システムソフトウェアと
オペレーティング・システム 84−14
（２０００．５．２５）

Javaによる TCP/IPプロトコルスタックの設計と実装

立野広樹（慶応義塾大学　政策・メディア研究科）
萩野達也（慶応義塾大学）

本研究では、Javaによる TCP/IP プロトコルスタックの設計と実装を行う。Javaでシステムモジュールを記述した場合、Java仮想機械が搭載されているすべての OS において、即時に実行可能な OS を実現する事ができる。この手法は、プロトコルスタックに限定する必要はなく、その他のシステムモジュールにも適用する事ができる。しかし、この手法には様々な問題点が存在する。まず、Javaはハードウェアへの直接的なアクセス方法を提供していない。また、Java は C 言語の持つ高度なシステムプログラム記述力を持っていない。そのため、制限された記述でどのようにシステムモジュールの設計実装を行うかという点も問題となる。本研究では、これらの問題に対する解決策を提案し、Javaで記述されたプロトコルスタックの設計と実装を行った。また、Javaによるプロトコルスタックの実現可能性と利点、性能の評価を行った。性能を評価した結果、JRE1.2.2 上で ping レーテンシ 0.62msec.、UDP スループット 521KB/sec. といった測定結果を得た。以上の結果により、Javaを用いたシステムモジュールの十分な実現性を示した。

Design and Implementation of TCP/IP Protocol Stack using Java

Hiroki Tateno　　Tatsuya Hagino
Keio University

In this research, we have implemented a new TCP/IP protocol stack in Java. We also present effectiveness and feasibility of constructing system software in Java. Writing system modules in Java makes operating system runnable on any platform which has Java Virtual Machine. This approach can be applied not only for protocol stacks, but also for any other system modules. However, writing system modules in Java has several problems. One is how to handle devices, because Java cannot access hardware directly. Another one is lack of low-level abstractions like ability to handle raw byte-arrays, which is necessary to implement system modules. In this research, we propose a solution for these problems, and designed and implemented a new TCP/IP protocol stack using Java. We have evaluated the performance of our protocol stack by measuring ping latency and UDP throughput. The result shows our protocol stack can be used in practice: ping latency is 0.62msec on JRE1.2.2 and UDP throughput 521KB/sec on avarage.

1 はじめに

本研究では、そのほぼすべての部分をJavaによって記述したTCP/IPプロトコルスタックを、設計、実装する。このシステムは、カーネル、プロトコルスタックなどのシステムモジュール、アプリケーションが同一保護空間上で実行される事を前提とする。アプリケーション層へのインタフェースは、java.netクラスライブラリと互換性を保つ。これによりユーザは、蓄積したアプリケーションを変更する事なく本研究における実装を使用する事ができる。ただし、ハードウェアへの直接アクセスの部分において、一部、C言語を用いて記述したプログラムを用いる。なぜならば、Javaはアーキテクチャ非依存を達成するためにハードウェアへの直接アクセスをサポートしていない為である。更に、ハードウェアにアクセスしない仮想デバイスドライバクラスを用意する事で、ハードウェアから完全に独立した状態でのプロトコルスタックのシミュレーション機能を提供する。プロトコルスタック本体は、各層別にいくつかのモジュールに分割し、独立したスレッドとして動作させる。また、Javaのモニタ機能を使用し、スレッド間の共有データの受け渡し機構を提供する。このような設計により、モジュールの高い独立性と、モジュールの追加及び削除における柔軟性を達成している。

2 Javaを用いる際の問題点

2.1 表現力

また、C言語においては、ポインタと配列の密接な関係や自由なポインタキャストが、システムモジュールを記述する上での非常に強力な表現力を提供している[6]。これは、コンピュータハードウェアが備えるメモリマップという概念に依存した原始的な機能である。より抽象度の高いJavaにおいてはこれらの手法を直接使用する事ができない為、同等の機能を極端な性能低下を避けつつどのように実現するのかが焦点となる。

2.2 ハードウェアへのアクセス

OS上で実際に稼働するシステムジュールを実現する為には、ハードウェアへのアクセスをどのように行うかという事が問題となる。Javaプログラムから「ハードウェア割り込み」を操作する為には、ハードウェアレジスタを操作しなければならない。しかし、Javaはメモリマップという概念を持たない為、これを行う事は不可能である。よって、ネットワークインタフェースから発生する割り込みを、C言語やC++言語で記述したネイティブメソッドの中に隠蔽しなければならない。ここで、どの部分でネイティブメソッド[4]で記述するかという問題が発生する。

最も低いレベルのデバイスアクセスをJavaで記述することは、非常に困難である。デバイスのアクセスは通常、クラスライブラリの中でホストOSの機能を呼び出す事で実現されている。よって、最も低いレベルのデバイスアクセスは、ネイティブメソッドを用いて記述する。本研究では、バイト列をネットワークカードに書き出すwriteメソッドと、ネットワークカードからバイト列を受け取るreadメソッドの両方をネイティブメソッドとして記述する。これにより、ハードウェアヘッダを操作を代表とする、ネットワークインタフェース層における高レベルの処理についてはJavaで記述する事が可能になる。

3 設計と実装

今回設計したプロトコルスタックの全体像を、図1に示す。

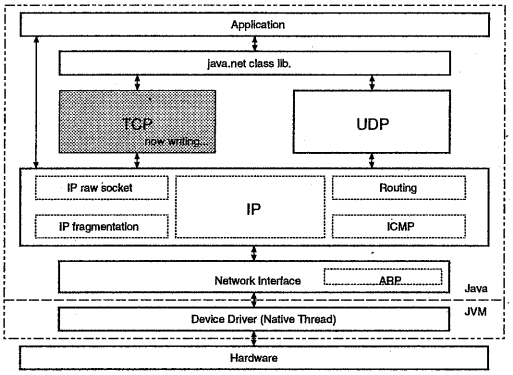

図 1: 全体概要

3.1 バッファ管理クラスの設計

TCP/IP プロトコルスタックでは、バイト列を確保し操作する部分の実装が、全体性能に大きな影響を与える。故に、できるだけバイト列のコピーを押さえたバッファ確保クラスを設計する必要がある。しかし、C 言語を利用した BSD の mbuf のような汎用メモリバッファを Java において実現するのは、Java の仕様上、難しい。しかし、Java にはポインタが無いかわりに、参照が存在する。これは、ポインタのようにアドレスを自由に変更できない欠点を持つ一方、厳密な参照数の管理とガベージコレクションが可能であるという利点を持つ。参照を利用して、C 言語におけるポインタと連結リストを活用したバッファ管理と異なる、Java の利点を生かしたバッファ管理クラスを設計する事が可能である。このクラスは、メモリ確保の厳密な一貫性を保つ事により、安全性を持っている。本研究では、2 種類の異なるバッファ管理クラスを設計し、それらの間の性能比較を行う。

設計したクラスの一方は、パケットをひとつの大きなバイト列として扱う設計である。バッファの入出力、分割、併合操作は、パケットが持つバイト列をコピーする事で実現する。これは、単一バッファモデルの設計であり、単純に実装できる為に速度性能が要求されない場合に使われる事が多い。しかしこの設計では、操作するバイト数に比例して操作コストが上昇するため、大きなバイト列の複雑な操作を行う場合は、それに比例した処理時間を要する。単一バッファモデルの実装は、Java ベースの Appletalk プロトコルスタックである hotlava[2] や、Java クラスライブラリ内の StringBuffer クラスなどで実際に使われている [5]。

もう一方は、参照にオフセットと長さを組み合わせたバッファ設計である。このバッファは、エレメントとパケットという二層構造を用いて構成される。

エレメントは、バイト列と、オフセットと長さを組み合わせたデータ構造である。図 2 に、エレメントの構造を示す。byte[] がバイト列であり、それに対してオフセットと長さが与えられ、バイト列の一部分が指定されている。Java はポインタを持たない為、バイト列は、バイト型の配列という意味以外のものを持ってない。よって、バイト列のある部分を指し示すポインタを作る事はできない。この為、参照の操作によってバイト列を自由に分割及び併合する事は不可能である。しかし、バイト列に対して、先頭を示すオフセットと長さを与え、任意の場所への入出力を実現するオブジェクトメソッドと組み合わせる事で、同等の機能を提供可能である。また、バイト列の一部分をコピーする場合、先述の古典的なバッファ実装では領域をコピーする必要があるが、エレメントの場合、バイト列本体に対するコピーは参照のみに止め、オフセットと長さを変更す

る事で、バイト列から一部分を取り出す操作を大幅に簡略化する事が可能である。

しかし、エレメントでは、ヘッダの分離や断片の再構成といった、バイト列の分離と再構成を伴う最適な操作を実現する事ができない。この為、エレメントを双端単方向連結リスト（先頭と末尾に対する参照を持った連結リスト）で結合することで、パケットという上層構造を実現する。図3に、パケットの構造を示す。このパケットに対しては、エレメントの追加、削除、パケットサイズの変更、パケットを複数のエレメントに領域分割するメソッドを実装する事で、TCP/IPプロトコルスタックに必要とされる各種の操作に、より最適化されたクラスを実現する事ができる。

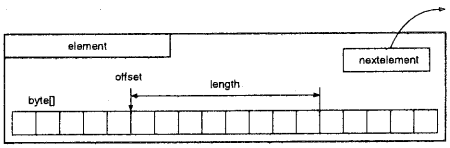

図 2: エレメント

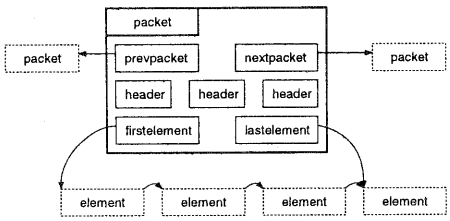

図 3: パケット

3.2 バイト列アクセスの設計

プロトコルスタックにおいては、バイト列を、構造をもったデータとしてアクセスする必要がある。多くのプロトコルスタック実装では、バイト列へのポインタを、各層のヘッダ構造体にキャストする事で、プロトコルヘッダ操作を実現している。構造体のメモリ格納イメージが判明している場合、バイト列に対するポインタを任意の構造体にキャストが可能である。しかし、この実装は、ハードウェアやコンパイラの設計に依存している。ポインタキャストを用いたアクセスは極めて高速であるが、バス幅の異なるCPU間の移植を行うような、CPUアーキテクチャが大きく変更される場合に問題が発生する。

また、Javaではバイト列の参照をクラスにキャストする事は不可能である。Javaにおけるキャストは変換メソッドであり、代入とは全く異なっている。内部構造が定まってない領域間で参照を交換する事は、言語の安全性を脅かす為である。不定領域間のキャストはJavaの設計時に、安全性を向上させるという明確な意図に基づいて排除されている。

本研究では、バイト列のポインタからpacketオブジェクトへのキャストではなく、バイト列ラッパークラスを設計するアプローチをとった。このクラスは、バイト列と各種プロトコルヘッダの相互変換を行う。

3.3 データ交換と制御切替の設計

本研究では、プロトコルスタックの各モジュールをスレッドとして実現する。Javaでは、スレッドとスレッド間の同期が言語的にサポートされている[3]。この為、プロセス指向的な実装を行う事でよりシンプルな設計と実装を実現する事ができる。また、各スレッド間のパケットの流れを、Javaにおけるモニタ機能で実装している。synchronized宣言された部分はモニタとして扱われ、複数のスレッドからの同時アクセスを制限する事が可能である。

各層を実現する為のスレッドと、スレッド間でデータを共有するためのモニタを分離する設計によ

り、オブジェクトを切り替える事で実装の動的な交換を行う事が可能である。これにより、使途によってプロトコルスタックの実装を変更するといった大きな設定可能性が実現可能である。

モニタを用いたスレッド間のパケット交換用キュークラスは、enqueue と dequeue というメソッドを持つ。それぞれ、キューにデータを追加／取得する機能を持つ。また、dequeue する場合にキューが空だった場合、enqueue する場合にキューが満杯だった場合は、wait 及び notify といったオブジェクトの同期メソッドを用いることでスレッド間の同期を実現する。このように、モニタの中にすべてのスレッド間同期の機能を実装する事で、スレッド側からは enqueue 及び dequeue のメソッドにスレッド間同期にカプセル化している。

本研究では、このキューをスレッド間データ交換に使用する。入力時にキューが空である場合や、出力時にキューが溢れている場合、wait 及び nofity がスレッド間に通知されスレッドの切り替えが発生する。このように、各スレッドにおける実行の流れを明確にする事で、よりプロトコルスタックの構造を単純にする事が可能になる。

3.4 実装

本研究では、ネットワークインタフェース層、ネットワーク層、トランスポート層、アプリケーション層について実装した。具体的には、ネットワークインターフェース、ルーティングテーブル、IP、UDP、API の 5 つのモジュール群について実装した。API については、Sun の提供する Java クラスライブラリで提供されている DatagramPacket、DatagramSocket、InetAddress の 3 つのクラスと互換性を保ったクラスを実装した。また、評価を進めるために使用したテストアプリケーションとして、単純な UDP 転送を行うサーバ／クライアントと ping のクラスを実装した。アプリケーションと API 部分を含めた実装行数の総計は 6468 行となる。これらのモジュールはマルチスレッドで構成されている。ネットワークインタフェース、IP、IP タイマ、アプリケーションがそれぞれ独立したスレッドである。これらのスレッドは、JVM の中でスケジューリングされ、協調して動作している。

本研究の評価は、SONY VAIO N505 上で行われた。また、Java 開発環境は、Blackdown JDK 1.1.8 for Linux glibc2 を用いた。

4 評価

本研究では、大きく 3 分野について評価を行った。まず、ハードウェア環境、ソフトウェア環境、パラメータを固定し、複数の異なる実装における測定を行った。次に、ハードウェア環境、実装、パラメータを固定し、ソフトウェア環境を変化させた場合の測定を行った。最後に、ハードウェア環境、ソフトウェア環境、実装を固定し、パラメータを変化させた場合の評価を行った。

4.1 測定項目

レーテンシを測定するために ping の応答速度を測定した。1000 パケット送信時の平均値を測定した。

スループットを測定するために、UDP パケットの送受信を行い、1 秒間に転送したバイト数を計測した。1000 パケット転送し、1 秒あたりの転送バイトを測定した。UDP パケットの送信にかかる時間を測定した。

プログラムが send メソッドを実行した時点から、

	単一バッファ	参照バッファ
スループット	62.24 KB/sec.	105.31 KB/sec.
送信	3.11 msec.	3.06 msec.
受信	3.51 msec.	0.23 msec.
分割	40.19 msec.	20.96 msec.
再構成	55.86 msec.	0.95 msec.

表 1: 実装による変化

ネットワークインタフェースが書き込みを完了した時点までの時間を送信時間とし、1000 パケット送信時の平均送信時間を計測した。UDP パケットの受信にかかる時間を測定した。

ネットワークインタフェースがパケットを受信してからユーザがパケットを受け取る時点までを受信時間とし、1000 パケット受信時の平均受信時間を測定した。

5000 バイトのパケットを 1500 バイトのパケット 3 個と 500 バイトのパケットに分割する時間を分割時間とし、500 パケット処理時の平均値を測定した。

分割された 1500 バイトのパケット 3 個と 500 バイトのパケットを 5000 バイトのパケットへ再構成する時間を再構成時間とし、500 パケット処理時の平均値を測定した。

4.2 実装を変化させた場合

測定項目は、UDP スループット、UDP 送信時間及び受信時間、分割と再構成にかかる時間の 4 種類である。

測定結果を表 1 に示す。参照バッファクラスは、単一バッファ管理クラスに比べて、大幅な性能改善を実現することができた。UDP スループットにおいては、1.69 倍、UDP 受信においては 15.26 倍、パケット分割においては 1.9 倍、パケット再構成においては 58 倍の速度向上を実現した。これらの結果は、バイト列コピーを大幅に削減した点に基づく。

	MS	Sun	Blackdown
ping レーテンシ	0.60	0.62	1.48
UDP スループット	493.10	521.35	105.11

表 2: VM による変化

単一バッファ管理クラスでは、すべての操作をバイト列確保とコピーによって行っていたが、参照バッファでは、参照のコピーとインデックスや長さの変更を使用することでバイト列コピーを大幅に削減した。この為、コピーで行っていた事を単純な参照の張り替えのみで実現する事ができたパケット再構成に関しては、大幅な性能改善を行うことが可能になった。しかし、パケット分割においては性能改善は 2 倍にとどまっている。この理由は、バイト列の中途参照をする事ができないためである。この為、分割地点をバイト列のオフセットと長さに基づき、連結リストを検索しなければならなくなった。このコストは、参照の張り替えに比べて高い。

4.3 ソフトウェア環境を変化させた場合

測定項目は、ping レーテンシと UDP スループットの 2 種類である。測定に用いたソフトウェア環境は、Sun JavaRuntimeEnvironment 1.2.2 (OS : Windows98)、Microsoft JVM (OS : Windows98)、Blackdown JDK 1.1.8 (OS : Vine Linux 1.1 + Kernel 2.2.13 : single user mode) の 3 環境である。レーテンシの単位は msec.、スループットの単位は KB/sec. である。

測定結果を、表 2 に示す。異なる JVM 間のパフォーマンスに大きく幅が開いている。今回評価に用いた全ての JVM は JIT を搭載しているが、BlackdownJDK は他の JVM に比べて 5 倍程度遅

い。また、Solaris 上での JDK1.1.3 では、実装したプロトコルスタック自体が動作しなかった。これは、JDK1.1.6 以前のスレッド実装、特にスケジューリングに問題がある為である。このように、Java プログラム自体には問題が存在しない場合であっても、JVM の実装によって Java プログラムの動作に差が発生してしまっている。これは、JVM 実装の問題であり、移植性の高い OS を実現する上では、JVM 間の振る舞いの差異を解決しなければならない。

4.4 パラメータを変化させた場合

サーバを 1 スレッドに固定し、クライアントスレッドを 1 から 50 まで変化させた場合について、測定を行った。測定結果を図 4 に示す。

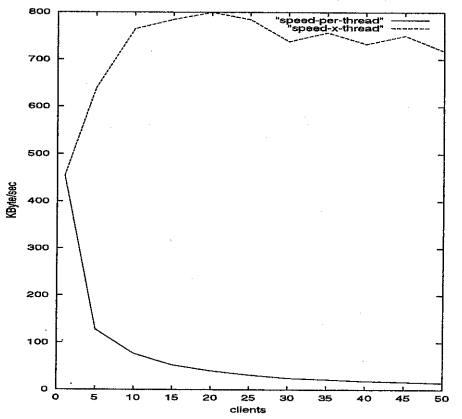

図 4: 1 server - n client 時時の UDP スループットの変化

この結果、送受信を行うスレッドを 20 スレッドまで増やすと、1 スレッド当たりの転送量は当然減少するが、全転送量（スレッド当たりの転送量にスレッド数を掛けたもの）は増加した。これは、確認応答のためにブロックされている時間に別のスレッドが転送を行う事ができる為である。20 スレッド以降、転送量は減少している。これは、実行時間が飽和している為である。スレッドの増加に対して、転送量の減少の幅は 1 割強程度である。これは、スレッド切り替えのコストが全体の 1%強程度である事を示している。このように、Java で記述されたプロトコルスタックは、単体で動作するだけではなく、OS 内で多数のスレッドと共存した場合にも十分な動作が可能であり、十分なスケーラビリティを持っている事が分かった。

4.4.1 ping レーテンシの傾向

また、ping レーテンシ測定 (測定環境 : Sun JRE 1.2.2 for Windows) には、一定のパターンが認められた。全体の 98.86%のパケットにおいて 1msec. 以内の遅延であり、0.6%が 50msec. の遅延、0.53%が 60msec. の遅延であった。このように、全体の 1.13%のパケットにおいて、99%弱のパケットに比して 50 倍から 60 倍の遅延が認められた。この遅延は、図 5 に示したような特定のパターンを示している。

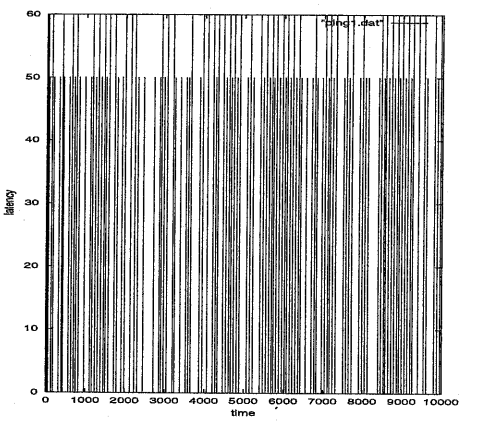

図 5: 時系列で見た ping レーテンシ

この原因を 2 つ考える事ができる。まず、ガベージコレクションの影響が大きく考えられる。そこ

で、プロトコルスタック実行時にJavaプログラムのプロファイルを取得し、内部状態を計測した。ガベージコレクションのコストは、トータルで55回、1625msec.程度であった。しかし、これだけでは先の遅延を説明するのに十分ではない。次に、スレッド間の同期コストを見てみる。waitを実行するコストは、平均すると1回あたり1.15msec.である。しかし、殆どのpingは1msec.以内に返信が返ってきている。これは、一部のwait実行に長時間費やしているという事を示している。また、notifyは全部あわせて1回の呼び出しにつき平均0.25msec.程度のコストである。このように、ガベージコレクションや同期コストによる遅延は、はっきりとした規則性を持っており、それはバッファ管理と同期コストに由来することが判明した。

5 まとめ

本研究では、高い移植性、安全性、設定可能性を持つシステムモジュールを実現するための研究の一環として、JavaによるTCP/IPプロトコルスタックの設計と実装を行った。Javaは、ポインタキャストを持たないため、どれだけ高速なバッファ操作を実現することができるかについて、各種実装を比較することで実験と評価を行った。また、スレッドとモニタによるマルチスレッドサポートといったJavaの特色を生かしたTCP/IPプロトコルスタックの設計を行った。

その結果、実装されたプロトコルスタックの性能を評価し、JRE1.2.2上でpingレーテンシ0.62msec.、UDPスループット521KB/sec.といった測定結果を得た。以上の結果により、Javaで記述されたプロトコルスタックは、実用可能性を持つパフォーマンスを出す事が可能である。

本研究では、実用に耐えうるUDP/IPプロトコルスタックの実装を行うことで、Javaを用いたシステムモジュールの十分な実現性を示した。

参考文献

[1] Douglas Comer, David Stevens 著 , 村井純, 楠本博之 訳, TCP/IPによるネットワーク構築 vol.II - 設計・実装・内部構造 - , 共立出版, 1995

[2] Bobby Krupczak "Protocol Subsystem Support for Efficient and Flexible Communication Service", IEEE INFOCOM '98

[3] Scott Oaks, Henry Wong 著, 戸松豊和 監訳, 西村 利浩 訳, JAVA スレッドプログラミング, オライリージャパン, 1997

[4] Rob Gordon 著, 林秀幸訳, JNI Java Native Interface プログラミング, プレンティスホール出版, 1998

[5] Chris Laffra 著, 株式会社ロングテール, 長尾高弘訳, advanced JAVA, アスキー出版局, 1997

[6] Brian W. Kernighan, Dennis M. Ritchie 著, 石田晴久訳, プログラミング言語 C 第二版, 共立出版, 1989