Java を用いたネットワークプログラミングスタイルの比較

Comparing network programming style using Java

屋比久 友秀 1, 河野 真治 2
Tomohide YABIKU, Shinji KONO

本稿では、Java 標準の API を用いて、1) Raw Socket, 2)Multi-Thread, 3) Non-blocking I/O の 3 種類のプログラミングスタイルによる 1 対 N の通信性能を評価した。通信性能は、メッセージサイズとノード数を変化させた場合のラウンドトリップタイムを計測した。また、Raw Socket の場合は、他の場合と比較してレスポンスタイムが悪く。ノード数の増加によるレスポンスの劣化が顕著に表れた。Multi-Thread の場合は、メッセージサイズが比較的小さい場合 (十数 K バイト以下) では、台数による変化が少なく安定したレスポンスを得られた。Non-Blocking I/O を用いた場合は、メッセージサイズの小さい領域では、最大で 4 倍程度 Multi-Threads と比較して悪くなるが、メッセージサイズの大きな領域 (約 60K バイト) では、Multi-Threads とほぼ同等か数十パーセント程度、レスポンス時間が良いことがわかった。

1　はじめに

ブロードバンドの普及によって、大規模かつデータ転送量の多いネットワークアプリケーションが利用されるようになってきた。その例として、オンラインネットワークゲームや P2P アプリケーションが典型的な例である。これらのアプリケーションは、参加可能な人数は大規模であるが、同時にインタラクション可能なユーザ数は、数人からせいぜい数十人程度である。これらのアプリケーションには、中央集中サーバ型のシステムで運用されているのがほとんどである。この場合、ユーザ数が多くなるにつれて、中央サーバの負荷が大きくなり、アプリケーションに必要とされるレスポンスが得られない場合がある。我々は、このような問題を解決するために、数百万のユーザが同時にインタラクション可能なネットワークアプリケーションの研究を行ってきた。このような大規模なユーザ間で通信を行うには、中央集中サーバ型の論理ネットワークでデータ転送を行うのではなく、インターネット上に分散配置された Agent によってデータ処理を行う方法がよいことがわかってきた [1][2]。

我々は、ツリー構造に Agent を配置し、この Agent に沿ってアプリケーションのデータ転送を行う Agent システムを開発した [1][2]。このフレームワーク上では、数千ユーザが同時にインタラクション可能なことがわかっている。しかしながら、数百万ユーザまで到達していないのが現状である。数百万ユーザの同時インタラクションを実現するためには、単純なツリー構造に Agent を配置するのではなく、ツリー構造の子ノードの数を増やしたり、メッシュ構造、ハイパーキューブなどの他のネットワークトポロジを用いて Agent システムを構築し、アプリケーションを実際に動作させて、大規模なユーザに耐えられるロバストなシステムを構築、検証する必要がある。

これらのネットワークトポロジを利用してネットワークアプリケーションを構築するには、どのような API を利用し、かつ、どの程度のデータ転送、レスポンスが得られかを前もって知っておく必要がある。

我々は、ネットワークトポロジを構成している Agent の性能を調べるために、1 台のノードに対して、どれくらいのノードを結合し、どの程度のデータ転送量でが最も効率が良いかを調べた。

[1]琉球大学理工学研究科総合知能工学専攻
yabiku@cr.ie.u-ryukyu.ac.jp
Intelligent System Engineering, Graduate School of Engineering and Science, University of the Ryukyus

[2]琉球大学工学部情報工学科
kono@ie.u-ryukyu.ac.jp
Infomation Engineering, faculty of engineering, University of the Ryukyus

2　ネットワークトポロジ

図 1 に我々が考えているネットワークトポロジを示す。黒丸で表されたものが Agent に対応するノードで、これらの Agent に対して、ユーザが利用する PC などの端末から接続し、アプリケーションを実行する。Agent は、データの集計や加工を行い指定されたノードへデータ転送を行うことを想定している。

これらのネットワークトポロジで、一つのノードに着目すると基本的な構成は、1 対 N の通信であること

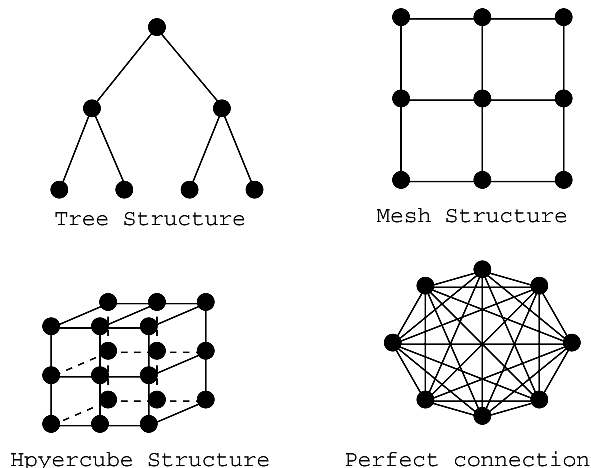

図 1: Network Topology

がわかる。Tree 構造では、1 対 3 の通信、Mesh 構造では、1 対 4 の通信、Hypercube 構造では、1 対 6 の通信が基本的な通信経路である。Perfect Connection(完全結合) では、1 つの Agent に対して、(全 Agent 数-1) の通信経路が必要である。これも基本的に 1 対 N の通信である。

これらの事から、いずれのネットワークトポロジでも、基本的には、1 対 N の通信を組み合わせて構築されていることがわかる。これらのネットワークトポロジでアプリケーションを構築する場合は、1 対 N 通信の性能を調べることで、全体の性能を見積もることができる。

3 1 対 N 通信プログラミング

以下に今回の実験で利用したプログラミングスタイルと Java 標準の API の通信方法について説明する。

3.1 Raw Socket

Raw Socket は、一般的なネットワークアプリケーションで利用されている単純なソケット通信である。図 2 にそのシーケンス図を示す。

Raw Socket の場合は、Server 側で他の Client と通信している場合は、処理が完了するまで待ち、その処理が完了したらデータ転送を行うという特徴がある。

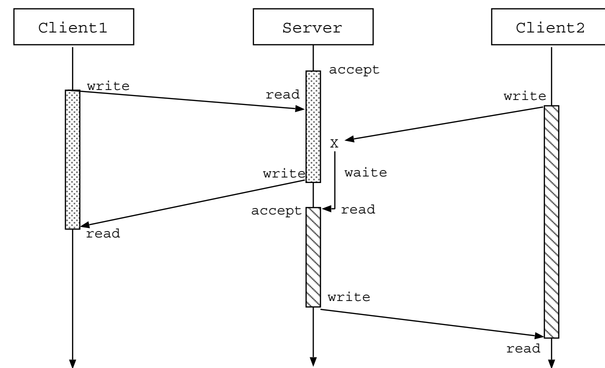

図 2: Raw Socket

3.2 Multi-Thread

Multi-Thread の場合は、他の Client が Server に接続している場合に、新たに接続要求を Server に送ると新規 Thread を生成し、その Thread 上で Client と通信を行うことができる。図 3 にそのシーケンス図を示す。新規ノードからの、データ転送の必要性がある場合にも、既存の接続ノードのデータ転送が完了するのを待たずにデータ転送が可能であることが特徴である。

Multi-Thread を用いてアプリケーションを構築する場合は、生成する Thread の数や、データ処理を行う Synchronized メソッドを利用する場合には注意する必要がある。Thread の数を無制限に増やせば、それだけリソースを消費してしまうからである。同時接続数を正確に見積り、必要な Thread 数を生成できるようにしなければならない。

Synchronized メソッドを利用する場合は、他のスレッドからアクセスできないようにロックするため、処理に時間がかかるような処理の場合は、結局、待ち時間が発生し全体としてパフォーマンスに影響がでるため、データ転送の効率が悪くなる場合がある。

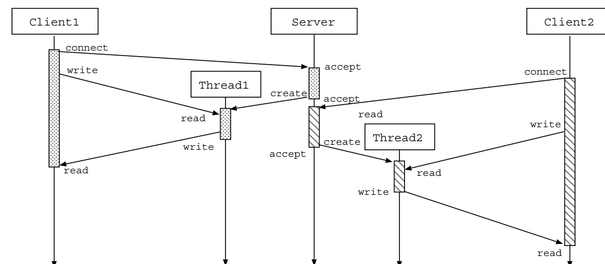

図 3: Multi-Threads

3.3 Non-blocking I/O

Non-blocking I/O は、JDK1.4 から新たに標準 API として追加された API である。特徴的なのは、非同期通信サポートしていることである。非同期通信では、転送するデータが完全に揃わなくても、他のノードからデータを受信することができる。これによってデータ転送が完全に終るのを待たずに、新規ノードからデータ転送を行うことができる。図 4 に、Non-blocking I/O のシーケンス図を示す。

Multi-Thread なプログラミングと比較して、Thread の数や Synchronized メソッドを気にする必要がなくアプリケーションを構築することが可能である。

しかしながら、Non-blocking I/O を用いた通信は、一般的にソースコードが複雑になる傾向がある。これは、データ転送が完全に終るのを待たずに、データを受信するため、受信データのバッファをプログラミング側で管理する必要があるからである。また、現在の JDK では、データ通信可能なデータ型は、Java がサポートしているプリミティブなデータ型だけをサポートしている。シリアライズされたオブジェクト転送を行う場合は、入出力のストリームバッファを自前で作成する必要がある。

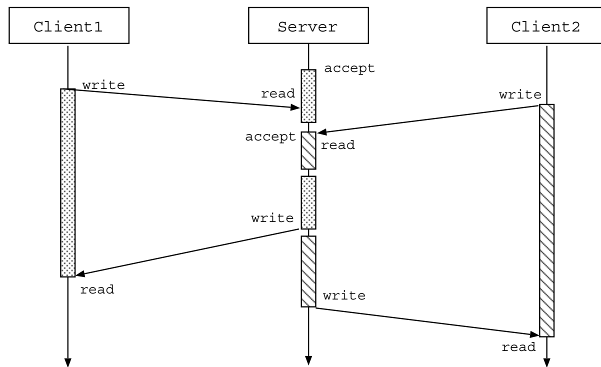

図 4: Non-blocking I/O

4 測定方法

測定方法は、図 5 に示すように、1 台のサーバに対して、複数のクライアント (最大 30 台) を用意し、それぞれのクライアントからサーバに対してデータを転送し、それをサーバ側からクライアントへ同じデータを送り返すという実験を行った。

データサイズは、1024 バイトから、65536 バイトまで変化させて計測した。また、スケーラビリティを計測するために、1 台から 30 台までクライアント数を変化させて計測を行った。

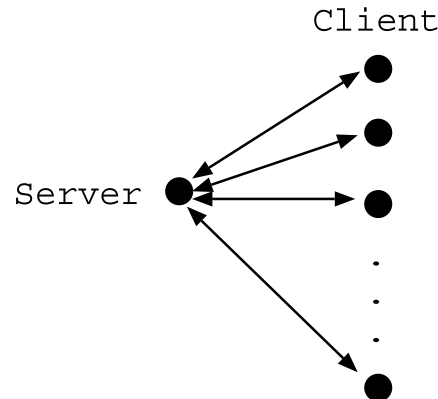

図 5: Experimental Method

表 1 に評価を行ったコンピュータおよびネットワーク環境を示す。

表 1: 評価環境

CPU	Pentium3 800MHz
キャッシュ	L1 キャッシュ:32KB L2 キャッシュ:256KB
メモリ	512Mbyte/node
ディスク	10GB/node
MB Base Clock	133Mhz
NIC	EtherExpressPro 100
スイッチ	Catalyst C2980-GA (back-borne 6GB/sec)
OS	Linux 2.4.26
JDK	JDK5.0 update 4 (Sun Microsystems)

5 ベンチマーク結果

以下に、それぞれの API およびプログラミングスタイルを用いた場合の測定結果を示す。

レスポンスタイムは、1 ノードがデータを送信して受信するまでの平均時間を計測した。

クライアント数は、1 台、5 台、10 台、30 台の場合についてプロットした。

また、レスポンスタイムのグラフからサーバのスループットを計測した。計算方法は、以下の式で行なった。

スループット $[byte/sec] = \frac{ノード数 \times データサイズ}{レスポンス時間}$

5.1 Raw Socket の測定結果

Raw Socket の測定結果を図 6 に示す。
クライアント数に応じてレスポンス時間が大きくなるのがわかる。また、データサイズが 8K バイトを越えたあたりから、徐々にレスポンスタイムの開きが大きくなっているのが特徴である。

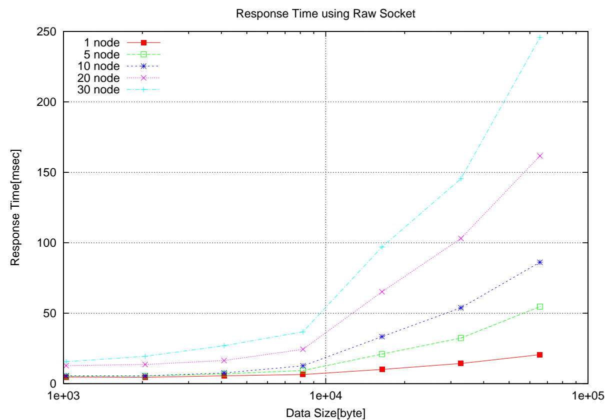

図 6: Response time using Raw Socket

図 7 にサーバのスループットを示す。Raw Socket を用いたスループットでは、最大で 8M byte/sec のスループットを得た。グラフでは、クライアント数が変化しても、スループットにはそれほど影響がでないことがわかった。

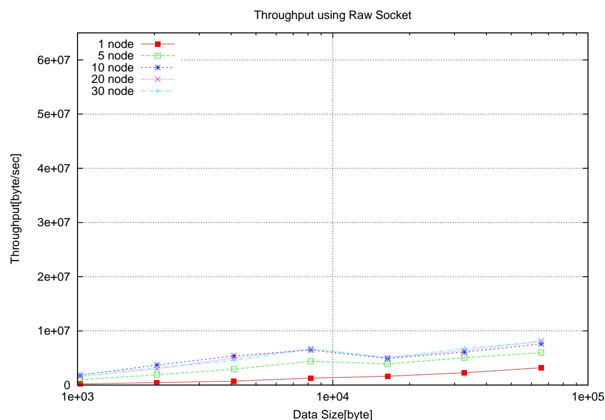

図 7: Throughput using Raw Socket

5.2 Multi-Threads の測定結果

Multi-Threads を用いた場合の測定結果を図 8 に示す。

Multi-Threads を用いた場合は、データサイズが小さい領域 (1.6K バイトまで) は、クライアント数が多くてもそれほどレスポンスタイムに差はなかった。データサイズが大きくなると、クライアント数に応じてレスポンス時間が長くなるが、クライアント数の増加数を考慮すれば、レスポンスの劣化が比較的穏やかである。

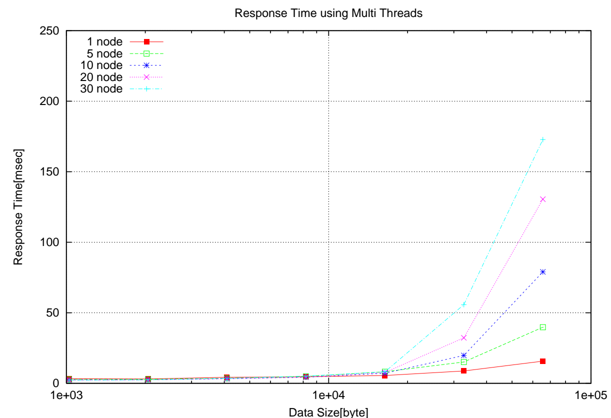

図 8: Response time using Multi-Threads

図 9 に Multi-Threads を用いた場合のサーバのスループットを示す。Multi-Threads の場合には、データサイズとノード数によって顕著に変化が表れている。ピーク性能では、30 ノードの時に、メッセージサイズが 16K バイトで、60M byte/sec の性能が出ている。

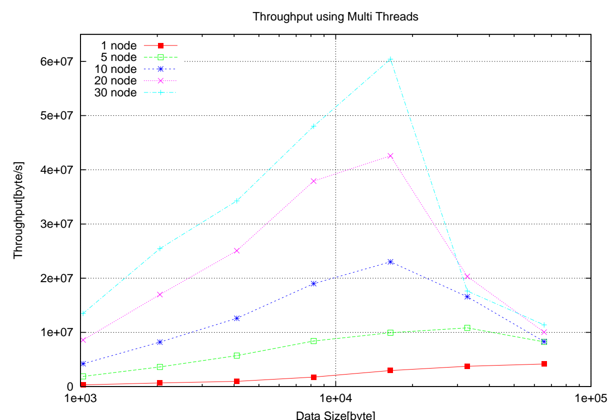

図 9: Throughput using Multi-Threads

5.3 Non-blocking I/O の測定結果

Non-blocking I/O を用いた場合の測定結果を図 10 に示す。

Non-blocking I/O では、クライアント数の増加によるレスポンス時間の影響が素直に表れている。同じメッセージサイズでは、クライアント数に比例した結果になった。また、Multi-Threads と比較した場合、メッセージサイズの小さい領域では、Non-blocking I/O の方がレスポンスが悪いが、メッセージサイズの大きい領域では、Non-blocking I/O の方がレスポンスが良いことがわかった。

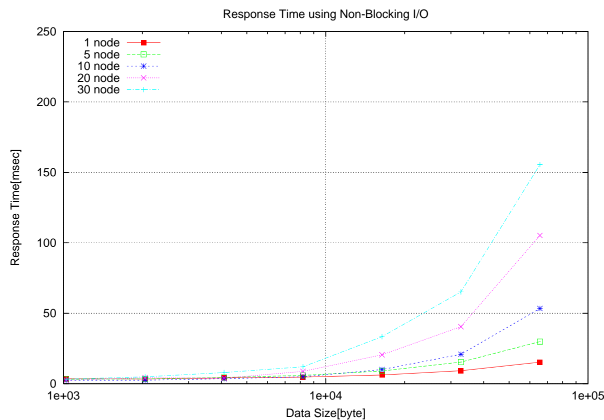

図 10: Response time using Non-Blocking I/O

図 11 に Non-blocking I/O を用いた場合の、サーバのスループットを示す。Non-blocking I/O では、メッセージサイズ、ノード数によって比較的変化は少なく、安定したスループットを示している。ピーク性能では、20 ノード、メッセージサイズが 4Kbyte の時に 20M byte/sec の性能を出している。

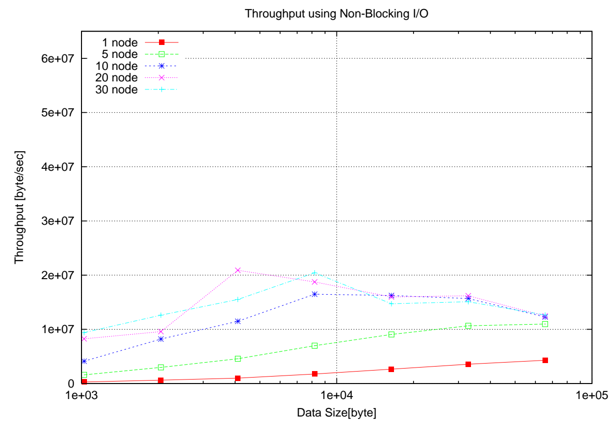

図 11: Throughput using Non-Blocking I/O

6 考察

Raw Socket では、クライアント数の増加とメッセージサイズの増加によるレスポンスの劣化が顕著に表れている。これは、データ転送の際に発生する待ち時間が影響している。データサイズが大きくなればなるほど、転送されるデータを全て受信するまで待つため、それだけ待ち時間が長くなり。また、実際にデータ転送を行なえるクライアントは 1 台だけなので、それだけ、待ち時間も長くなる。他の 2 つの方法と比較すると、数倍から数十倍のレスポンス時間を必要とする。

Multi-Threads では、データ転送に待ち時間が発生しないため、比較的レスポンス時間が短く良い結果を得ることができた。しかしデータサイズが大きくなると各スレッドが利用するバッファが大きくなり、リソースを消費する傾向になるそのためデータサイズが大きい場合では、Non-Blocking I/O よりもレスポンス時間が長くなっている。また、今回の測定では行なえなかったが、データを処理する場合にメインスレッドで処理をする場合は、synchoronized メソッドを用いる場合がる。この場合は、各スレッドで待ち時間が発生し、今回の結果とは異なる結果になる可能性がある。

Non-blocking I/O では、データサイズが小さい場合には、Mult-Threads と比較して数倍程度、レスポンスは悪くなるが、データサイズが大きくなると Multi-Threads の場合とほぼ同じレスポンスを返すことがわかった。これは、Non-blocking I/O の場合には、転送データが完全に受信する前に、他のノードからデータを受信するため、データの組み立てを行なう必要がある。そのため、データサイズが小さい場合には、このデータの組み立てにかかるオーバヘッドが大きくなるからである。逆にデータサイズが大きい場合には、データの組み立てにかかる時間が相対的に小さくなるからである。

7 まとめと今後の課題

Raw Socket, Multi-Threads, Non-blocking I/O の 3 種類のプログラミングスタイルで 1 対 N の通信性能を比較、評価した。Raw Socket を用いた通信が性能が悪いことがわかった。今回の測定では、メッセージサイズが小さい場合は、Multi-Threads が性能

が良く、メッセージサイズが大きい場合には、Non-blocking I/O の性能が良いことがわかった。しかしながら、今回の測定では、最大のメッセージサイズが 65K バイト程度であるため、これ以上大きなメッセージで Non-blocking I/O の性能に変化があるのかを確認する必要がある。また、実際の Agent システムに今回検証したプログラミングスタイルを適用し、性能評価を行なう必要がある。

今後の課題として、Java 標準の API だけでなく、我々が開発した UDP ベースの通信ライブラリ Suci[3][4][5][6] を用いて、今回のような測定を行ない、より大規模なネットワークへ拡張可能であるかを検証を行なう必要がある。

参考文献

[1] 小杉 隆二 , 河野 真治. Tree 構造と Mesh 構造に対応した大規模ネットワークゲーム AgentSystem を用いたシミュレーション. 日本ソフトウェア科学会第 21 回大会論文集, Sep, 2004

[2] 小杉 隆二, 河野 真治. Tree 構造と Mesh 構造に対応した大規模ネットワークゲーム Agent. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), June, 2004

[3] 山城 潤, 河野 真治. Java によるユーザトランスポート層の実現と評価. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May, 2003

[4] 山城 潤 , 河野 真治. 通信ライブラリ Suci for Java の性能改善と評価. 日本ソフトウェア科学会第 21 回大会論文集, Sep, 2004

[5] 屋比久 友秀, 河野 真治. Suci を用いた高レベル通信ライブラリの設計. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May, 2003

[6] 屋比久 友秀, 河野 真治. 並列分散ライブラリ Suci の実装と評価. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), June, 2002