当するクライアントの振舞いは考慮されていない．思考期間中のリクエスト間隔を模倣するため，Pareto 分布にしたがってリクエストを生成するようにした．一般に，思考期間は Pareto 関数で近似するのがよいことが知られている [9]．Pareto 関数の係数は，文献 [9] の値を用いた．なお，持続期間内のリクエスト間隔は SPECWeb99 のものを用いている．このベンチマークのリクエスト間隔の分布は図 7 のようになっており，実測により求めた図 2 とほぼ同様の分布が得られている．今回，動的ページに対する要求は行わず，静的ページに対する要求のみとする．

実験では，表 2 の 2 つの負荷を用いる．最初の **(a) SPECWeb 標準**は，SPECWeb99 の規約で定められている標準設定である．もう一つの **(b) ファイル大**は，画像を多く含むサイトなどを想定した平均ファイル・サイズの大きい負荷である．

5.3 実験結果: (a) SPECWeb 標準

図 7 にリクエスト間隔のヒストグラムと，本機構が設定した keep-alive 時間を示す．持続期間内の山は 300 ミリ秒を中心に分散している．本機構では 450 ミリ秒を keep-alive 時間として設定した．

図 8 に keep-alive 時間調整の時間的変化を示す．自動調整機構はデフォルト値である 15 秒から，最終的に 450 ミリ秒に落ち着いた．制御回数 50，80，130

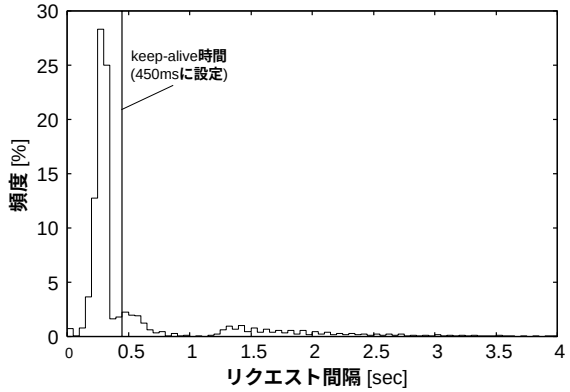

図 7: リクエスト間隔の分布 (SPECWeb 標準)

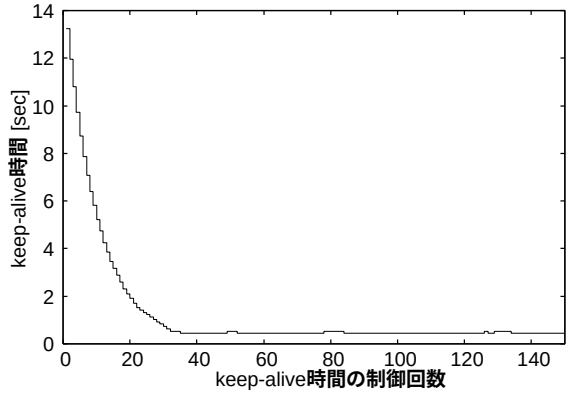

図 8: keep-alive 時間の時間的変化 (SPECWeb 標準)

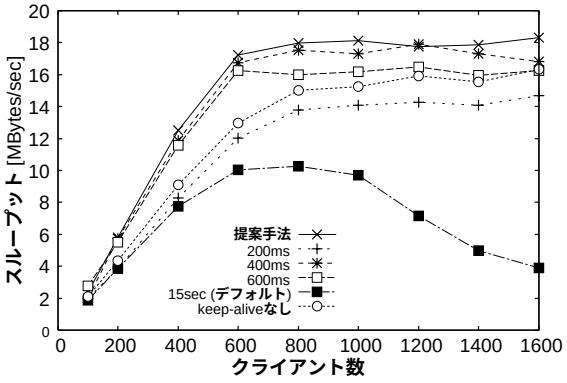

図 9: スループット (SPECWeb 標準)

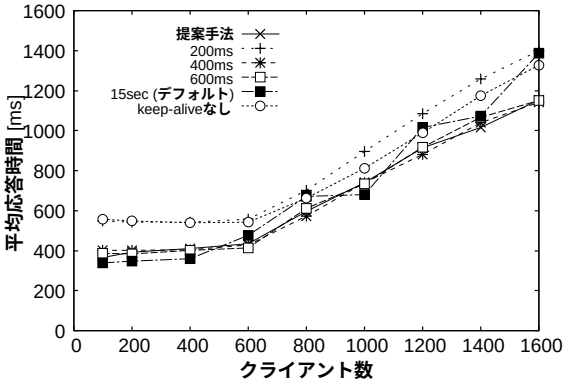

図 10: 平均応答時間 (SPECWeb 標準)

回のあたりで 3.2.2 節の調整方式により，keep-alive 時間の修正が行なわれている．

収束の速さは式 (3) の α の値（3.2.3 節）により調節可能である．実験では，$\alpha = 0.9$ としている．この値は，最終的に設定される keep-alive 時間に影響を与えないが，負荷の変化に対する俊敏さを決める．

keep-alive 時間を変化させたときの，スループットと平均応答時間の結果をそれぞれ図 9，図 10 に示す．手動設定の場合には 400 ミリ秒が一番よい．スループットが最大であり，平均応答時間は他と変わらない値を示しているからである．本機構では，管理者の介入を必要とせず，keep-alive 時間を 400 ミリ秒に近い 450 ミリ秒に設定している．サーバのデフォルトの値である 15 秒と比較すると，平均応答時間は変らず，スループットは 27.5–368.5%向上となっている．

その他の結果を見ると，デフォルトの 15 秒ではスループットがあまり伸びず，クライアント数が 1000 以降，接続の失敗のためにスループットが低下している．平均応答時間も同様に乱れているが，クライアント数が `MaxClients` の値である 512 より小さい場合には，思考期間による接続の使用率低下の影響を受けないため，最もよい平均応答時間を示している．200 ミリ秒では，keep-alive 時間が短すぎ，持続期間内の接続も切断されている．よって，スループットが低下し，平均応答時間も増加している．600 ミリ秒では，keep-alive 時間が大きくなり始め，思考期間による TCP 接続の使用率低下の影響を受け始める．よって，スループットが低下している．keep-alive 接続なしの場合には，リクエストごとに個別に再接続するため，平均応答時間が増加している．スループットも同様に低い値を示している．

5.4 実験結果: (b) ファイル大

図 11 にリクエスト間隔の分布と本機構が設定した keep-alive 時間を示す．図 7 の SPECWeb 標準の場合と異なり，持続期間内の山が広がっている．これは，ファイル転送時間の増加とともに，リクエスト間隔が大きくなるためである．本機構では，山の広

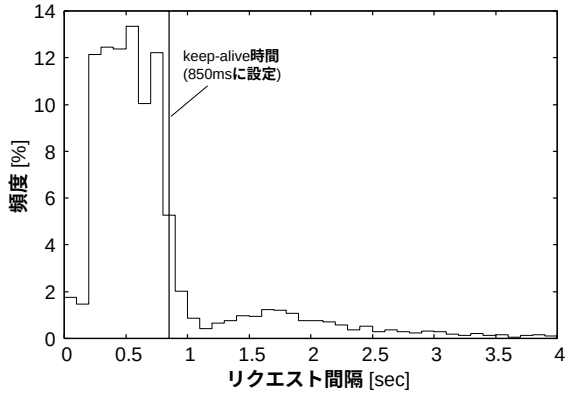

図 11: リクエスト間隔の時間分布 (ファイル大)

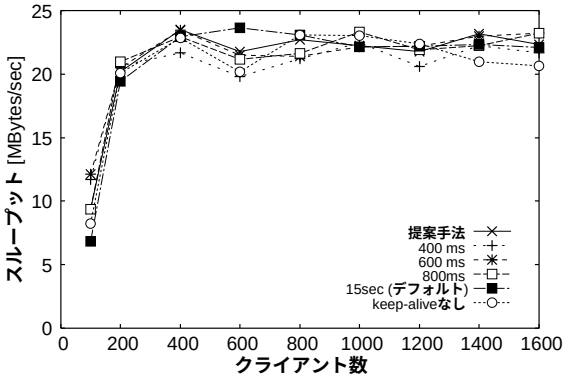

図 12: スループット (ファイル大)

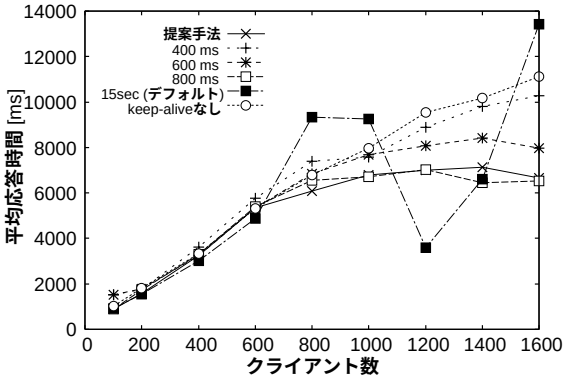

図 13: 応答時間 (ファイル大)

がりに応じて，keep-alive 時間を 850 ミリ秒に設定した．

keep-alive 時間を変化させた場合のスループットを図 12，平均応答時間を図 13 に示す．図 13 の平均応答時間では，keep-alive 時間設定の効果が現れている．手動で設定した場合，keep-alive 時間が大きいほど平均応答時間がよく，800 ミリ秒の場合に最もよい．本機構では 800 ミリ秒に近い 850 ミリ秒に自動的に設定し，平均応答時間の増加を抑えている．しかし，keep-alive 時間を大きく設定しているデフォルトの 15 秒では，クライアント数 800 以降，接続失敗のため応答時間のばらつきが大きい．

一方，スループットは，keep-alive 時間を変化させてもあまり変化しない．ファイル転送に多くの時間を要し，接続にかかる時間の影響が相対的に小さく，keep-alive 時間調整の効果が打ち消されているためである．そのため，keep-alive なしの場合もほぼ同様のスループットを示している．測定のばらつきが大きいのは，平均ファイル・サイズの増加にともなって，要求するファイルごとに必要な処理時間が大きく異なるためである．

6 関連研究

6.1 keep-alive 接続の制御

Barford ら [17] は，同一ウェブページを構成する画像や動画などをすべて取得するたびに接続を切断することを推奨しており，文献中ではこれを *early close* と呼んでいる．しかし，*early close* をどう実現するかについては論じられていない．本論文で提案する手法は，この *early close* を自動的に行う機構と見ることもできる．

同時接続数がサーバの最大同時接続数に達した場合，切断すべき接続を選択するアルゴリズムとして LRU (Least Recently Used) を用いることを Mogul[18] は提案している．しかし，LRU 方式では持続期間と思考期間を区別していないため，接続要求が次々とサーバに到着した場合，持続期間中の接続でも切断される可能性がある．これを防ぐため，LRU 方式で切断する対象を一定時間以上データを送受信しない接続をとすることが考えられる．この閾値を求めるために，本論文の自動設定機構を用いることができる．

6.2 汎用的なパラメータ設定

Active Harmony [19] では，クラスタ上で動作するウェブサーバを対象にパラメータの自動設定を行っている．パラメータ設定を数学的な最適化問題に帰着し，シンプレックス法により最適解を求めている．汎用性が高い反面，発見的手法であるため，与える初期解により，極小解に陥ったり，解が得られるまで時間がかかることがある．また，Active Harmony

では keep-alive 時間の設定は行われていない．

サーバの構造を単純化した数学的モデルを作成し，制御理論によりパラメータ設定を行っているものに文献 [10, 20, 21] がある．しかし，現在のウェブサーバの構造は複雑化しており，モデルの妥当性を検証することが難しい．Diao ら [10] は，Apache を対象に CPU とメモリで構成されるモデルを作成し，最大同時接続数と keep-alive 時間の 2 つのパラメータを設定している．CPU とメモリの使用量がパラメータの線形結合で求められると仮定し，人工的な負荷をサーバに与え，モデルに現れる係数を求めている．CPU とメモリの使用量は負荷に依存して決まると考えられ，負荷が変わるとどの程度正しく動作するかについては議論されていない．Doyle ら [20] はメモリとディスクの 2 つの資源をモデル化し，共有ホスティングを行なうウェブサーバを対象としている．メモリとディスクのみを対象としているため，現実のシステムとの対応を見極め，導入する必要がある．Abdelzaher ら [21] は Apache を対象に，古典的な制御理論の手法を用いて，アドミッション・コントロールを行っている．サーバの環境ごとにモデルの各係数を求める必要があり，クライアントからの負荷が大きく変わるたびに係数を求め直さなければいけない可能性がある．

6.3 ウェブサーバの性能向上

ウェブサーバの性能を向上させる研究は，数多く行われている．サーバの最大同時接続数を向上させる試みとして Flash[22]，SEDA[23]，Capriccio[24]，Zeus[25] など，新しいサーバ構成法が研究されている．接続数に対するスケーラビリティは向上したものの，接続ごとの状態管理や入出力の監視など，接続を維持するためのコストは新しいサーバ構成法でも問題となると考えられる．本論文の提案方式は新しいサーバ構成法とも組み合わせて利用することができる．

また，既存の Apache の性能向上を対象とした研究として，Hu ら [26] は時間のかかるシステムコールの結果をキャッシュすることによる高速化を行っている．Nahum ら [27] は，ネットワーク I/O 関数の変更による高速化や送信バイト列のチェックサムのキャッシュなど TCP 階層での最適化を行っている．これらは低階層での最適化を対象としており，本研究の keep-alive 時間の自動設定機構と組み合わせて用いることができる．

7 まとめ

本論文では，ウェブサーバの keep-alive 時間の自動設定法を提案し，実装方法を示した．提案手法は管理者の介入を必要とせず，サーバの環境に応じて，手動設定の最適な値に近い keep-alive 時間に自動設定する．本機構はリクエスト間隔を見ながら keep-alive 時間を設定する．また，提案手法は既存のサーバを変更せず，利用可能である．

2 つの異なる負荷に対して実験を行なったところ，それぞれ keep-alive 時間を適切に設定し，ウェブサーバの性能を維持することができた．

参考文献

[1] Xi, B., Liu, Z., Raghavachari, M., Xia, C. H. and Zhang, L.: A Smart Hill-Climbing Algorithm for Application Server Configuration, *WWW Conf. 2004* (2004).

[2] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J. and Treuhaft, N.: Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies, *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175* (2002).

[3] Oppenheimer, D., Ganapathi, A. and Patterson, D. A.: Why do Internet services fail, and what can be done about it?, *USENIX Symp. on Internet Technologies and Systems* (2003).

[4] Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R. P. and Nguyen, T. D.: Understanding and Dealing with Operator Mistakes in Internet Services, *6th USENIX Symp. on Operating Systems Design and Implementation* (2004).

[5] Ganek, A. G. and Corbi, T. A.: The Dawning of the Autonomic Computing Era, *IBM Systems Journal*, Vol. 42, No. 1, pp. 5–18 (2003).

[6] The Apache Software Foundation: Apache HTTP Server (1995). `http://www.apache.org/`.

[7] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 (1999).

[8] Nielsen, H. F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. W. and Lilley, C.: Network Performance Effects of HTTP/1.1, CSS1, and PNG, *ACM SIGCOMM'97* (1997).

[9] Barford, P. and Crovella, M.: Generating Representative Web Workloads for Network and Server Performance Evaluation, *ACM SIGMETRICS'98*, pp. 151–160 (1998).

[10] Diao, Y., Gandhi, N., Hellerstein, J., Parekh, S. and Tilbury, D.: Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics

With Application to the Apache Web server, *7th IEEE/IFIP Symp. on Integrated Network Management* (2001).

[11] Allman, M.: A Web Server's View of the Transport Layer, *ACM SIGCOMM Computer Communication Review*, Vol. 30, No. 5, pp. 10–20 (2000).

[12] Jiang, H. and Dovrolis, C.: Passive Esitimation of TCP Round-Trip Times, *ACM SIGCOMM Computer Communication Review*, Vol. 32, No. 3, pp. 75–88 (2002).

[13] Jung, J., Krishnamurthy, B. and Rabinovich, M.: Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites, *WWW Conf. 2002* (2002).

[14] Fox, A., Gribble, S., Chawathe, Y., Brewer, E. and Gauthier, P.: Cluster-Based Scalable Network Services, *16th ACM Symp. on Operating System Principles*, pp. 78–91 (1997).

[15] Welsh, M. and Culler, D.: Adaptive Overload Control for Busy Intenet Servers, *4th USENIX Symp. on Internet Technologies and Systems* (2003).

[16] Standard Performance Evaluation Corporation: The SPECweb99 benchmark (1999). `http://www.spec.org/osg/web99/`.

[17] Barford, P. and Crovella, M.: A Performance Evaluation of Hyper Text Transfer Protocols, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27, No. 1, pp. 188–197 (1999).

[18] Mogul, J. C.: The Case for Persistent-Connection HTTP, *ACM SIGCOMM'95* (1995).

[19] Chung, I.-H. and Hollingsworth, J. K.: Automated Cluster-Based Web Service Performance Tuning, *13th IEEE Int'l Symp. on High Performance Distributed Computing* (2004).

[20] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W. and Vahdat, A. M.: Model-Based Resource Provisioning in a Web Service Utility, *4th USENIX Symp. on Internet Technologies and Systems* (2003).

[21] Abdelzaher, T. F., Shin, K. G. and Bhatti, N.: Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 13, No. 1, pp. 80–96 (2002).

[22] Pai, V. S., Druschel, P. and Zwaenepoel, W.: Flash: An Efficient and Portable Web Server, *1999 USENIX Annual Tech. Conf.* (1999).

[23] Welsh, M., Culler, D. and Brewer, E.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, *Proc. 18th ACM Symp. on Operating Systems Principles*, pp. 230–243 (2001).

[24] Behren, R., Condit, J., Zhou, F., Necula, G. C. and Brewer, E.: Capriccio: Scalable Threads for Internet Services, *19th ACM Symp. on Operating Systems Principles* (2003).

[25] Zeus Technologies: Zeus Web Server. `http://www.zeus.com/`.

[26] Hu, Y., Nanda, A. and Yang, Q.: Measurement, Analysis and Performance Improvement of the Apache Web Server, *Proc. 18th IEEE Int'l Performance, Computing and Communications Conference* (1999).

[27] Nahum, E., Barzilai, T. and Kandlur, D. D.: Performance Issues in WWW Servers, *IEEE/ACM Transations on Networking*, Vol. 10, No. 1, pp. 2–11 (2002).