コネクション集合における通信の局所性を考慮したベンチマークツールの試作

河合 栄治　山口 英

奈良先端科学技術大学院大学

概要

インターネットの利用が多様化し，コンピュータにおける通信機能に対する要求も多様化してきている．本研究では，多数のコネクションを並行処理するサーバシステムのアーキテクチャ設計にヒントを与えるベンチマークツールの試作を行った．本システムの特徴は通信の局所性を設定することができることであり，様々な通信機能要求に対するテストを行うことができる．

An Implementation of a Benchmark Tool Considering Communication Locality

Eiji Kawai　Suguru Yamaguchi

Nara Institute of Science and Technology

Abstract

Utilization of Internet technologies is now widespread among various kinds of systems, and performance requirements of Internet communications are diversified. In this study, we developed a benchmark tool that evaluates system architectures with a large number of concurrent connections. In the benchmark tool, communication locality is configurable, and it can give system developers helpful hints to implement high performance systems.

1　はじめに

インターネット技術の利用が多様化し，通信機能に対する要求も同じく多様化してきている．例えば，科学技術計算等で用いられる大規模計算環境は，インターネット技術で接続されたコンピュータクラスタを用いて構築されることが多くなってきている．また，センサーネットワークなど，従来とは異なるインターネットの利用のされ方も出現してきている．

こうしたサービスに求められる高性能な通信機能を実現するために，これまでに様々なネットワークI/O管理機構が開発されてきている．このネットワークI/O管理機構とは，複数の通信を効率よく管理する多重化処理の仕組みのことである．

一方で，ネットワークサーバを実装したり，クラスタコンピューティング用の通信ライブラリを実装したりする際には，どの技術をどのように適用するのが最適なのか不明であることが多い．そこで本研究では，様々なネットワークI/O管理機構の性能評価を可能にするベンチマークツールの開発を行った．

2　ネットワークI/O管理機構

まず，代表的なネットワークI/O管理機構について概観し，利点と欠点について整理する．

2.1　マルチプロセス

マルチプロセスとは，各記述子に対してプロセスを割り当てる方式である．広く利用されているApache

Web サーバでは，この形式が採用されている．

マルチプロセスの利点は，プログラミングモデルが単純であり実装が容易であることである．各プロセスは，単一の記述子のみを管理すれば良い．一方で，OS 内のプロセス管理データ領域やプロセス切り替え処理といったプロセス管理のオーバヘッドが大きいという欠点がマルチプロセスにはある[*1]．

2.2 マルチスレッド

マルチスレッドとは，各記述子に対してスレッドを割り当てる方式であり，先に述べたマルチプロセスに似ている．

スレッドは，軽量プロセスと呼ばれることもあるように，その管理オーバヘッドが小さい．そのため，一般的にはマルチプロセスよりもマルチスレッドの方がスケーラビリティに優れているとされている．

2.3 ポーリング I/O

ポーリング I/O とは，複数の記述子の状態を取得（ポーリング）し，即時完了可能な I/O のみを記述子集合の中から選択的に実行する方式である．実装としては，`select()` や `poll()` が代表的である．

ポーリング I/O 方式は，単一の実行体（プロセスやスレッド）で多数の記述子を扱うことができるため，効率的であるという利点がある．一方で，記述子集合の走査が必要であり，記述子の数が巨大になるとその処理コストが大きくなるという問題がある．

2.4 明示的イベント通知機構

明示的なイベント通知機構 [1] とは，受信データの到着といった各記述子上で発生したイベントをサービスプロセスに何らかの手段によって通知する方式である．標準化された実装例には POSIX 実時間シグナル [2] があるが，ネットワーク I/O においては適用に問題が多いため，各プラットホームで独自に実装されることが多い．Linux であれば epoll 機構 [3]，BSD 系であれば kqueue 機構 [4]，Solaris であれば poll デバイスファイル機構 [5] が実装されている．

明示的なイベント通知機構は，オーバヘッドが小さいため，記述子の数に対するスケーラビリティが高い．

3 ベンチマークテストにおける通信

次に，ベンチマークツールにおける通信特性について考察し，設定パラメータを定義する．

3.1 通信特性

3.1.1 通信の局所性

通信における局所性の考え方は，様々なサービス要求を考慮する上で重要である．本研究では，この通信の局所性を，記述子集合において実際に通信が発生する部分集合によって定義した．具体的には，記述子において通信が発生する可能性が高い場合にそれを**アクティブ**であると定義し，そのような記述子を**アクティブ記述子**と呼ぶ．つまり，記述子集合におけるアクティブ記述子の割合が高ければ通信の局所性は低く，逆にアクティブ記述子の割合が低ければ通信の局所性は高くなる．

3.1.2 トラヒック特性

本ベンチマークツールでは，個々の通信で交換されるデータのサイズや，送受信のレートなど，既存のベンチマークツールでも考慮されているトラヒック特性についても設定可能とした．これらのパラメータは，サービスの状況などから得られることが多い．例えば，Web サーバの場合，提供するコンテンツセットとそれらへのアクセス傾向から，データサイズの分布や通信の発生頻度が分かる．

3.2 ベンチマークテストパラメータ

本ベンチマークツールは，TCP 通信を対象とし，計測対象となるホスト（サーバ）と，1 台以上のホスト（クライアント）との間で通信を行うものとする．その通信特性として，次に挙げるパラメータを定義

[*1] あらかじめ一定数のプロセスを生成し，サービス完了時にはプロセスを終了せず再利用することで，プロセス生成・終了のオーバヘッドは回避することができる．

した.

- **記述子総数**：サーバにおいて確立される TCP コネクションの総数.
- **アクティブ率**：記述子集合におけるアクティブ記述子の割合.
- **要求データサイズ**：クライアントからサーバに対して送信されるデータのサイズ.
- **応答データサイズ**：サーバからクライアントに対して送信されるデータのサイズ.
- **クライアントデータ送信レート**：クライアントからサーバに対してデータが送信される 1 秒あたりの回数.
- **ネットワーク I/O 管理機構**：2 章で述べた I/O 多重化の仕組み.

4 ベンチマークツールの設計

開発したベンチマークツールは，クライアントとサーバで構成される．本節では，これらの設計について述べる.

4.1 クライアント

クライアントは，設定されたパラメータ値に従ってサーバに対して擬似的な要求（ランダムなバイト列）を送信し，応答（同じくランダムなバイト列）を受信する．実装は，サーバに対して設定された数のコネクションを確立した後，要求を送る writer スレッドと応答を受信する reader スレッドの二つのスレッドに分かれるマルチスレッドプログラムとなっている．この 2 つのスレッドの基本動作は次の通りである.

writer スレッド

1. データ送信タイミングまで待機する.
2. データを送信する記述子をアクティブ記述子集合から選択する.
3. タイムスタンプを取得する.
4. データを当該記述子に書き込む.
5. 1 に戻る.

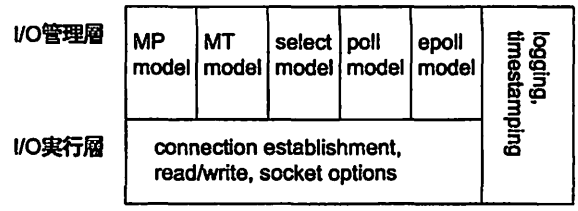

図 1 サーバプログラムのアーキテクチャ

reader スレッド

1. データ到着イベントを受信する.
2. 当該記述子からデータを読み込む.
3. タイムスタンプを取得する.
4. ログを記録する.
5. 1 に戻る.

現時点では，reader スレッドのイベント処理において Linux 独自の epoll 機構を用いて実装を行っているため，クライアントについてはバージョン 2.6 系の Linux 上でしか動作しない．またタイムスタンプには，x86 系プロセッサの RDTSC 命令で取得できるプロセッサクロックカウンタを用いている.

4.2 サーバ

サーバは，クライアントから送られた要求を受信すると，即時に応答を送信する．本サーバプログラムでは，ネットワーク I/O 管理機構を図 1 に示すように I/O 管理層として実装し，各実装モデルを切り替え可能にしている.

5 評価実験

実験では 5 台のクライアントホストと 1 台のサーバホストがギガビットイーサネットで接続される環境を用いた．なお，各ホストのシステム仕様は表 1 に示したとおりである．この実験環境において，各クライアントは要求データの送信時と応答データの受信時にタイムスタンプをログに記録し，それらの差であるサービス遅延時間を計測した．なお，それぞれの実験では 3 分間ログを記録している.

実験で用いたパラメータは，表 2 に示したとおりである．今回の実験では，WWW の通信モデルを例

表1　機器の仕様

サーバ	CPU: Pentium III 800MHz x 2 (SMP), MEM: 2GB, NIC: Intel Pro/1000T 82543GC (e1000)
	OS: Fedora Core 5 (Linux 2.6.15-1.2054_FC5)
クライアント	CPU: Pentium III 866MHz, MEM: 512MB, NIC: NetGear GA620T (acenic)
	OS: Fedora Core 5 (Linux 2.6.15-1.2054_FC5)

表2　実験パラメータ

パラメータ	値
記述子総数	400・2,000・10,000（各クライアントは 80・400・2,000）
アクティブ率	1/64 (1.5625 %)・1/16 (6.25 %)・1/4 (25 %)・1 (100 %)
要求データサイズ	128 byte
応答データサイズ	10 kbyte
クライアントデータ送信レート	125・500・2,000（各クライアントは 25・100・400）
ネットワーク I/O 管理機構	MP・MT・select・poll・epoll

表3　サービス遅延時間特性（アクティブ率 = 1，左: 平均値，中: 中央値，右: 標準偏差，単位: ミリ秒）.

記述子総数	レート	MP	MT	select	poll	epoll
400	125	0.565/0.563/0.126	0.563/0.560/0.121	1.13/1.13/0.146	1.14/1.14/0.141	0.557/0.555/0.121
400	500	0.574/0.568/0.141	0.557/0.553/0.138	1.18/1.13/0.232	1.22/1.17/0.245	0.569/0.564/0.138
400	2000	0.560/0.542/0.216	0.523/0.515/0.198	1.35/1.33/0.360	1.39/1.38/0.369	0.523/0.515/0.183
2000	125	0.568/0.565/0.120	0.574/0.569/0.124	5.05/4.78/1.14	5.39/5.18/1.32	0.558/0.557/0.118
2000	500	0.584/0.580/0.129	0.591/0.581/0.139	4.77/4.75/1.46	5.10/5.03/1.59	0.578/0.572/0.127
2000	2000	0.617/0.594/0.212	0.589/0.577/0.193	4.80/4.75/1.40	5.92/5.91/1.64	0.578/0.570/0.167
10000	125	0.588/0.583/0.127	0.563/0.562/0.115	20.3/19.1/7.14	22.1/21.9/7.61	0.572/0.563/0.128
10000	500	0.587/0.579/0.136	0.587/0.579/0.135	17.1/17.1/5.91	18.5/18.5/6.43	0.560/0.558/0.124
10000	2000	0.732/0.684/0.240	0.717/0.670/0.229	23.5/23.5/6.89	22.1/22.0/6.85	0.609/0.591/0.168

として用い，要求データサイズと応答データサイズは，それぞれ 128 byte と 10 kbyte に固定している.

5.1　サービス遅延時間

まず，サービス遅延時間の大まかな傾向を見るために，アクティブ率を 1（100 %）に設定した場合のサービス遅延時間特性を表 3 に示す．この結果からまず分かるのは，サービス遅延時間をその傾向によって，select・poll のグループ（ポーリング I/O グループ）と，MP・MT・epoll のグループ（非ポーリング I/O グループ）に分けることができることである．ポーリング I/O グループのサービス遅延時間は非ポーリング I/O グループのそれより大きく，記述子総数にほぼ比例している．ポーリング I/O は，記述子テーブルの走査を行うため，記述子総数が増加するに従ってその分処理コストが大きくなるのである.

また，記述子総数に対してスケーラビリティに問題があるとされることが一般的な MP モデルと MT モデルにおいて，記述子数を最大で 10,000 にまで増加しても性能的な劣化がほとんど見られず，epoll モデルと同等の平均サービス遅延時間を達成している．これには 2 つの要因があると考えている.

1 つは，今回の実験で用いた Linux バージョン 2.6 系のカーネルが，プロセスとスレッドのスケジュール処理にほとんど差異がなく，さらにはスケーラビリティ向上のための数多くの改良が導入されていることである [6]．そのため，多数のプロセスやスレッドを起動してもオーバヘッドが小さいという特長を有している.

もう 1 つは，各クライアントからの最大データ送信レートとサービス遅延時間の関係から，サーバにおける実行キューが十分小さい範囲に収まったこと

表4 サービス遅延時間特性（記述子総数 = 10,000，左: 平均値，中: 中央値，右: 標準偏差，単位: ミリ秒）.

アクティブ率	レート	MP	MT	select	poll	epoll
1/4	125	0.576/0.574/0.125	0.596/0.572/0.147	20.6/20.2/6.91	21.9/21.7/7.45	0.598/0.573/0.140
1/4	500	0.581/0.575/0.135	0.574/0.565/0.135	17.2/17.0/5.92	18.7/18.6/6.45	0.565/0.561/0.128
1/4	2000	0.655/0.606/0.232	0.620/0.595/0.201	23.3/23.3/6.86	21.5/21.5/6.59	0.603/0.582/0.174
1/16	125	0.568/0.566/0.124	0.560/0.558/0.117	20.5/19.8/7.27	22.0/21.7/7.71	0.557/0.555/0.126
1/16	500	0.580/0.573/0.139	0.575/0.568/0.145	17.2/17.1/5.96	18.5/18.4/6.46	0.559/0.555/0.130
1/16	2000	0.630/0.586/0.247	0.587/0.573/0.210	19.1/19.0/6.03	21.4/21.4/6.58	0.543/0.540/0.178
1/64	125	0.560/0.559/0.127	0.555/0.552/0.131	19.9/19.3/7.04	21.2/21.1/7.74	0.548/0.548/0.123
1/64	500	0.529/0.532/0.153	0.533/0.533/0.157	16.6/16.5/5.87	18.5/18.5/6.47	0.522/0.525/0.150
1/64	2000	0.450/0.410/0.167	0.442/0.400/0.163	19.0/18.8/5.99	21.0/20.8/6.51	0.454/0.412/0.165

である．実験中，各クライアントは最大で毎秒 400 回のデータを送信しており，その場合でリクエストの送信間隔は 2.5 ミリ秒である．一方，サービス遅延時間は 1 ミリ秒未満であり，サーバでは最大でクライアントの数（すなわち 5）までしか実行キューは長くならない．

次に，記述子総数を 10,000 に設定し，アクティブ率を変化させたときのサービス遅延時間特性を表 4 に示す．これらの結果から，アクティブ率が小さくなる（通信の局所性が高まる）につれ，特に非ポーリング I/O グループのサービス遅延時間が小さくなっていることが分かる．これは，通信が発生する記述子が限定されると，処理におけるメモリアクセスの局所性が高まることにより，メモリキャッシュのヒット率が向上し，処理能力が向上したことが要因だと考えられる．

表5 送信タイミングのずれにおける標準偏差の最大値（単位: マイクロ秒）

	MP	MT	select	poll	epoll
標準偏差	7.08	7.15	13.7	14.5	8.11
中央値	-0.214	-0.134	-2.53	-2.60	-0.271
最大値	337	824	667	904	396

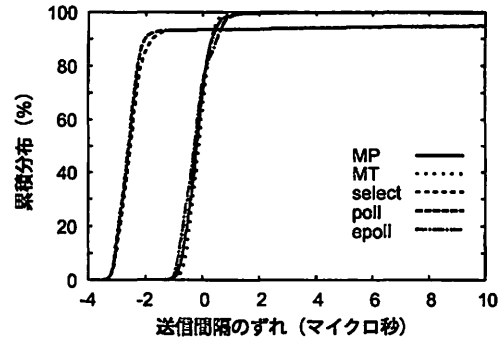

図2 送信タイミングのずれの累積分布（標準偏差が最大の場合）

5.2 要求データ送信タイミング

次に，クライアントにおいて要求データが設定されたレートで決定されるタイミングで正しく送られていたのか検証する．具体的には，設定された送信レート情報を用いて正規化された要求データ送信タイミングからの実際の送信のずれを算出した．

d を設定された送信レートから計算される要求データの送信間隔とし，$t_1, t_2, \cdots, t_n$ を実際の送信タイムスタンプ列とする．このとき，t_i^* を

$$t_i^* = \frac{1}{n}\sum_k t_k - \frac{(n-1)d}{2} + (i-1)d$$

によって定義すると，$\sum(t_i - t_i^*)$ は 0 となる．つまり，$t_1^*, t_2^*, \cdots, t_n^*$ は正規化された要求データ送信タイミングとなる．よって，この $(t_i - t_i^*)$ の分布を調査した．

この送信タイミングのずれが大きいと実験の精度の観点から問題になることから，今回の実験においてそのずれの分布の標準偏差の最も大きかったものをネットワーク I/O 管理機構別に取り出して検証した．表 5 にその結果を示す．また図 2 に，ずれの累積分布を示す．

このように，送信タイミングのずれについてもポーリング I/O のグループと非ポーリング I/O のグループに分けることができる．非ポーリング I/O では，最大で数百マイクロ秒の大きなずれが少数発生するも

のの，おおむね ±1 マイクロ秒のずれに収まっている．今回の実験では，各クライアントは最大で毎秒 400 回の要求データ送信を行っており，その場合の送信間隔 d は 2.5 ミリ秒であることと比較すると十分高い精度が実現できていると言うことができる．

一方ポーリング I/O では，10 マイクロ秒を超えるずれが 5 % 程度の送信において発生し，その影響によりずれの中央値が小さくかつ標準偏差が大きくなっている[*2]．しかし，90 % を越える送信は ±1 マイクロ秒のずれ（約-3.5 から-1.5 マイクロ秒の範囲）に収まっていることも分かる．また，ずれの最大値も 1 ミリ秒以下であり，実験結果に深刻な影響を与えるものではなかった．

6 まとめ

本研究では，多数のコネクションを扱うシステムのアーキテクチャ，すなわち I/O 管理機構の性能比較評価を行うベンチマークツールを試作した．

実装においては，I/O 管理機構の部分をモジュール化することで，MP モデル，MT モデル，select モデル，poll モデル，epoll モデルを切り替え可能とした．また，通信モデルではアクティブ記述子の概念を導入することにより，通信の局所性を設定可能にした．

Web トラヒックモデルを用いた Linux バージョン 2.6 上での各モデルの性能評価では，MP モデル，MT モデル，epoll モデルが select モデルと poll モデルより性能が優れていた．また，一般的には記述子数に対してスケーラビリティに劣るとされることの多い MP モデルおよび MT モデルが，epoll モデルとほぼ同等の性能を有していることも判明した．さらに，通信の局所性を高く設定すると，MP モデルと MT モデルだけでなく，epoll モデルでも性能が向上した．

今後は，BSD 系 OS の kqueue 機構や Solaris の poll デバイスファイル機構を用いた実装を完成させ，OS 実装の性能比較を行う予定である．また，ネットワーク環境をより多様化させ，遅延やその揺らぎ，パケットロスなどを設定した擬似的な広域ネットワーク環境下でのサービス性能を評価したいと考えている．

[*2] ずれの平均値は 0 であることに注意．

謝辞

本研究の一部は，日本学術振興会科学研究費補助金若手研究（B）（課題番号 18700061）による助成を受けている．

参考文献

[1] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pp. 199–212, Monterey, California, USA, June 1999. USENIX.

[2] Bill O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., 1995.

[3] Linux manual page of `epoll(4)`, October 2002.

[4] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pp. 141–153, Boston, Massachusetts, USA, June 2001. USENIX.

[5] Shridhar Acharya. Using the devpoll (/dev/poll) Interface. Technical Articles, March 2002. `http://access1.sun.com/techarticles/devpoll.html`.

[6] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.