ネットワークモジュールの自動配布による通信の高速化

光来 健一 †　千葉 滋 ‡　益田 隆司 †

† 東京大学大学院 理学系研究科 情報科学専攻
‡ 筑波大学 電子・情報工学系,　科学技術振興事業団 さきがけ研究 21

要旨

ネットワークが高速になるに従って、データ転送そのものよりもネットワークプロトコルの処理の方がネットワーク通信のボトルネックになってきている。プロトコル処理のオーバヘッドをできるだけ減らすためには、ネットワーク構成などの実行環境に合わせて専用のプロトコルを使えるようにするのが望ましい。そこで我々は、実行環境に合わせてネットワークモジュールを自動的に配布することによって、通信を高速化できるようにする機構を提案する。我々はこの機構を NetBSD に実装し、さらに、標準でないプロトコルを使う時にプロトコル番号の割り当てを動的に解決するためのメタプロトコルを開発した。そして実行環境に合わせた専用プロトコルを用いることによって、ネットワーク通信を高速化できる可能性が十分にあることを実験によって示した。

High-Performance Communication with Automatic Distribution of Network Modules

Kenichi Kourai†　Shigeru Chiba‡　Takashi Masuda†

†Department of Information Science, Graduate School of Science, University of Tokyo
‡Institute of Information Science and Electronics, University of Tsukuba,
PREST, Japan Science Technology Corp.

Abstract

The processing of network protocols becomes a major bottleneck of network communication rather than data transfer itself because of high-speed networks. To reduce the overheads of protocol processing, it is desirable to use protocols optimized depending on execution environments like network constitutions. Therefore we propose a mechanism that enables high-performance communication with automatic distribution of network modules of operating systems depending on execution environments. We have implemented this mechanism in the NetBSD operating system and then developed a meta protocol that dynamically resolves the assignment problem of protocol numbers when non-standard protocols are used. Moreover we confirmed that optimized protocols are possible enough to make network communication more efficient.

1　はじめに

近年、ネットワークがますます高速になり、ギガビットイーサネットも実用化されつつある。このようにネットワークが高速になってくると、データ転送そのものよりも、OS 内部で行われるネットワークプロトコルの処理の方がネットワーク通信のボトルネックになってくる。

このため、ネットワークプロトコルを高速化する研究が盛んに行われている [6, 7, 5]。しかしながら、既に標準となっている汎用プロトコルはその振舞いを変更することはできないので、高速化にはおのずと限界がある。そこで、新たに高速な通信を行うことのできるプロトコルも提案されているが、ネットワークプロトコルは通信し合うマシン全ての OS に実装されなければならないので、標準でないプロトコルは特定の環境以外で使用するのは難しいという問題がある。

そこで我々は、ネットワーク構成などの実行環境に合わせてネットワークモジュールを自動的に配布し、高速な通信を行うことができる専用プロトコルを使えるようにする機構を提案する。OS が自動的にネットワークモジュールを配布することにより、ユーザに意識させずに、柔軟に最適なプロトコルを使うことが可能になる。また、新しいネットワークプロトコルを使う際に問題となる、プロトコルを一意に識別するための番号 (プロトコル番号) の割り当ては、通信相手との間で動的に解決する。

我々は必要に応じてネットワークモジュールを通信相手に配布する機構を NetBSD に実装した。さらに、標準でないプロトコルに対してプロトコル番号を動的に解決するために、DPNAP(Dynamic Protocol Number Agreement Protocol) と呼ばれるメタプロトコルを開発した。この DPNAP を用いて、プロトコル番号を静的に決めるのが難しい、新しく作られたネットワークプロトコルを使って自由に通信することを可能にしている。

一例として、通信し合う二台のマシンが同一のイーサネットセグメントに接続されているという状況を考え、この実行環境に最適化された専用プロトコルを実装して、どの程度の性能向上が可能かを調べる実験を行った。この専用プロトコルと汎用プロトコルである UDP/IP と TCP/IP についてレイテンシを測定した結果、ネットワークの転送速度が 100Mbps の時、この専用プロトコルを使った場合には UDP/IP に対して 16%、TCP/IP に対して 38%の性能向上が見られた。

以下、2 章では研究の背景と我々の提案するネットワークモジュールの自動配布について述べ、3 章ではその実装について説明する。4 章では基礎的なデータを取るために行った実験について述べる。そして 5 章で関連研究に触れ、6 章で本稿をまとめて今後の課題を述べる。

2　ネットワークモジュールの自動配布

2.1　背景

ネットワークが高速になるにつれて、データの転送そのものよりも、OS 内部でのネットワークプロトコルの処理の方がボトルネックになってきている。現在、100Mbps の転送速度をもったイーサネットが標準になっているが、近い将来、1Gbps の転送速度をもつギガビットイーサネットが主流になると考えられる。もちろん、CPU やメモリもそれに合わせて高速化してはいるが、メモリやバスの速度がボトルネックになってネットワークの転送速度の向上に追いつくのは難しい。

そこで、ネットワークプロトコルの処理を軽減する研究が盛んに行われている。従来の UNIX では、送信時にユーザバッファからカーネル空間へ、受信時にカーネル空間からユーザバッファへ、最低一回ずつのコピーが必要とされるが、多くのシステムでこれらのコピーをゼロにすることに成功している [3, 2]。また、実行環境に合わせてプロトコルスタックの一部を実行しないようにしたりすることによって、実行時のオーバヘッドを減らすという研究もなされている [6, 7]。その他にも論文 [5] などで、コードの配置を工夫することでキャッシュのヒット率を上げ、プロトコル処理にかかる時間を短くできることが報告されている。

しかし、TCP/IP のような汎用プロトコルを使っていては、性能を改善するのには限界がある。例えば、TCP/IP は信頼性の低いネットワークであるインターネットを使う場合に、できるだけ効率のよい通信ができるように設計されている。そのため、より信頼性の高いネットワークで構成されるイントラネットの内部からしかアクセスされないウェブサーバとの通信に TCP/IP を使っていては、ネットワークの性能を十分に引き出すのは難しい。しかしながら、現実には実行環境を意識せずにアプリケーションを作成するために汎用プロトコルが使われていることが多い。

2.2 ネットワークモジュールの自動配布

ネットワーク通信を高速化するためには、実行環境に合わせて最適なプロトコルを使えるようにするのが望ましい。実行環境としては、例えば、通信し合うマシンが同一のイーサネットセグメント上にあるかどうか、パケットのデータサイズや送受信の頻度、アプリケーションの種類などが考えられる。また、グループ通信においてはグループに属しているマシンの数なども重要な要素になる。しかし、ネットワークプロトコルは通信し合う全てのマシンの OS に実装されていなければならないため、様々な専用プロトコルを自由に使えるようにするのは難しい。

そこで我々は、実行環境に合わせてネットワークモジュールを自動的に配布し、通信を高速化する機構を提案する。ネットワークモジュールはプロトコルを実装したプログラムであり、実行時に必要に応じて配布することにより、柔軟に最適なプロトコルを使うことが可能になる。さらに、あらかじめ通信し合うマシン全部がネットワークモジュールをもっていなければ通信できないという従来の問題も解決することができる。また、必要に応じて OS が自動的にネットワークモジュールを配布するので、ユーザに意識させることなく通信を高速化することが可能になる。これにより、あらかじめユーザが FTP などでネットワークモジュールを取ってきておく必要がなくなる。

我々の提案する自動配布機構を用いて、新しいネットワークモジュールを配布し、自由に使えるようにするために、プロトコル番号は通信する相手との間で動的に解決する。TCP などの標準のプロトコルに対しては、プロトコルを一意に識別するためのプロトコル番号は決められているので問題にはならないが、標準でないプロトコルに対してプロトコル番号を固定にするのは現実的ではない。もし全く違うプロトコルに対して同じプロトコル番号を割り当ててしまうと、正しく通信できないだけでなく、相互に通信の邪魔をしてしまう可能性もある。また事前に相手の存在を知らず、必要に応じて一時的に作られるネットワーク (アドホックネットワーク) での通信を考えると、プロトコル番号を割り当てるためのサーバを用意するのも適当ではない。

ネットワークモジュールの自動配布機構は様々なアプリケーションの基盤技術として使用することができる。例えば、エージェントシステムにおいてエージェントが移動する際に、移動先との間のネットワーク構成に応じて最適なプロトコルを選択、使用することができる。また、分散システムにおいて新しいネットワークプロトコルをインストールしたり、最新版にアップグレードしたりする際に、全てのマシンに対して行う必要がなくなり、管理のコストを削減することも可能である。

3 実装

我々はネットワークモジュールを自動配布する機構を NetBSD 1.3.2 に実装した。本章ではまず、新しく作られたネットワークプロトコルのようにプロトコル番号を静的に決めるのが難しい場合に、プロトコル番号を動的に解決するために使われるメタプロトコルについて説明する。次に、どのようにネットワークモジュールの自動配布を行うのかについて述べる。

3.1 動的にプロトコル番号を解決するためのメタプロトコル

我々は DPNAP(Dynamic Protocol Number Agreement Protocol) というメタプロトコルを開発し、プロトコル番号が静的に決まっていないネットワークプロトコルを用いて通信することを可能にしている。各マシンの OS は標準でないネットワークプロトコルに対して、その OS の中だけで一意に識別できるプロトコル番号を割り当てるので、プロトコルとプロトコル番号の対応はマシン毎に異なる可能性がある。そこで各マシンの OS は DPNAP を用いて、通信相手の OS が使っているプロトコル番号がどのネットワークプロトコルに対して割り当てられているのかを知るための変換テーブルを作成する。この変換テーブルは通信相手のネットワークアドレスとプロトコル番号の組から、それに対応するプロトコルの受信処理ルーチンへのポインタを得ることを可能にする。この変換テーブルはパケットの受信時にのみ使われ、パケットが到着した時に OS はそのヘッダの情報を基に変換テーブルを参照し、どのプロトコルを使ってパケットが送られてきたかを判別する。

図 1 はパケットが到着した時の様子を示している。ネットワークアドレス A をもった送信側がプロトコル番号 100 のパケットを送った場合を考える。このパケットの受信側では、変換テーブルを参照することによって、ネットワークアドレス A からのプロトコル番号 100 のパケットはプロトコル 1 を使って送られてきたことが分かる。それゆえ、受信側で

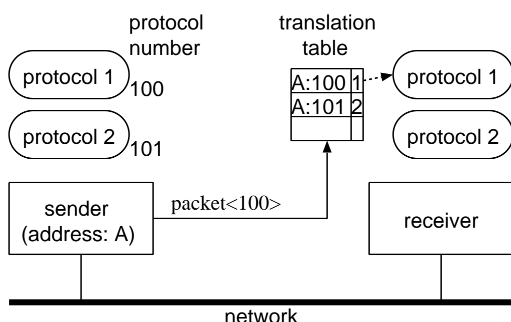

図 1: パケット受信時に変換テーブルを参照してプロトコルを判別する様子

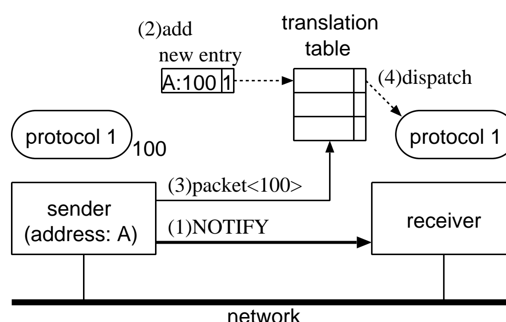

図 2: 最初のパケット送信に先立って送られる *NOTIFY* メッセージ

はプロトコル 1 のパケットの受信処理ルーチンを正しく実行することができる。

通信しようとしている相手にそのプロトコルを使ってパケットを送るのが初めてならば、システムは通信に先立って相手に DPNAP の *NOTIFY* メッセージを送る。この *NOTIFY* メッセージは自分 (送信元) のネットワークアドレスとこれから使おうとしているプロトコルに割り当てられたプロトコル番号、そのプロトコルの名前、バージョンなどから成る。プロトコルの名前とバージョンは、ネットワークモジュールを同定することのできる一意なものでなければならない。このグローバルな名前付けは静的に一意なプロトコル番号を割り当てる場合と同じ問題を内包しているが、プロトコルの内容を反映したできるだけ長い名前をつけることにより、衝突を回避しやすくすることができる。そして *NOTIFY* メッセージを受け取った側では、送られてきたネットワークアドレスとプロトコル番号の組から対応するプロトコルの受信処理ルーチンへのポインタを得られるように変換テーブルにエントリを格納する。送信側が *NOTIFY* メッセージを送った後でパケットを送り、受信側で正しいプロトコルを使って処理される様子を図 2 に示す。

このアルゴリズムを用いれば、たとえ *NOTIFY* メッセージの送信側のマシンが再起動したとしても、通信に混乱が生じることはない。マシンが再起動するとプロトコル番号の割り当て方が再起動する前と変わってしまう可能性があり、再起動前に通信していたマシンの保持している変換テーブルとの整合性が取れなくなるかもしれない。しかし、パケットを最初に送る時にはシステムが再び *NOTIFY* メッセージを通信相手に送る。この *NOTIFY* メッセージを受け取った受信側では、変換テーブルに古いエントリがあればそれを削除して新しいエントリを追加することにより、変換テーブルの整合性を保つことができる。

NOTIFY メッセージの受信側のマシンが再起動した場合には、変換テーブルにエントリが存在しないパケットが送られてくる可能性がある。これは通信を始める際に *NOTIFY* メッセージによって送られてきた情報が失なわれてしまうためである。このようにどのプロトコルを使っているのか不明なパケットが送られてきた時には、システムは DPNAP の *REQUEST* メッセージをそのパケットの送信元に送る。*REQUEST* メッセージは送られてきたパケットのヘッダに格納されていたプロトコル番号などから成る。*REQUEST* メッセージを受け取った送信元では、送られてきたプロトコル番号に対応するプロトコルの名前とバージョンを DPNAP の *REPLY* メッセージとして送り返す。受信側がこの *REPLY* メッセージを受け取ると、*NOTIFY* メッセージを受け取った時と同様に変換テーブルにエントリを追加する。この様子は図 3 に示されている。

受信側が *REPLY* メッセージを受け取るまでの間に到着した、変換テーブルにエントリのないパケットは、変換テーブルが参照できるようになるまで保持しておく。そして、*REPLY* メッセージを受け取って変換テーブルが参照できるようになってから、保持しておいたパケットを処理する。

DPNAP はブロードキャストやマルチキャストを使って通信を行う際にも、一対一通信の場合と同様に使用することができる。途中からネットワークに参加して、通信に先立って *NOTIFY* メッセージを受け取れなかったマシンは、*REQUEST* メッセージをパケットの送信元に送ることによって変換テーブルを作成し、正しく通信することができる。

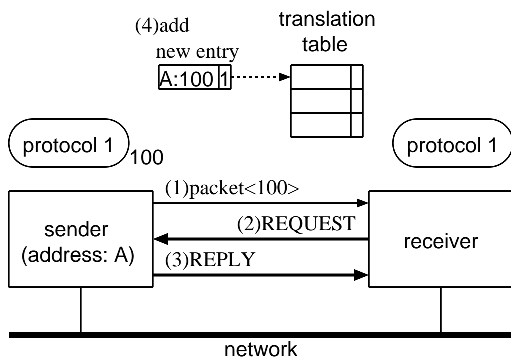

図 3: *REQUEST* メッセージによる変換テーブルへのエントリの追加

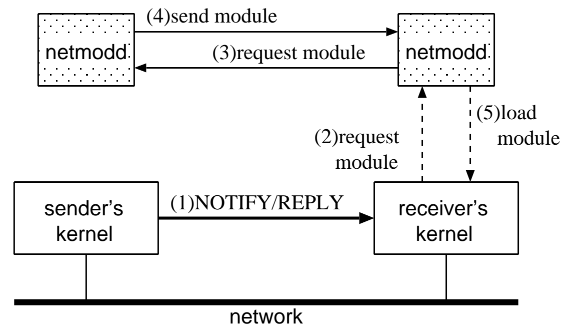

図 4: ネットワークを介したモジュールの自動配布

3.2 ネットワークモジュールの自動配布機構

DPNAP の *NOTIFY* メッセージまたは *REPLY* メッセージを受け取った時、システムは変換テーブルにエントリを登録するが、その際に、送られてきたプロトコルの名前とバージョンに対応するネットワークモジュールを検索する。ネットワークモジュールは OS カーネルの一部であり、デバイスドライバより上位のネットワーク層やトランスポート層などのプロトコルを実装したプログラムである。システムにまだ該当するネットワークモジュールがロードされていなければ、プロセス間通信 (IPC) の機構を使ってユーザレベル・デーモン netmodd にモジュールロードの要求を出す。netmodd はまず、ローカルディスク上に指定されたネットワークモジュールが存在するかどうかを調べ、もしあればそのモジュールをシステムにロードする。[1]現在の実装では、モジュールをシステムに動的にロードするために、NetBSD のローダブル・カーネル・モジュール (LKM) の機構を用いている。

指定されたネットワークモジュールがローカルディスク上になければ、netmodd は DPNAP の *NOTIFY* メッセージまたは *REPLY* メッセージの送信元に対してネットワークモジュールを要求する。送信元は要求されたネットワークモジュールを送り返し、netmodd がそれを受け取ると、そのネットワークモジュールをシステムにロードする。この様子を図 4 に示す。この時点で、追加されたプロトコルに対して、その OS の中で一意に識別できるプロトコル番号が割り当てられる。

[1] この機能は Linux の kerneld と同等である。

ネットワークモジュールの配布を完全に自動化するためには、通常 ifconfig を使って行われるネットワークアドレスの設定も、モジュールがシステムにロードされた時点で自動的に行う必要がある。どのようにネットワークアドレスを設定するかが問題となるが、現在の実装では、ネットワーク・インタフェース・カードに一意に割り当てられているイーサネットアドレス、または、既にシステムに設定されている IP アドレスを使用することが可能になっている。

一方、アプリケーションが socket システムコールを発行した段階で必要なネットワークモジュールがシステムに存在しない場合には、そのアプリケーションは停止させられる。この状態で DPNAP の *NOTIFY* メッセージが送られてくると、netmodd によってネットワークモジュールがロードされ、アプリケーションの実行は再開される。ただし、*NOTIFY* メッセージが送られてくるのは最初のパケットが送られてくる直前なので、netmodd がネットワークモジュールをロードし、ソケットのバインドが完了するまでの間に送られてきたパケットは処理できず、そのままでは破棄されてしまう。そこで、ソケットのバインドが完了するまでに送られてきたパケットは保持しておき、バインドが完了してから処理する。これにより、ネットワークモジュールがあらかじめシステムに存在した時とほぼ同等の振舞いをさせることができる。

3.3 ソケットインタフェースの変更

従来の socket システムコールのインタフェースは、引数にプロトコル番号を指定しなければならず、ネットワークモジュールが必要に応じて配布されて、その時点でプロトコル番号が決まるような状況には

層	NEP	UDP/IP	TCP/IP
システムコールトラップ	1.1	1.1	1.1
ソケット (送信)	4.4	4.5	4.4
プロトコル (送信)	1.6	4.9	6.2
イーサネット (送信)	10.4	13.8	15.9
送信合計	17.5	24.3	27.6
イーサネット (受信)	17.9	18.7	18.6
割り込み待ち	1.6	1.6	1.6
プロトコル (受信)	3.4	7.0	30.3*
スケジューリング待ち	5.4	5.4	5.4
ソケット (受信)	4.3	4.9	4.9
システムコール終了処理	1.9	1.9	1.9
受信合計	34.5	39.5	62.7
ネットワーク (10Mbps)	68.5	68.7	103.2
ネットワーク (100Mbps)	10.8	11.4	11.4
合計 (10Mbps)	120.5	132.5	193.5
合計 (100Mbps)	62.8	75.2	101.7

表 1: NEP, UDP/IP, TCP/IP のレイテンシとその内訳 (μs) (*ACK の送出を含む)

適していない。そこで新しく作ったプロトコルの場合には、プロトコル番号の代わりにプロトコルの名前とバージョンを指定するようにする。その後で行われるソケットのバインド等には、ネットワークモジュールがロードされた時に割り当てられたプロトコル番号を使用する。[2]

4 実験

同一のイーサネットセグメントに接続された二台のマシンが通信する場合に特化した専用プロトコル NEP を実装し、このプロトコルを用いて基礎的な実験を行った。NEP はイーサネットドライバに直接パケットを出し、イーサネットドライバから直接パケットを受けとる、データリンク層の上に直接作られたプロトコルである。イーサネットドライバの制限を越える大きなパケットを扱うことができない[3]という点を除いては、UDP と同様、ポートによるパケットの多重化やチェックサムの計算などを行う。ただし、そのパケットはイーサネットセグメントを越えることはできず、TCP のような信頼性の保証もしない。

この実験は 10Mbps または 100Mbps のイーサネットで接続された二台の PC (Pentium II/400MHz, メモリ 128MB) を使って行った。ネットワーク・インタフェース・カードには 3COM Fast EtherLink を用い、OS には我々が手を加えた NetBSD 1.3.2 を使用した。

4.1 専用プロトコルの使用による性能向上

特定の実行環境での使用を目的として作成した専用プロトコル NEP が、UDP/IP や TCP/IP を使った場合に比べてどの程度ネットワーク通信を高速化できるかを調べる実験を行った。この実験では、1 バイトのデータを送った時の end-to-end のレイテンシを測定し、プロトコル処理のどの層がどの程度高速化できているかを調べた。

この実験の結果は表 1 のようになった。この実験結果から、ネットワークの転送速度が 100Mbps の時には、NEP は UDP/IP に対して 16%の性能向上、TCP/IP に対しては 38%の性能向上を達成できていることが分かる。転送速度が 10Mbps の時でも、UDP/IP に対して 9%の性能向上、TCP/IP に対しては 38%の性能向上を達成できている。TCP/IP に関して 10Mbps の場合に 100Mbps の場合と同等の性能向上できている理由は、ACK を送信するために 10Mbps の場合にはネットワークが飽和状態にな

[2] このプロトコル番号は追加したシステムコール getprotonum を使って得られる。
[3] 1500 バイトまで。

り、実際には 10Mbps の転送速度が出ていないためと考えられる。

表 1 の内、プロトコル層だけを見てみると、NEP の送信処理にかかる時間は UDP/IP の 33%、TCP/IP の 26%にまで削減できている。また、NEP の受信処理にかかる時間は UDP/IP の 49%、TCP/IP の 11%で済んでいる。

この実験のレイテンシの測定は繰り返しパケットを送って平均を取ったものなので、命令キャッシュやデータキャッシュが非常に良く効いている状態になっている。そのため表 1 ではイーサネットドライバの処理、つまり、ネットワーク・インタフェース・カードに対する I/O が大きなボトルネックになっているように見える。しかし、実際のアプリケーションではネットワーク通信以外にも様々な処理が行われる上、送信するデータの内容も変化するので、これほどキャッシュが効くことはまずありえない。それゆえプロトコル処理に占めるソフトウェアの部分がさらに大きくなり、専用プロトコルの NEP と汎用プロトコルの UDP/IP、TCP/IP の性能差はさらに開くと考えられる。

これらの実験データおよび考察より、実行環境に合わせて専用プロトコルを使うことにより、プロトコル処理のオーバヘッドを減らす余地は十分にあると思われる。

また、ネットワークの転送速度が 100Mbps の場合には 10Mbps の場合と比べて、レイテンシ全体に占めるデータ転送の割合は 57%から 17%へと減少している。ネットワークの転送速度がさらに速くなれば、データ転送がレイテンシ全体に占める割合はますます小さくなり、プロトコル処理がますますボトルネックになってくるということを裏付ける結果となっている。

4.2 動的なプロトコル番号解決によるオーバヘッド

DPNAP を使って動的にプロトコル番号を解決するようにすると、パケットの送信時と受信時に余分な処理が必要になり、その分のオーバヘッドがかかる。具体的には、パケットの送信時に DPNAP の *NOTIFY* メッセージをすでに送っているかどうかのチェックが必要になる。またパケットの受信時には、送られてきたパケットがどのプロトコルを使って送られてきたのかを調べるために変換テーブルを参照する必要がある。これらのオーバヘッドがどの程度であるか調べるために、静的に固定のプロトコル番号を割り当てた NEP と動的にプロトコル番号

解決の仕方	10Mbps	100Mbps
静的	120.5	62.8
動的	121.2	63.9

表 2: プロトコル番号の解決の仕方による NEP のレイテンシの違い (μs)

を解決した NEP の両方について、1 バイトのデータを送った時の end-to-end のレイテンシを測定した。

この実験の結果は表 2 の通りである。動的にプロトコル番号を解決した NEP は静的にプロトコル番号を割り当てた NEP に比べて、10Mbps で 0.6%、100Mbps で 1.8%のオーバヘッドを被っている。しかしながら 4.1 章で示された性能向上の割合から見ると、このオーバヘッドは十分に小さいので、問題にはならないと思われる。

4.3 ネットワークモジュールの配布に伴なう遅延

必要に応じてネットワークモジュールを配布する場合、ネットワークを介してモジュールを送ってシステムにロードするため、パケットを受け取れるようになるまで遅延が生じる。モジュールのサイズが 4,683 バイトでネットワークの転送速度が 10Mbps の時、この遅延は 250ms に達し、1 バイトのデータを送った時の NEP の end-to-end のレイテンシ 0.1ms に比べて非常に大きくなっている。しかしその内訳を見てみると、ネットワークを介してモジュールを送るのにかかっている時間は 5ms 程度であり、ほとんどの時間はシステムへのモジュールのロードに費されている。このことより、Linux のように必要に応じてモジュールをシステムにロードするシステムでは、ネットワークを介したモジュール配布の影響はほとんどないと言える。

5 関連研究

SPIN [1] や VINO [8] といった拡張可能 OS は、専用プロトコルを拡張モジュールという形で作成し、システムに動的に追加することを可能にしている。拡張可能 OS は拡張モジュールがシステムの他の部分に悪影響を与えないように保護するので、安定な拡張モジュールに関しても一定のオーバヘッドがかかる。そのため、専用プロトコルを使うことによる性能向上を意味のないものにしてしまう可能性があ

る。また、単一のシステムでの拡張についてのみ考えられており、分散システム全体をシームレスに拡張することについては考えられていない。

JavaOS [4] や論文 [9] のシステムなどでは専用プロトコルをアプレットという形でサーバからダウンロードし、システムに組み込むことを可能にしている。この方法の問題点は、必ずアプレットを提供するサーバが存在しなければならず、各マシンは必要なアプレットをダウンロードできるサーバの位置を知っていなければならない点である。さらに、通信を始める前にあらかじめ必要なアプレットをダウンロードしておく必要もある。

6 まとめと今後の課題

実行環境に合わせてネットワークモジュールを自動配布し、ユーザに意識させずに通信を高速化する機構を提案した。また、新しいネットワークプロトコルを使う際のプロトコル番号割り当ての問題を解決するために、DPNAP というメタプロトコルを開発した。そして、ネットワークモジュールを自動配布する機構と DPNAP を NetBSD に実装し、実行環境に合わせて専用プロトコルを使えば、十分に性能が改善され得ることを実験によって示した。

我々のシステムの現在の実装にはいくつかの制限がある。一つは、ネットワークモジュールをシステムにロードする際に自動的に設定するネットワークアドレスに、プロトコル独自のものを割り当てることができない点である。Appletalk のように独自のネットワークアドレスを使うことは十分に考えられるので、自由にネットワークアドレスを割り当てられるようにする必要がある。また、モジュールの自動配布機構や DPNAP が今のところ、イーサネット層の上に直接作られるプロトコルに対してしか実装されていない。しかし、本稿で述べている方法を IP 層の上に作られるプロトコルなどにも適用することは可能であると思われる。

通信相手から配布されたネットワークモジュールは不安定かもしれず、場合によっては悪意を持っていることもあり得る。このような場合に LKM を用いて、安全性を考慮せずにネットワークモジュールをシステムに組み込むことは問題になる。そこで、我々がこれまでに提案してきている多段階保護機構 [10] を発展的に用いて、安全なモジュールの性能を低下させることなく、安全でないモジュールに対しては十分に保護できるようにすることを考えている。

また現在、ネットワークモジュールは実行可能形式で配布されており、同一の計算機アーキテクチャ間だけで可能となっている。しかし現実には様々な計算機アーキテクチャのマシンが使われているので、できるだけ性能を低下させずに、異なるアーキテクチャ間でもネットワークモジュールを配布できるようにする必要がある。

参考文献

[1] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, S. and Eggers, C.: Extensibility, Safety and Performance in the SPIN Operating System, in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284 (1995).

[2] Chu, H. J.: Zero-Copy TCP in Solaris, in *Proceedings of 1996 USENIX Technical Conference*, pp. 253–264 (1996).

[3] Druschel, P. and Peterson, L. L.: Fbufs: A High-Bandwidth Cross-Domain Transfer Facility, in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 189–202 (1993).

[4] Madany, P. W.: *JavaOS: A Standalone Java Environment*, Sun Microsystems (1996).

[5] Mosberger, D., Peterson, L. L., Bridges, P. G. and O'Malley, S.: Analysis of Techniques to Improve Protocol Processing Latency, in *Proceedings of the SIGCOMM'96 Symposium*, pp. 73–84 (1996).

[6] O'Malley, S. W. and Peterson, L. L.: A Dynamic Network Architecture, *ACM Transactions on Computer Systems*, Vol. 10, No. 2, pp. 110–143 (1992).

[7] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J. and Zhang, K.: Optimistic Incremental Specialization: Streamlining a Commercial Operating System, in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 314–324 (1995).

[8] Seltzer, M. I., Endo, Y., Small, C. and Smith, K. A.: Dealing With Disaster: Surviving Misbehaved Kernel Extensions, in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227 (1996).

[9] Tschudin, C.: Flexible Protocol Stacks, in *Proceedings of the SIGCOMM'91 Symposium*, pp. 197–205 (1991).

[10] 光来健一, 千葉滋, 益田隆司：多段階保護機構：拡張可能 OS の新しい Fail-safe 機構, 情報処理学会 論文誌, Vol. 39, No. 11, pp. 3054–3064 (1998).