第 5 部

プロセス間通信

第 1 章

序論

1.1 はじめに

コンピュータネットワーク技術の発展により、分散環境におけるアプリケーションが重要性を増してきている。ローカルエリアネットワークでは多くのアプリケーションに RPC(Remote Procedure Call) [74][73] が使われているが、もともとローカルエリアネットワークで利用されることを意識して作られているので、その機構をそのまま広域の分散環境で用いることには問題が多い。現在のネットワークでは広域分散環境における分散処理が注目を集めているが、広域分散ネットワークの構築には適切なプロセス間通信、特に、適切な RPC の開発が必要不可欠である。

分散環境では資源の共有が可能で、NFS(Network File System)[75] などにより、ユーザは遠隔資源をあたかもローカルの資源であるかのように利用できる。また、ローカルにはないサービスでもネットワークを介して受けることができる。しかし、ネットワークが広域になると、その状況によって遠隔資源にアクセスできなかったり、遠隔のサービスを受けることができなくなってしまうことがある。遠隔ということだけでも転送時間などに関して不利なことは明らかである。様々なネットワークの接続でコンピュータの種類や オペレーティングシステムの種類も多様になり、その複雑さははかり知れない。

プロセス間通信の手段として広域分散環境で RPC を利用するには、より効率の良い方式を採る必要がある。どこにいるどんな相手に対しても順応できる NFS を構築するためには、RPC の最適利用を支援する機構を作らなければならないのである。さらに、ユーザの様々な要求を取り入れられるような RPC が提供されるようになれば、有用なアプリケーションが数多く作られ、ネットワーク環境がより良いものになっていくであろう。

1.2 本研究の目的

現在、ローカルエリアネットワークにおいては、NFS を利用することでユーザ環境が整えられている。しかし、ethernet のような高速の回線でなければこのようなサービスはうまく機能していない。実際、9.6Kbps のような低速の回線を使用したネットワーク上では NFS は利用されていない。また、広域ネットワークでは、RPC を用いたアプリケーションが、高速用に設定されているタイムアウトによって通信できない場合がある。

本研究では、これらを改善していくために、RPC としてのデータ通信を効率良く行なうことを考える。現在利用されている RPC や ネットワークサービスを考察し、その上で、ネットワークの状態によるデータサイズと応答時間の関係を解析し、RPC の最適利用のための支援システムを設計する。また、NFS を利用する際に指定するデータサイズやタイムアウトなどのパラメータは、管理者によって経験的に設定されているが、状況に応じてその値を自動的に探るネットワーク診断システムを提案する。RPC を利用する際にこの機構を用いることで、より快適な環境が構築できるのである。

1.3 本論文の構成

この章では目的などについて述べたが、以下、本論文では、第 2 章で、コンピュータネットワークと RPC について述べ、特に、Sun RPC と NFS について解析する。第 3 章で、現在 RPC の基盤となっているデータグラム通信について考察し、第 4 章で、実際のネットワークにおいて計測を行なう。第 5 章では、第 4 章の計測結果をもとに RPC の最適利用のために提供されるべき情報を整理する。第 6 章で、プロトタイプの試作を試み、今後のシステムの設計への足掛かりとする。

第 2 章

コンピュータネットワークと RPC

2.1 広域分散ネットワーク

遠隔地に分散する計算機資源を共有する目的で広域ネットワークが広く構築されている。近距離の分散資源を共有するローカルエリアネットワーク同士を結合し、新たに大規模なネットワークを構築するネットワーク間接続という技術がその基礎を支えているのである [15]。

ローカルネットワークと広域ネットワークを比較すると、Table2.1 のようになる。

	ローカルエリアネットワーク	広域ネットワーク
地域性	構内に閉じている。 数 km 程度の範囲。	広域。 国際的規模に及ぶものもある。
公衆性	一企業、一大学などプライベート。	通信業者などによって構築され、 その対象となるユーザは不特定多数。
多様性	高速な単一の通信技術を利用。	性質・能力の異なる多種の通信技術 が混在する。
通信速度	数十 Mbps 程度。	数十 Kbps 以下が主体。

表 2.1: ローカルネットワークと広域ネットワークの比較

ネットワークの広域化による利点は遠隔資源を共有したり、多くの人々の間でのコミュニケーションを可能としたり、あるいは、いくつかのシステムの結合で負荷の分散ができることである。現在、広域のネットワークを利用して電子メールやニュースシステムによる情報交換が活発に行なわれたり、遠隔資源や遠隔サービスを利用したアプリケーションも考案されてきている。更に、マルチメディアを扱うネットワークコミュニケーションの研究も行なわれている。

しかしながら、広域性や多様性のために生じる問題も多い。ユーザに資源の位置を意識させないために、遠隔資源をアクセスする場合にも、ローカルエリアネットワークの資源に対するアクセスと同様に高速に行なわれなければならない。更に、広域分散環境においても、ユーザに対してローカルエリアネットワークと同じサービスが提供され、ネットワークを意識せずに利用できる環境が構築されることが望まれる。

2.2 広域分散環境における RPC

2.2.1 クライアント／サーバ モデル

ネットワークを利用したサービスはプロセス間通信を用いて実現されている。このプロセス間通信を行なうシステムは、クライアント／サーバモデルを用ている。クライアントはサービスの消費者で、サーバは提供者である。クライアントはサーバにパラメータを渡してサービスを要求し、サーバは適切な処理をして応答する。rsh、rwho、phone などのコマンドや RPC もこのモデルに従っている。

2.2.2 RPC モデル

RPC のモデルはローカルのプロシージャコールのモデルに似ていて、呼び出し側のプロセスはあたかも自分の中のプロシージャを呼び出しているかのように、別のプロセスのプロシージャに実行を依頼できる。これらのプロセスは独立しているため、もはや、物理的に同じ計算機上に共存している必要はないのである。ただし、RPC の場合はネットワークを介してデータの交換を行わなければならないので、パラメータと結果の受渡しに特別な配慮が必要である。

RPC におけるクライアント／サーバモデルは、次のように考えることができる。

サービス
: 一つ以上のリモートプログラム (リモートプロシージャの集まり) を集めたもの。

サーバ
: マシンにはいくつかのネットワークサービスが実装されている。休止状態のサーバプロセスは、コールメッセージが到着すると、プロシージャのパラメータを取り出して結果を計算し、リプライメッセージをクライアントに送り返す。

クライアント
: サービスに対するリモートプロシージャを起動させるソフトウェアである。プロシージャへのパラメータを含んだコールメッセージをサーバプロセスに送り、プロシージャからの結果の含まれるリプライメッセージの到着を待つ。

ユーザは、RPC の簡単な仕様を知れば、ソケットなどのプロセス間通信の詳細 ([70][71] 参照) を知らなくてもネットワークアプリケーションを書くことができるのである。ユーザはサーバプロセスとの間でどのように通信が行なわれているかを知る必要はないのである。

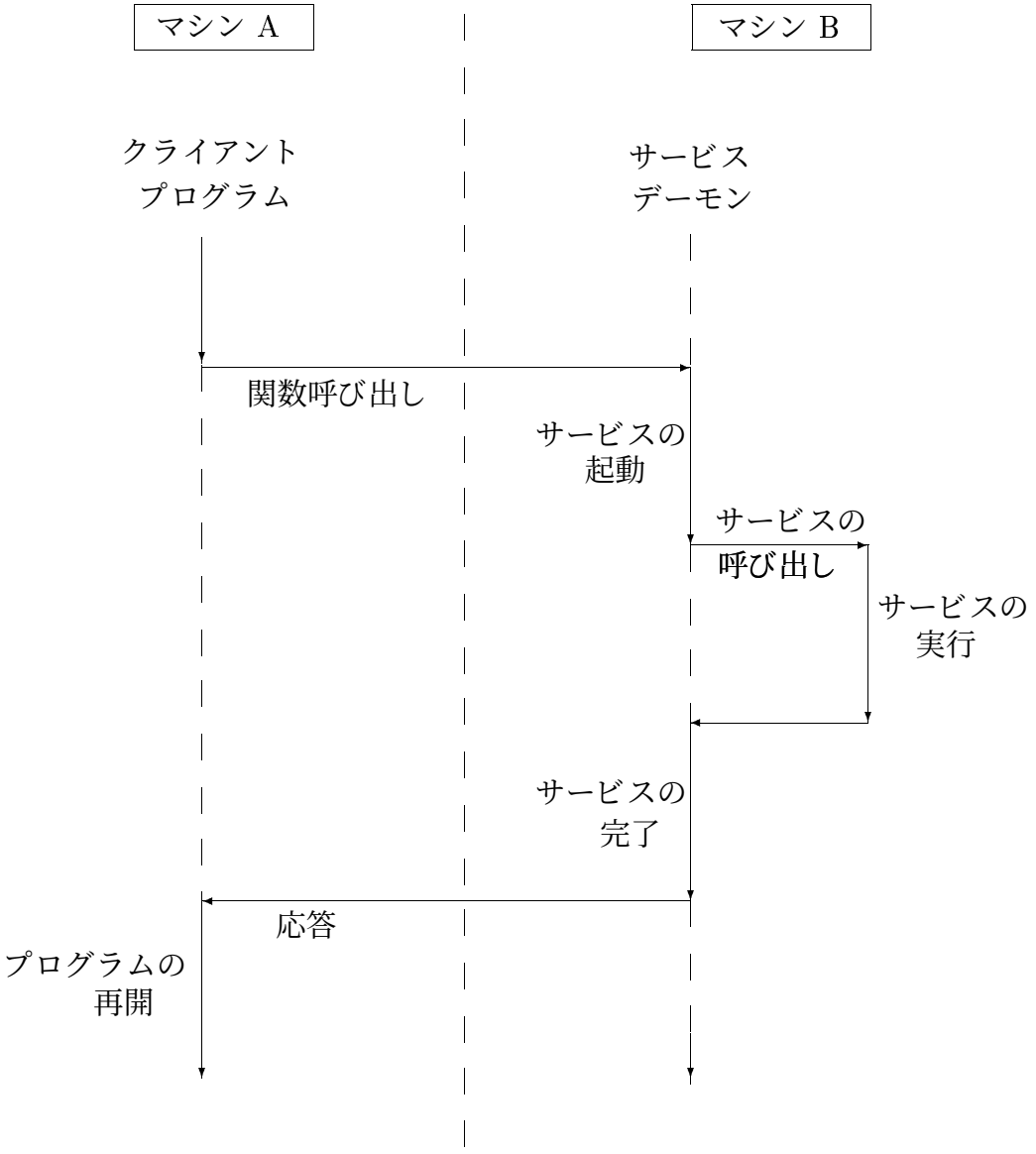


図 2.1: RPC を用いたネットワークコミュニケーション

2.2.3　広域分散環境に適した RPC

ローカルエリアネットワークでは多くのアプリケーションに RPC が使われている。RPC は分散アプリケーションを構築するのに適した手続き型のプロセス間通信であり、広域分散環境における RPC の重要性が増している。多種の通信技術が混在している中で、その詳細を意識せずに通信のできる手続き呼出しの機構が評価されている。しかし、従来の RPC は広域のネットワーク環境に対しては考慮されていないので、広域分散環境に適した RPC が必要になってくる。

広域分散環境では回線が低速であり、プロセス間の通信に比較的長い時間がかかる。従って、広域分散環境においては高速なプロセス間通信が必要になる。他のシステム上にあるプロシージャを、あたかも自分のローカルなプロシージャであるかのように利用できるためには、ユーザに通信時間の長さを意識させてはならない。効率の良い RPC を行なうことで、この高速化を提供しなくてはならない。

また、ネットワークの輻輳の制御はローカルエリアネットワークの場合より複雑である。ネットワークを混雑させるような通信は、広域分散環境においては致命的である。大量データ転送を伴う時や、サーバの負荷が大きい時には、軽率な再送処理は状況を悪化させてしまうので、状況に応じた適切な方法で再送が行なわれなければならない。RPC としても適切な再送制御を行なう必要がある。

更に、ネットワークが広域化する中で、クライアントの要求に対して適切なサーバをどのように選択するかということも考える必要がある [51]。

2.3　Sun RPC

ここでは、既存の RPC として、Sun Microsystem 社の RPC について解析する [73][74]。

クライアントはプロシージャを呼ぶことによって、サーバにデータパケットを送る。パケットが到着するとサーバはサービス処理ルーチンを呼んで要求されたサービスを実行し、リプライパケットを送ってプロシージャコールがクライアントに戻される。

2.3.1　三つのインタフェース

Sun RPC のインタフェースは以下の三つの階層に分けられている。

- highest layer
- intermediate layer

- lowest layer

2.3.1.1 highest layer

このレイヤでは、すでにリモートプロシージャがライブラリとして用意されているので、ローカルなプロシージャコールと同じようにプログラムの中で使用するだけでよい。RPC を意識せずに使うことができ、ユーザに透過性を提供している。現在提供されているライブラリルーチンには、次のようなものがある。

rnusers() ··· リモートマシンのユーザの数
rusers() ··· リモートマシンのユーザ情報
havedisk() ··· リモートマシンがディスクを持っているか
rstat() ··· リモートカーネルのパフォーマンスを得る
rwall() ··· 特定のマシンへの書き込み
getmaster() ··· YP マスタの名前を得る
getrpcport() ··· RPC のポート番号を得る
yppasswd() ··· YP のユーザパスワードの更新

2.3.1.2 intermediate layer

callrpc() と registerrpc() という二つの関数を用いる。callrpc() がクライアント側、registerrpc() がサーバ側の関数である。様々なアプリケーションのために設計されていて、利用に際してソケットなどについて知る必要はない。

クライアントから呼ばれた callrpc() はサーバの registerrpc() に処理を依頼し、結果が戻ってくるまでブロッキングされる。 依頼を受けた registerrpc() は適当なプロシージャを呼んで処理し、結果を callrpc() に返す。

callrpc() は 8 個のパラメータを持つ。リモートマシンの名前でサーバマシンを指定し、プログラム番号、バージョン番号、プロシージャ番号で、プロシージャを指定する。さらに、プロシージャに渡すパラメータのデータ型とそのアドレス、プロシージャから受け取る結果のデータ型とそのアドレスを与える。registerrpc() は使うべきプロシージャを全て登録してから無限ループに入ってサービス要求の到着を待っている。パラメータは６個で、プログラム番号、バージョン番号、プロシージャ番号、及び、そのサービスの処理を行なうプロシージャの名前、プロシージャへのパラメータのデータ型、結果のデータ型を与える。

これらの関数では UDP しかサポートされていないので、8Kbyte 以上のデータ転送は行なえない。また、TCP を使いたい場合は、lowest layer

でプログラムを作成しなくてはならない。

2.3.1.3　lowest layer

このレイヤでは RPC のためのライブラリルーチンが用意されていて、ソケットやネットワークライブラリルーチンを使って RPC を作成する。

まず、サーバ側では、svcudp_create() 又は svctcp_create() を用いて、transport handle を得る。pmap_unset() でポートマッパのテーブルから前のポート番号の登録を消したあとで、svc_register() で指定されたプログラム番号、バージョン番号、プロシージャ番号を再登録する。最後に、svc_run() でサービス要求を待つ。

クライアント側では、gethostbyname() でサーバマシンのネットワークアドレスを得て、そのアドレスをもとにして clntudp_create() 又は clnttcp_create() を用いて、RPC のクライアントを作る。つぎに、clnt_call() を用いてサーバプロセスに処理を依頼し、結果が返ってくるまで待つ。指定したタイムアウト以内に戻らないと失敗になる。clnt_destroy() で、関連した空間をすべて解放するが、使用していた socket は閉じない。これは、同じ socket を複数のクライアントが使っている時のためである。

2.3.2　XDR

クライアントマシンとサーバマシンが異なる種類の計算機の場合、扱うデータが同じでも、異なる値を表わすことがある。これは、計算機によってデータの評価の仕方が違うためで、プロセス間でデータの転送を行なう場合は注意が必要である。

Sun RPC は、異なる種類の計算機間でも使用することができるようになっている。データを送り出す前に XDR(eXternal Data Representation) に換算することで任意のデータ構造が扱えるのである [69]。XDR はポータブルなデータ転送のための仕様である。RPC と共に用いることで、プロセス間通信のための標準入出力ライブラリを提供するので、ソケットベースのプロセス間通信の詳細には触れずにソケットにアクセスすることができるのである。整数や小数はもちろん、配列や共用体、ポインタなどに対していくつかのルーチンが提供されているが、それらを組合わせてユーザが新しいルーチンを作ることもできる。

2.3.3　port mapper program

ポートマッパプログラムは RPC プログラム番号とバージョン番号を UDP/IP や TCP/IP のポート番号にマップする。クライアントプログラ

ムはサーバがサポートしているリモートプログラムのポート番号を知るためにポートマッパを使う。

まず、サーバプログラムはそのホスト上のポートマッパにエントリを登録する。クライアントプログラムはリモートプログラムのポート番号を知りたい時、サーバのポートマッパに RPC コールメッセージを送り尋ねる。サーバマシンにリモートプログラムがサポートされていれば、ポートマッパは対応するポート番号を RPC のリプライメッセージで返す。そして、リモートプログラムのポートに RPC のコールメッセージを送ることが出来るのである。

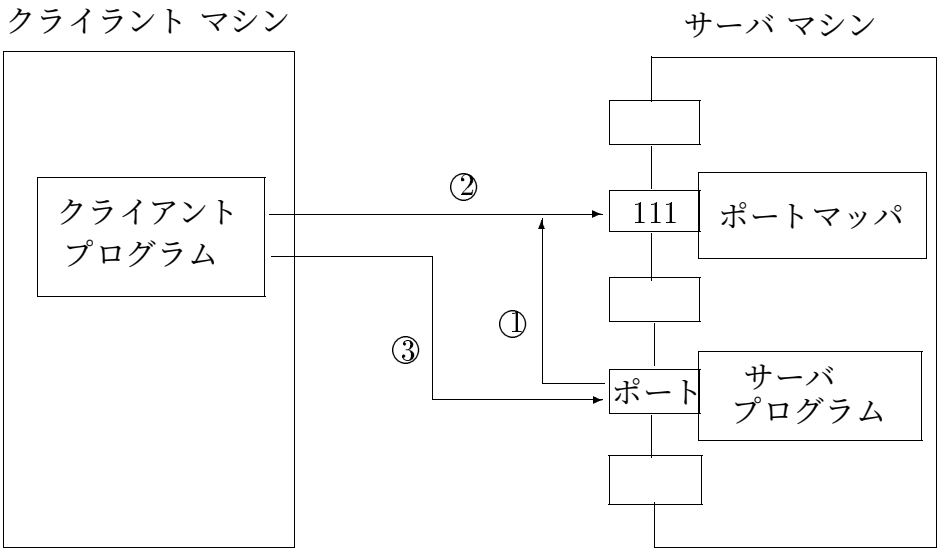
このプログラムにより、リモートプログラムの動的なバインドが可能になる。つまり、あるネットワークサービスがサーバマシンによって違うポートを持つことが出来るのである。

2.3.4 Sun RPC の評価

RPC のインタフェースは三つのレイヤで提供されていた。highest layer では、非常に簡単にプログラムを作ることができたが、そのためのライブラリルーチンとして提供されているものは少なく、物足りなさを感じる。intermediate layer では、タイムアウトを扱うことができないので、RPC を用いて有効に機能するネットワークアプリケーションを作成するためには、lowest layer を使わなくてはならない。プロセス間通信の詳細を知らなくてもネットワークアプリケーションプログラムが書けるという RPC であるが、有用に RPC を利用するためには、やはりソケットやネットワークライブラリルーチンを使わなくてはならないのである。従って、上位のレイヤに RPC のライブラリルーチンが作成され、様々な要求に対応できるようになる必要がある。

しかし、ある程度プロセス間通信のプログラムを扱ったことのあるユーザには、lowest layer が十分なプリミティブを提供している。プロトコルとして、UDP/IP と TCP/IP の両方がサポートされている上、ブロードキャストやバッチ処理の RPC も用意されている。現在、あまり使われていないが、認証確認のフィールドも用意されており、セキュリティが問題となる広域分散環境では有用である。また、RPC はクライアントとサーバが同等の立場にあり、サーバがクライアントになることも可能であって、プロシージャ同士の通信も行なうことができると考えられる。

Sun RPC の特徴である、ポートマッパの機能も有用である。これは、クライアントの問い合わせに対してサーバの情報を提供するものであるが、このポートマッパにより、ポート番号を意識せずに、プログラムを書くことができ、システムごとの異なる設定も可能となる。また、Sun RPC は



1 サーバはポートマッパに登録をする
2 クライアントはポートマッパからサーバのポートを得る
3 クライアントはサーバにコールメッセージを送る

図 2.2: ポートマッピング

XDR の上に構築されているので、異なる種類の計算機間で利用できるということも、特徴として挙げられる。

Sun RPC は lowest layer を用いることで、拡張性のあるパッケージであるといえるが、ネットワークアプリケーションの一部として RPC を利用する際に、その拡張をプログラマに望むのではなく、RPC として様々な機能を提供するべきである。現在、広域ネットワークでの利用についてはあまり考慮されておらず、確認応答など、他の RPC の機構に比べて劣っている [83]。Sun RPC は、再送に関して充分な機能が提供されていないようであるが、確認応答の機構によっては再送の制御の方法も変わってくることになる。

さらに、多くの RPC はその性質から同期型の機構で実現されているが、Athenaプロジェクトでは非同期型の RPC が設計されている [64]。今後の広域分散環境におけるネットワークアプリケーションを考えると、非同期型の RPC が必要になる場合も数多くある。Sun RPC では、そのようなアプリケーションを有効に作成ことはできないのである。

2.4 NFS と Sun RPC

2.4.1 ネットワークサービス

基本的なネットワークサービスのほとんどはクライアント／サーバモデルに従っている。Network File System(NFS) や Yellow Page(YP) Database Service[68], Network Lock Manager, Status Monitor などがそうである。Sun OS では RPC の機構を基にこれらのネットワークサービスを実装しているところに特徴がある [72]。RPC と XDR の提供するサービスを基礎とするので、これらのネットワークサービスはネットワークアプリケーションということになり、XDR/RPC/NFS という環境は、本質的に、更に拡大していく可能性のあるものであることがわかる。つまり、新たなネットワークサービスを容易に付加することが可能である。

NFS はオペレーティングシステムとは独立したサービスである。これによりユーザはネットワークを介してファイルシステムの一部をマウントすることができて、それらがローカルなものであるかのように扱うことができる。Sun OS 4.0 ではユーザやホストの認証確認を行なうセキュアなマウントも加えられている。

YP は NFS の促進する大規模ネットワークの管理を容易にするために設計されたネットワークサービスで、読み出しのみのデータベースである。パスワードやグループ、ネットワーク、ホストなどの情報提供に使われている。

Network Lock Manager は一つのファイルを同時に更新することを避けるためにファイルにロックをかける。これを実装するシステムコールは同一計算機上の別のプロセスを除外するだけでであるが、カーネルが RPC を用いてロックマネージャにシステムコールを送るので、複数の計算機上で同一のファイルシステムを共有できるのである。このロックマネージャは stateful であるので、サーバやクライアントのクラッシュに対処することができる。

Network Status Monitor はネットワークアプリケーションが計算機のリブートを検出したり、アプリケーションに固有な回復の機構を起動させることを可能にする。

2.4.2　NFS

NFS は異種類の計算機やオペレーティングシステム、ネットワークが接続される環境で、ファイルを共有するための機能である。オペレーティングシステムの種類に対する透過性は RPC の機構を用いているためであり、計算機の種類に対する透過性は XDR の機構を用いているためである。NFS は分散オペレーティングシステムとしてではなく、ネットワークサービスとして設計されており、唯一のオペレーティングシステムに限定することなくネットワークサービスに対する分散アプリケーションをサポートすることができるのである。さらに、NFS は必要なディスクを全て利用できるようにすることに目標をおいている。ネットワーク上にあればすべての情報にアクセスすることができ、ネットワーク上のプリンタやスーパコンピュータも利用可能になるのである [75]。

2.4.2.1　リモートファイルシステムの mount

あるファイルを参照したいが、自分の計算機にはそれが存在していなくて、他の計算機上には存在するような時、 mount コマンドによりそのファイルを参照することができる。これはファイルシステムを含むディレクトリ階層であっても可能である。つまり、他の計算機上のファイルシステムの部分木を自分の計算機のファイルシステムに継ぎ足すことができる。

2.4.2.2　データアクセスにおける透過性

ユーザはデータのネットワークアドレスを知らなくても望みのファイルを扱うことができる。ユーザには NFS でマウントされたファイルシステムは自分の計算機にあるディスクと全く同じように扱うことができる。ローカルディスク上のファイルへの読み書きと、NFS でマウントされたディス

ク上のファイルへの読み書きには違いは全くないのである。しかし、実際はネットワーク上の情報は分散しているのである。

また、必要な作業のためのツールのすべてを一ヶ所で供給することは不可能で、各サービスは一つのネットワークの中で統合されなければならない。このような統合のために、NFS はオペレーティングシステムに依存しない柔軟性を持っているのである。

2.4.2.3 NFS の実装

UNIX のファイルシステムは i ノードに対応するディレクトリやファイルから構成されている。i ノードはファイルの位置、大きさ、アクセスの許可、アクセス時間などのファイル管理情報を含んでいる。この i ノードは一つのファイルシステムの中では唯一の番号を持つが、他のファイルシステムの i ノードと同じ番号を持つことになる。従って、NFS でマウントした場合に i ノードの一貫性が失われる。この問題に対して Sun は v ノードと呼ばれるデータ構造に基づく VFS(Virtual File System) を設計した。VFS ではネットワーク上においてもファイルが唯一に決定できる。

この VFS は RPC や XDR の機構を基に NFS インタフェースを定義し、実装している。ローカルな VFS の場合には、直接クライアントマシンに接続されたディスク上のファイルシステムデータにアクセスされるが、リモートな VFS の場合には、アクセス要求はネットワーク上の RPC や XDR のレイヤを通過することになる。現在は、UDP/IP プロトコルを使って実装されている。そして、サーバ側では RPC と XDR のレイヤを通過して NFS サーバに要求が到着するとローカルな VFS でアクセスするために v ノードを使いサービスに答えるのである (Figure2.3参照)。

NFS の実装は次のような５つの透過性を提供している。

- ファイルシステムの種類の透過性
  vnode により、様々な種類のファイルシステムを扱える。

- ファイルシステムの位置の透過性
  ローカルな VFS とリモートな VFS に違いはないのでファイルシステムデータの位置は透過的である。

- オペレーティングシステムの種類の透過性
  RPC の機構により、ネットワーク上の様々なオペレーティングシステムとの通信が行なえる。

- 計算機の種類の透過性
  XDR の機構により、ネットワーク上の様々な計算機との通信が行な

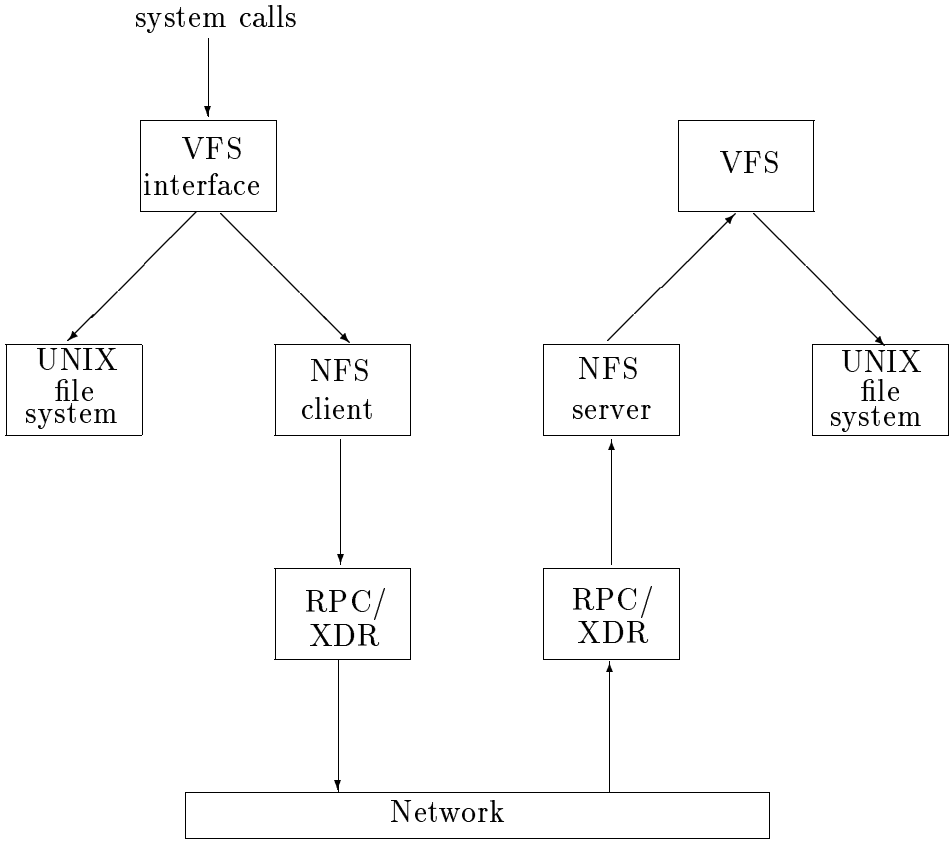


図 2.3: NFS のサーバとクライアントのつながり

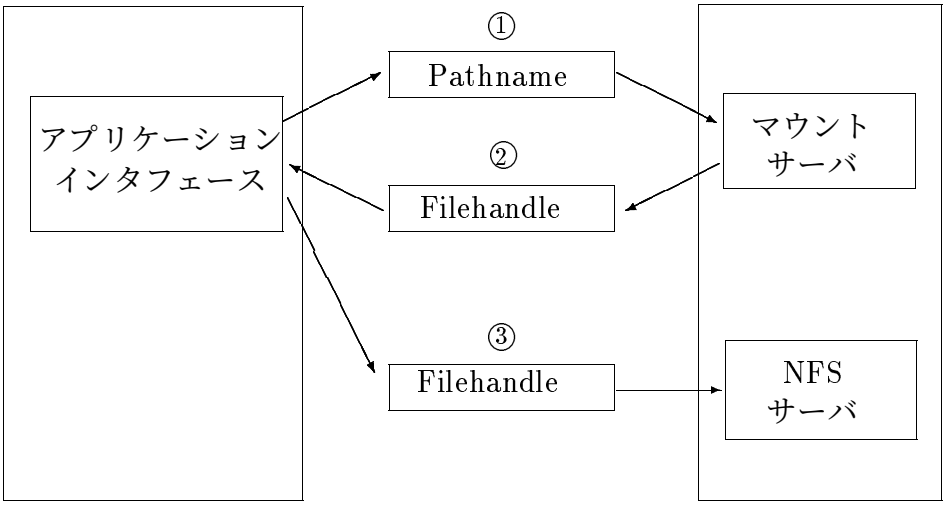
える。

- ネットワークの種類の透過性
  RPC と XDR が様々なトランスポートプロトコルを扱えるので、ネットワークの種類に透過的である。

2.4.2.4 クライアント／サーバ モデル

NFS のインタフェースは filehandle と呼ばれる識別子をファイル操作に使用している。filehandle はサーバとクライアントの間で受渡しが成される。

まず、クライアントはマウントサーバに RPC で、要求するファイルシステムに対する filehandle を得る。filehandle を得たクライアントプログラムは NFS プロシージャを使って別の filehandle を得たり、ファイルシステムの中を行き来することができる。Figuer2.4 にクライアントプログラム、マウントサーバ、NFS サーバの行動を示す。



1 クライアントはマウントサーバに pathname を送る
2 マウントサーバは対応する filehandle を返す
3 クライアントは NFS サーバに filehandle を送る

図 2.4: マウントサーバと NFS サーバ

NFS サーバは stateless である。つまり、サーバは一つのトランザクションの後にそのクライアントに関して何も情報を持っていない。例えば、open() という操作が UNIX インタフェースにはあるが、NFS にはない。これは stateful である。NFS 操作で使うために UNIX 操作における情報をクライアントが保持しているのである。stateless サーバの利点は、クライアントやサーバやネットワークなどの失敗に対する強さである。クライアントが失敗してもサーバは何ら影響されない。サーバあるいはネットワークが失敗した時は、クライアントがサーバあるいはネットワークが回復するまで NFS 操作を続ければよい。異なるシステムの混在する複雑なネットワークではこのような強さが本質的に重要になってくるのである。

2.4.3 RPC を用いた NFS

NFS はいくつかの透過性を提供していたが、NFS の本質ともいえる位置の透過性を実現するために VFS という機構を用いていた。そして、その VFS は RPC と XDR を利用していて、オペレーティングシステムや計算機の種類に関係なく、様々なネットワークを介して構築することができた。

NFS でシステムコールが呼ばれると、目的のファイルの vnode を見つけ、オペレーションルーチンが呼ばれる。このオペレーションルーチンの中で、vnode から目的のファイルがローカルのものかリモートのものかを判断をして、ふさわしいオペレーションルーチンを呼ぶ。リモートファイルの場合には、vnode は変換されて NFS クライアントに渡され、さらに、RPC を用いて NFS サーバに渡される。

RPC は NFS の一部として利用されていて、うまく機能している。RPC を利用することで、NFS のサーバとクライアント間のインタフェースを簡単にしているのである。もちろん、RPC は単独で簡単に利用することができるが、むしろ、ネットワークサービスの中で利用されることを考えていて、比較的大きな規模のネットワークアプリケーションでも有効に利用される。

第 3 章

データグラムによるデータ転送

TCP を用いた RPC もサポートされているが、実際に利用されているのは ほとんどが UDP である。プロシージャコールという性質から、時間をかけてコネクションを張る TCP よりも、UDP の方が効率が良いと思われるからである。

ここでは、UDP を基盤とする RPC を広域分散環境で利用するにあたり、データグラムを効率よく転送をするための条件を考えてみる。

3.1 再送処理

3.1.1 再送制御の必要性

多くの RPC では、クライアントはサーバにサービスを要求してから結果を受け取るまでブロッキングされる。もし、サーバがクラッシュしたり無限にループしていたりすると、永久に結果を受け取ることはできない。しかし、データの転送やサービスの処理に時間がかかっている可能性もある。クライアントは返答が返ってくるかどうか、その返答を受け取るまでわからないのである。サーバの負荷が重い時には、再送を繰り返すことは更に負荷をかけることになり悪循環であるし、サービスの処理にかかる時間よりもタイムアウトの間隔が短いと再送が繰り返されるだけである。

再送はできるだけ避けなければならない。サーバに要求メッセージが到着しているにもかかかわらず再送を行なうことは、その通信が遅くなるばかりではなく、ネットワークに輻輳を生じさせることにもなる。広域分散環境では、個々のアプリケーションが最小限のコストで通信できるように制御していくことが重要である。そこで、RPC としても再送制御をする必要がある。Sprite RPC では要求メッセージに対する確認応答をまず送り、それから処理を始めるようにしている [83]。これにより、相手ホストに対するラウンドトリップタイムを過ぎても確認応答を受け取らないと、クライアントは要求メッセージが受理されていないことがわかる。

しかし、Sun RPC では再送に関してはあまり考慮されてない。lowest layer で、ユーザが RPC に対するタイムアウトと各パケット転送のタイムアウトを指定することで再送回数を指定することができるが、その数値の選択はユーザに任されている。ユーザはその値を経験的に定めるしかない。hightest layer と intermediate layer では全く指定することができないので、刻々と変化するネットワークの状態に対応するためには RPC としてこれらの機能を提供する必要がある。

3.1.2 タイムアウトの不適当な設定による再送

再送はタイムアウトの間隔を越えても応答が返ってこない時に起こる。タイムアウトが長過ぎるとパケットが喪失した時の発見が遅くなる。しかし、タイムアウトが短いと、ネットワークが混雑している時やサーバの負荷が重い時には、実際に要求が受理されていても、すぐ再送が行われてしまう。パケットが喪失したり、あるいは、サービスが適当に処理されないような時にのみ再送がなされるべきである。単に、ネットワークが混雑しているとか処理が遅いということで再送がされるべきではない。しかし、Sun RPC のような同期型の RPC ではこれらのことを区別することは不可能である。ユーザにとっては、応答が戻ってこないということに過ぎないのである。

そこで、適切にタイムアウトを設定することが要求される。しかし、この設定にはさまざまな要因を考慮しなければならない。3.2で述べるような応答時間に関するパラメータに影響されることは勿論であるが、その他にも考慮するべきことは多い。たとえば、エラー率の大きい回線を使用する時にはパケットが喪失する可能性が高いので、タイムアウトは短い方が全体としての効率は良くなるであろう。逆に、信頼性の高い回線を使用する時には多少長めにタイムアウトをとっても良いと考えられる。ネットワークの混雑で到着の遅れた応答メッセージを受け取ることができるかも知れないからである。

3.1.3 マシンの能力の限界による再送

広域ネットワークではさまざまな計算機が接続されており、その性能は異なる。計算機によっては、あまりに速くパケットが到着するとそのすべてを受け取ることができないかもしれない。例えば、Sun1、2 シリーズの初期のものはそのインタフェースのアーキテクチャにより、性能の良いインタフェースを持つ計算機とは実用上通信できないという欠陥があった。クライアントマシンとサーバマシンだけでなく、途中のゲートウェイの性能が劣っている場合も通信はできなくなる。ゲートウェイにおける混雑が

おこらないように、パケットの量を制御することはネットワーク全体がうまく機能するためにも必要である。

また、ネットワークのインタフェースは一度に転送することのできるデータサイズが決まっている。これが Maximum Transfer Unit(MTU) である。この MTU を越すパケットは下位のレイヤでフラグメントされる。フラグメントされたデータが連続して送られると受け取れなくなる計算機がある。転送した計算機の上位レイヤは応答が来ないと判断すると再送のパケットを送出するが、再送を繰り返しても通信をすることは出来ないのである。このような場合は、パケットを送出する際に、時間をおくことで、パケットが連続して到着しないようにしてやることが必要である。

この問題は下位レイヤの解決すべき問題であるが、NFS のように大量のデータ転送が考えられる場合にはその一部として考える必要もある。UDP/IP のサポートする RPC メッセージはデータグラムとして転送されるが、8Kbyte までしか一度に送ることはできない。それ以上のデータを転送する時には、いくつかのデータにして下位レイヤに渡すことになる。もし、RPC としてパケットの送出が速過ぎるような時には、連続して送ることは避け、間隔をおいて送出しなければならない。あるいは、データサイズを減らし、小さなパケットにして送り出さなければならない。

3.2　応答時間に関するパラメータ

RPC の要求メッセージを出してから、結果を受け取るまでの応答時間がわかれば、それに合わせてタイムアウトを指定することができる。適切なタイムアウトが指定できれば、再送を最小限におさえることができる。応答時間は、遅延、データサイズ、バンド幅、処理時間などによって決る。

3.2.1　遅延

遅延はラウンドトリップタイムの 1/2 であると考えられる。ローカルエリアネットワークの場合はほとんど無視することができ、ローカルエリアネットワーク上のプロトコルは通信時間が短時間であることを仮定している。しかし、ネットワークが広域になると、もはや通信時間を無視することはできず、従来のローカルエリアネットワーク上のプロシージャコールはそのままではうまく機能しない。

遅延はネットワークの状況を最も反映していて、その負荷によって決まる動的な量である。回線やゲートウェイの混雑度は時間とともに変化しており、その状況を把握することは難しい。多くのネットワークアプリケーションにとって必要不可欠なこの情報を、正確に、かつ迅速に得ることはま

だ実現されていない。その瞬間の遅延を得ることは不可能である。しかし、データを送出する直前にラウンドトリップタイムを実際に測定すれば、それはそのデータを転送する際の遅延と考えることができる。ただし、この場合、小さなデータを遅延の大きなデスティネーションに転送するような時には、オーバヘッドが大きく、著しくパフォーマンスが落ちることになる。また、データ転送をする度にこのような方法をとっていてはトラフィックが増えて、ネットワークに余分な負荷がかかり、実用的ではない。そこで、通常からネットワークの状況を監視して、その情報をキャッシュしておくことが考えられる。これは正確さに欠けるが、現在の状況を判断するのに十分な情報が提供できるのであれば有効である。

また、応答時間はデスティネーションホストまでの経路にも左右される。例えば、衛星回線を使用する時には、専用回線を使用する時よりもその値は著しく大きくなる。しかし、経路制御は下位レイヤに任されているので RPC としてこれらを扱うことはできない。定められた経路での遅延をいかに正確に得るかが問題になる。

3.2.2 データサイズ

通信時間に最も関わってくるのがデータサイズである。転送するデータの量が大きければ、それだけ転送にかかる時間も長くなるのは当然である。データサイズが小さいとヘッダのオーバヘッドは大きくなるが、フラグメントのオーバヘッドはデータサイズが大きい方が大きくなる。ネットワークアプリケーションを作成する際には、これらの要因が実際の応答時間にどう影響してくるかを知る必要があるであろう。特に、NFS のように、さまざまな サイズのデータを扱う場合は、その動向を正しく把握しなければならない。

3.2.3 バンド幅

バンド幅は回線によって決まる静的な量である。その回線を使用した時に 1 秒で送ることのできる最大のビット数である。ethernet では 10 Mbps、専用回線は 64K や 19.2K、9.6K などが提供されている。また、衛星回線では、もっと広いバンド幅でサービスが行われている。

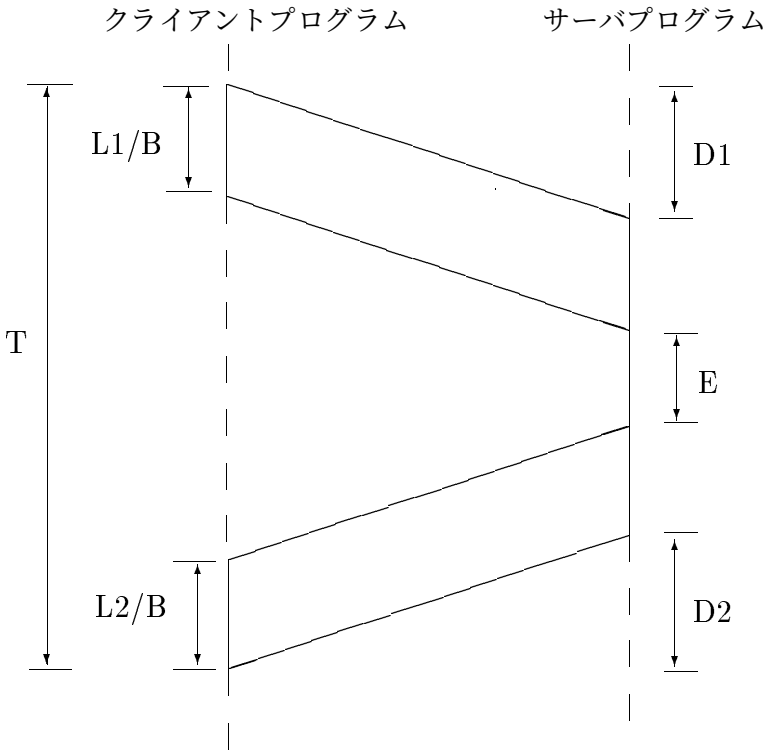
一般に、広域ネットワークでは、回線が低速であることが特徴として挙げられる。これはローカルエリアネットワークと大きく異なる点である。広域分散環境におけるアプリケーションをうまく動作させるためには、この特徴を考慮しなくてはならない。

3.2.4　処理時間

処理時間とはサーバの計算時間のことを意味している。当然、サービスによって異なるが、サービスを実行する計算機の性能、および、負荷によっても変わってくる。サービスの種類によって決まる固有の量はサーバマシンの種類や状態とは無関係で、あらかじめ予測することができる。また、サーバマシンの性能はサービスの種類とは無関係で、ホストが決まればこれも特定することができる。しかし、サーバの負荷は他のクライアントとの関係や計算機の状態によって変わる動的な量である。

3.2.5　応答時間

以上の要因により、応答時間が決定できる。Figure3.1 のように考えることができる。実際のデータサイズをバンド幅で割ったものがデータの転送にかかる時間である。正確には、データをエンコードしたりデコードするときのオーバヘッドなどを考慮する必要もある。



L1, L2 : データサイズ
B : バンド幅
D1, D2 : 遅延
E : 処理時間
T : 応答時間

$$T = D1 + L1/B + E + L2/B + D2$$

図 3.1: 応答時間の解析

バンド幅 (B) は回線によって固有の量であるが、その他の、データサイズ (L1、L2)、遅延 (D1、D2)、処理時間 (E) 等は様々な要因によって変化する量である。ネットワークの混雑の状況が遅延に関わり、サービスの種類が処理時間に関わってくる。これらのパラメータがどのように関わってくるのか、考慮するべきパラメータは何かということを把握すれば、応答時間の予測を行なうことができるのである。

タイムアウトは実際の応答時間よりも長くとらなければならない。もし、短いタイムアウトを指定すると、クライアントは絶対にこの通信を成功させることはできない。低速の回線で、事実上マウントできないのは大きいデータを転送しようとすると応答時間が長くなってしまい、NFS のタイムアウトを越してしまうためである。一回の転送サイズを小さくしてタイムアウトを極端に長くすれば、ユーザインタフェースは悪くても、マウントすることが可能である。

第 4 章

実測

4.1 データサイズと応答時間

RPC を効率良く利用するためには、適切なタイムアウトの決定を行なう必要がある。そのためには応答時間といくつかのパラメータとの関係を知らなくてはならない。ここでは、データサイズの応答時間への影響を、実際のネットワーク上で計測をした。

実験をおっこなったネットワークの接続の状況を Figure4.1に示す。

4.1.1 実験プログラムの説明

クライアントプログラムは指定されたデータサイズを持つ RPC 要求メッセージをサーバに送り応答を待つ。その際、sendto システムコールでデータを送出する直前と、recvfrom システムコールで応答を受け取った直後に gettimeofday システムコールで応答時間を計測する。エラーの把握のため、再送は行なっていない。

サーバプログラムは RPC 要求メッセージが到着すると、データをデコードする。本来は、それをサービスプロシージャに渡して、サービス処理をするのであるが、ここではそのまま応答メッセージの中にエンコードして、クライアントプログラムに送り返す。これにより、サーバの処理時間を無視できる。ただし、デコードとエンコードに費やされる時間は考慮していない。

RPC モデルとしては要求メッセージと応答メッセージはデータサイズが異なるが、この実験に関しては同じものを転送しているので、そのサイズは全く同じである。

4.1.2 ネットワークの接続状況との関係

現在、ローカルエリアネットワークでは計算機と計算機の接続に ethernet を使用しているのが一般的である。また、広域ネットワークでは 9.6Kbps

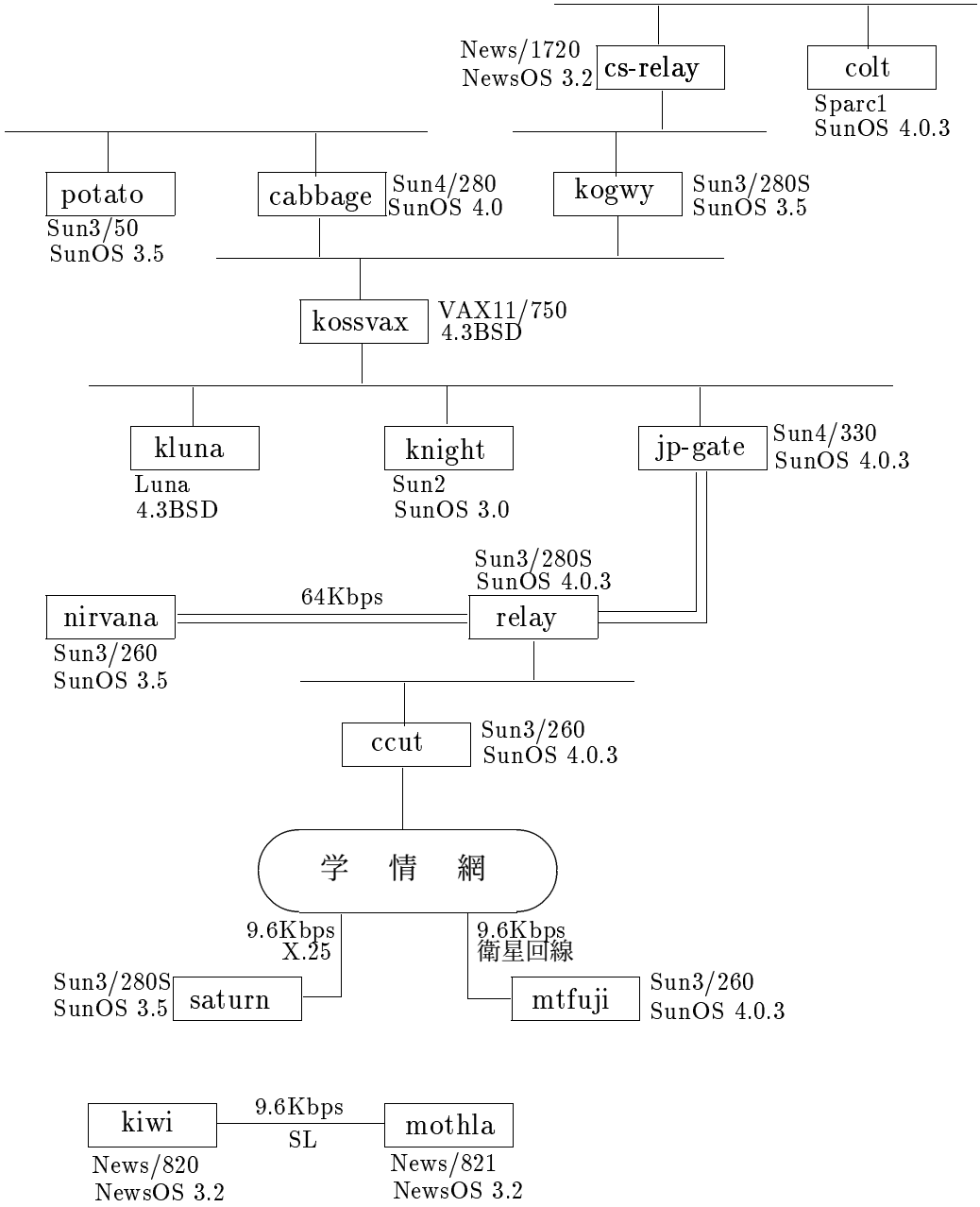


図 4.1: ネットワーク接続図

や 64Kbps などの回線を使用していることが多い。利用する回線の種類やゲートウェイの有無が、データサイズと応答時間の関係に影響してくると思われる。途中のゲートウェイでフラグメントされる場合もあるし、低速の回線へのゲートウェイでフロー制御のためにバッファリングされてしまうかも知れないからである。

(1) ethernet の場合

次のような場合について実験を行なった。

- 同じ ethernet 上
- ゲートウェイが一つ
- ゲートウェイが二つ

これらのいずれの場合も、データサイズと応答時間の間に線形的な関係がある (付録 A FigureA.1参照)。ethernet の場合にはゲートウェイの有無によって線形的な関係が崩されることはないといってよい。

しかし、ゲートウェイの性能により、応答時間そのものに大きな差が出る。cabbage–kluna 間にはゲートウェイとして VAX が 1 つ存在していて、cabbage–colt 間には Sun3 と News がゲートウェイをしている。ゲートウェイが一つの場合の cabbage–kluna 間の通信の方が、ゲートウェイが二つの場合の cabbage–colt 間の通信よりも、8000byte のデータを転送する際に約 2.7 倍、時間にして 0.12 秒ほど長くかかっている。

(2) 64Kbps の専用回線の場合

- ゲートウェイ無し
- ゲートウェイが一つ

の場合について実験を行なった。ehternet の場合と同様にデータサイズと応答時間の関係は線形的で、ゲートウェイを介さない時はほぼ完全な直線になっている (付録 A FigureA.2参照)。ゲートウェイが一つ存在する場合は多少傾きが大きくなっていて、応答時間も長くなっている。jp-gate–nirvana 間に存在するゲートウェイは、ゲートウェイが無しの場合のサーバマシンの relay であるので、応答時間が長くなっているのは当然である。

$$T = D1 + L1/B + E + L2/B + D2$$

の式を用いると、いま、L1 = L2、E = 0、であり、D1 = D2 と考えると、jp-gate–relay 間は

$$T = 2(D1 + L1/B)$$

という応答時間になる。また、jp-gate–nirvana 間は、

$$T = 2(D1' + L1/B) + O$$

となる。ここで、O はゲートウェイにおけるオーバヘッドである。D1′ は jp-gate–nirvana 間の遅延であり、D1′ > D1 である。jp-gate–nirvana のグラフの方が jp-gate–relay のグラフよりも傾きが大きく、上方にあるのが理解される。

(3) 9.6Kbps の回線の場合

次のような場合を考えてみた。

- X.25 網国内回線
- X.25 網衛星回線
- シリアル回線

X.25 網の国内回線と衛星回線を用いた場合は、計測結果にかなりのバラツキがあった (付録 A FigureA.3、FigureA.4参照)。平均をとると線形になっていることが読みとれるが、最長応答時間と最短応答時間には、7 秒近くの差がある場合もある。平均値をそのデータサイズに対する応答時間とすることには無理がある。もっと、バラツキを吸収するような代表値を考えなくてはならない。データサイズが大きい方が傾きが緩やかになるのは、データサイズが大きいとパケット喪失が多くなり、戻ってくるパケット数が少ないため、バラツキが小さくなるからである。

シリアル回線を使用した実験は、他のトラフィックが全くなく、計算機にも負荷はかかっていない状態で行なった。この場合は、完全な直線であって、バラツキも全くない (付録 A FigureA.5参照)。T = 2(D1 + L1/B) の式を考えてみると、B は定数で L1 は変数である。D1 は本来変数であるが、ここでは、実験パケットしか流れていないことを考えると、ある時刻には一定の値を持つと考えてよい。すると、応答時間 T はデータサイズ L1 の一時関数になるが、そのグラフは理論と一致して直線になっていることがわかる。しかし、上の二つの例では、ホスト ccut が平常から負荷の大きい計算機である上、ネットワークにもかなりのトラフィックがあるので上で見たようなバラツキが現れてくるのである。

(4) バンド幅の違う回線を接続した場合

jp-gate–saturn は Figure4.1 にあるように、64Kbps と 9.6Kbps の回線がゲートウェイ一つで接続されている。9.6Kbps の回線を使っているので、やはり、かなりのバラツキがある。平均をとっても、今までのように線形であるとはいい難いようである (付録 A FigureA.6参照)。実際のネットワークの複雑な様子がうかがえる。

4.1.3 ネットワークの混雑状況との関係

ネットワークの混雑の影響も考えられる。回線が混雑していればそれだけ通信は遅くなり、すいている時よりも応答時間は余分にかかることになる。ここでは、その回線に spray をしてネットワークが混雑する状況を作り出した。spray は指定したホストに対し RPC メッセージを連続的に送り、転送効率とともに相手ホストがどれだけデータを受け取れたかをテストするもので、パケット数、各パケットを転送する間隔、データサイズなどが指定できる。タイムアウトを 0 にしてあり、応答パケットを期待しないし、サーバプロセスも応答を返すことをしていない。指定されたサイズのデータを全て送った後で、相手ホストにいくつのパケットを受け取ったかを RPC で尋ねる。計測を行なう回線の外側の計算機同士でお互いに spray を実行して、クライアントマシンとサーバマシンに負荷をかけずに、その回線を故意に混雑させた。

(1) ethrnet の場合

ネットワークの混雑状況はほとんど影響していない。平常時における測定と全く同じ計測結果を得た (付録 A FigureA.7参照)。実験時は、その回線を使用していて、かなりネットワークが混んでいるという感じを受けたが、ethernet のような高速の回線では応答時間には影響を及ぼさないようである。

(2) 64Kbps の専用回線の場合

この場合は、混雑させた時の方が、平常時よりも応答時間は長くなっていて、データサイズが大きいほどその影響を受けやすいようである (付録 A FigureA.8参照)。8000byte のデータ転送では平常時のときの実験と約 0.4 秒の差が出た。しかし、応答時間は長くなっているが、それでも、データサイズと応答時間の線形的な関係はそのままであるといって良い。

(3) 9.6Kbps の回線の場合

シリアル回線でホスト同士を接続して、そこで spray を実行しながら実験を行なおうとしたが、応答パケットを全く受け取ることができなかった。spray の応答も受け取れなかった。9.6Kbps ではバンド幅が狭いのでトラフィックが少なくてもすぐ回線が混んでしまうようである。

X.25 網の場合も spray の応答パケットが受け取れなかったので、実験プログラムは走らせなかった。

4.1.4　サーバの負荷との関係

NFS のようなアプリケーションでは、サーバに RPC の要求メッセージが過度に集中する場合も考えなくてはならない。ここでは、spray コマンドを自ホストに対して実行し、ネットワークとは関係なく、純粋にサーバのみに負荷をかけた。ロードの平均が 6 くらいの状態を保って同じプログラムを走らせた。

(1) ethernet の場合

cabbage–kluna の通信でサーバである kluna に負荷をかけた。負荷をかけた時の方が応答時間は長くなるが、値としては負荷をかけない時とあまり変わらず、タイムアウトの値に影響するほどの差はない (付録 A FigureA.7 参照)。0.01 秒程度の差であるので、0.02 秒単位で動いているクロックがあることを考えるとこの値は問題ではない。

(2) 64Kbps の専用回線の場合

jp-gate–relay の通信でサーバである relay に負荷をかけたが、この場合は平常時との差が全く現れなかった (付録 A FigureA.8参照)。正確には 0.01 秒程度の差があることもあるが、これは応答時間が 1 秒程度であることを考えると、無視できる範囲である。

4.1.5　ゲートウェイの負荷との関係

ゲートウェイの負荷がパケットの転送に、実際にどのように影響するのかを計測した。経路制御を行なうのはゲートウェイの IP レイヤで、パケットを一時的に取り込みデスティネーションに向けて送り出すのである。ゲートウェイの負荷が大きい時にはその動作が遅くなることも予想される。計測は、経路上のゲートウェイで、自ホストに対して spray コマンドを実行して、ロード平均が 6 くらいの状態で行なった。

(1) ethernet の場合

potato–kluna の通信でゲートウェイの一つである cabbage に負荷をかけて実験を行なった。しかし、全くその影響を見ることはできなかった (付録 A FigureA.9参照)。

(2)64Kbps の専用回線の場合

jp-gate–nirvana の通信で、ゲートウェイである ralay に負荷をかけた。この場合も、全く影響は現れなかった (付録 A FigureA.8参照)。

ここでは、ゲートウェイに負荷をかけるために、自ホストに対して spray を行なったが、その効果がじゅうぶんでなかったのかもしれない。

4.2 エラー率

これらの負荷を考えた場合、もう一つ考慮しなくてはいけないのは、パケット喪失の起こる頻度である。エラー率というのは、再送の制御をする際、かなり重要になってくる点である。計測に使ったプログラムでは再送を行なわないようにしてあって、エラー状況も観測できるようにした。平常時には ethernet も 64Kbps の専用回線もごく稀にしか落さず、再送制御をすればほとんど問題はない。

しかし、ネットワークを混雑させた場合は、極端にその回線の信頼性が落ている。実験によると、ethernet の場合で最大 9% のパケットを落していて、4500byte 以上になると、3 〜 4% くらいは常に落してしまう。また、64Kbps の専用回線では、8000byte のデータを送るのに、22% のパケットを落している。このプログラムの最小データサイズである 86byte のデータを転送する時でさえ、1% 落してしまっている。

サーバマシンに負荷をかけた場合は、ethernet ではデータサイズが大きくなるにつれ 10% 程度落した。ところが、64Kbps の専用回線では平常時と変わらずほとんどすべてのパケットを受け取ることができていて、エラー率は無視できる。

一方、ゲートウェイに負荷がかかった場合には、その影響は受けていないようで、平常時と同様ほぼ 100%送ることができている。

転送における信頼性は、再送制御によってハードウェア的なエラーをカバーすることができるかどうかによって決まってくる。ethernet や 64kbps の専用回線であれば、再送の回数を適当に指定することで、高い信頼性が得られる。しかし、9.6Kbps の回線ではデータサイズが大きくなるにつれ、

(100 のパケットのうち応答のなかったパケット数)

	86[btye]	1002	2000	3000	4000	5000	6000	7000	8000
(1)	0	0	0	0	0	0	0	0	0
(2)	0	0	0	0*	–	–	–	–	–
(3)	0	0	0	0	0	0	0	0	0
(4)	0	0	0	0	0	0	0	0	0
(5)	0	0	0	0	0	0	0	1	0
(6)	2	0	0	2	2	3	4	3	9
(7)	1	2	2	3	4	8	12	7	10
(8)	0	0	0	0	0	1	0	0	0

* 2994byte での値

(1)cabbage–potato:ゲートウェイなし
(2)kluna–knight:ゲートウェイなし
(3)cabbage–kluna:ゲートウェイ１つ
(4)cabbage-colt:ゲートウェイ２つ
(5)potato–kluna:ゲートウェイなし２つ
(6)cabbage–kluna:ネットワークを混雑させた場合
(7)cabbage–kluna:サーバに負荷を与えた場合
(8)potato–kluna:ゲートウェイに負荷を与えた場合

表 4.1: ethernet でのパケット喪失状況

(100 のパケットのうち応答のなかったパケット数)

	86[btye]	1002	2000	3000	4000	5000	6000	7000	8000
(1)	0	0	0	0	0	0	0	0	0
(2)	0	0	0	0	0	0	1	0	4
(3)	1	6	10	13	17	15	18	16	22
(4)	3	0	0	0	0	0	0	0	0
(5)	0	0	0	0	0	0	0	0	0

(1)jp-gate–relay:ゲートウェイなし
(2)jp-gate–nirvana:ゲートウェイ２つ
(3)jp-gate-relay:ネットワークを混雑させた場合
(4)jp-gate–relay:サーバに負荷を与えた場合
(5)jp-gate–nirvana:ゲートウェイに負荷を与えた場合

表 4.2: 64Kbps 専用回線 でのパケット喪失状況

(50 のパケットのうち応答のなかったパケット数)

	86[btye]	1002	2000	3000	4000	5000	6000	7000	8000
(1)	0	0	0	0	1	4	26	41	–
(2)	0	0	0	2	2	11	27	46	–
(3)	0	0	0	0	0	0	0	0	–
(4)	2	0	0	0	8	6	20	36	–

(1)ccut–sarutn:国内回線 (X.25)
(2)ccut–mtfuji:衛星回線 (X.25)
(3)kiwi–mothla:シリアル回線
(4)jp-gate–saturn:バンド幅の違う回線の接続の場合

表 4.3: 9600bps 回線、及び、バンド幅の違う回線の接続 でのパケット喪失状況

パケット喪失率も大きくなっていて、8000byte のデータは送ることができなかった。このような場合にはそのエラー率を考慮して再送を制御する必要がある。

4.3　データサイズと転送効率

経験的には、データサイズは大きい方が転送効率が良いと考えられている。計測の結果からも、ほぼそのように考えられる。例えば、64Kbps の専用回線でゲートウェイを一つ介する場合、4000btye のデータを転送するのに 1389ms、8000byte では 2434ms かかっている。4000byte のデータを二回送るよりも 8000byte のデータを一回送る方が良いということになる。実験を行なった他のほとんどの場合、同じ考察をすることができる。

しかし、シリアル回線を使用して他のトラフィック無しに行なった実験では、逆の結果を得たのである。3000byte のデータ転送に 6343ms、6000byte では 12751ms を要している。多少ではあるが、3000btye のデータを二回送った方が良いことになる。この場合は、計算機にほとんど負荷がなく、実験パケット以外のトラフィックは全くないので実際のネットワークの状況とは異なっているのであるが、単に、データサイズは大きい方が効率が良いとは言えないことになる。また、RPC として、データを分割して送った方が良いかどうかを考える必要がある。実際、同一ホスト上の同一プロシージャを再利用する場合はキャッシュを用いるので早い計算機の場合には 10ms 以下、遅い場合は 40ms 程度かかることもある。新たに転送を行なう場合は数百 ms 程度である。計算機の能力によって、データサイズが大きい方がいいのか小さい方がいいのかが決まってくるであろう。

ただし、シリアル回線の実験結果は一般的なネットワークの状況とはかけ離れているので、やはり、実際のネットワーク上で行った実験の結果が有効であろう。応答時間に関してデータサイズが大き方が良いとなれば、RPC としてのオーバヘッドを考慮する必要ななくなる。一概に、データサイズが大きい方が良いとは限らないということを知った上で、状況を良く考慮してアプリケーションを作成することが大切である。

第 5 章

RPC の最適利用のための支援システム

実際のネットワークにおけるくつかの計測をもとに、RPC を利用する際の最適なデータサイズとタイムアウトを決定する機構を考えてみる。

5.1 データサイズ

一般的な見方として、転送するデータサイズは大きい方が良いということであったが、一概にそうとはいえない実験結果があった。実際に通信したいホストとのリンクではどうなのかを知らなくてはならない。いくつかの結果から、データを分けて送った方がいいのか、それとも大きい方が効率がよいのかということを判定するような機構が必要である。

RPC として一つのパケットにかかるオーバヘッドを α とすると、

$$f(x) + 2\alpha$$

$$f(2x) + \alpha$$

の大小比較になり、この値の小さい方がよいことになる。ただし、$f(x), f(2x)$ はデータサイズが $x, 2x$ の時の応答時間である。

このオーバヘッドの値は、計算機の能力によって決まってくるであろう。もし、データサイズが大きい方が効率が良いのであれば、このオーバヘッドを考慮する必要はない。データサイズが小さい方が効率が良いという時にこの値を知り、それを反映するようにする。実際には、データサイズは大きい方が効率が良いようなので、このオーバヘッドについてはあまり細かい考察をしないことにする。

そこで、データサイズは大きい方が良いとなると、相手ホストに対して送ることのできる最大のデータサイズを知らなくてはならない。ethernet や 64Kbps の専用回線では、8000byte のデータを送ることができた。しかし、9.6Kbps の回線では 7500byte のデータはエラー率は高くても何とか送れるが、8000byte のデータは一つも送ることができなかった。また、Sun1

や Sun2 では、3000byte 以上のデータは受け取ることができなかった。つまり、ネットワークの状況や計算機の種類によって送ることのできる最大のデータサイズというものが決まってくる。安定した通信を行なうためには、エラー率を考慮して安全に送れる最大データサイズを知ることが必要である。

この回線のエラー率は再送するべき回数にも影響するが、すでに再送回数が決定されているとすると、その再送の後、応答を受け取ることのできない率を RPC におけるエラー率と考えることもできる。この RPC におけるエラー率がある数字以下になるようなデータサイズが、安定した通信を行なえるデータサイズということになる。信頼性が低く、パケット喪失の多い回線では、ある確率で必ずパケットが戻ってくるようなデータサイズを、転送するべきデータサイズとする必要がある。しかし、ethernet のように信頼性があり、再送によって RPC としてのエラーがほとんどないような回線では、単純に送ることのできる最大のデータサイズが最適なデータサイズということになる。

送ることのできる最大データサイズを知る方法として通常からトラフィックを監視しておく方法と、必要な時に実際に送ってみるという方法が考えられる。目的は、どんな相手に対しても NFS を問題なく使える環境を作ることであるので、最終的にはネットワークの通常のトラフィックから情報を得るのがよい。ここでは、RPC を最適な効率で利用できるように、データサイズとタイムアウトを適切に選択するようなシステムを作ることを考えているので、実際に測定用データを送る方法をとる。このシステムがうまく動作するようになった後で、NFS の中に組み込むことを考える。

最大データサイズを得るには、応答パケットが戻ってくるに十分な時間待たなければならない。十分な時間の待機の後、戻ってこなければ小さいサイズのパケットを、戻ってきたならば大きいサイズのパケットを送ってみることにする。そして、最終的に求めるべきデータサイズというものに収束させる。

5.2　タイムアウト

転送するデータサイズはそのリンクで安全に送れる最大のデータサイズが良いということであった。ここでいう “安全” にという言葉は再送によって、RPC の失敗を回避することができるという意味である。その安全に送れる最大データサイズを送受するのにかかる応答時間を知ることができれば、タイムアウトをその応答時間から適切に決めることができる。

パケットが必ず返ってくるという保証があるのならタイムアウトは長

く取った方がよい。ethernet を使用しているローカルエリアネットワークでは、信頼性も高く、応答時間も無視できるほどであるのでタイムアウトは多少長く取っても問題はない。むしろ、このような回線では長くとるべきである。回線が混雑しているときやサーバマシンに負荷のかかっているときには一割程度のエラーがあるが、それでも、応答時間が短いので、再送がなされても効率が極端に落ちることはない。

問題は、信頼性の低い回線を使用する際のタイムアウトの決定である。64Kbps の専用回線では平常時にはパケットを落すことはほとんどないが、ネットワークが混雑している時にはかなりの率でパケットを喪失するようである。また、9.6Kbps の回線では、データサイズが大きくなると極端に信頼性が落ちていた。このような場合は、何パーセントのパケットが返ってくるかという指定が必要になってくる。その値を満たすようなデータサイズを見つけ出さなければならない。そして、そのデータサイズのパケットが送り出されて戻ってくるまでの応答時間を知り、その値にしたがってタイムアウトを決定するべきである。

さらに、適切なタイムアウトの決定には、応答時間のバラツキを無視することはできない。このバラツキは、最大値をとれば、平均値をとった時と同様に線形になることがうかがえる (付録 A FigureA.3、FigureA.4 参照)。平均値のグラフの傾きと最大値のグラフの傾きとの関係は何ともいえない。ccut–saturn 間ではほぼ同じ傾きのであるし、ccut–mtfuji 間では最大値のグラフの傾きの方が大きい。jp-gate–saturn 間でも、やはり、最大値のグラフの傾きの方が大きい。しかし、いずれの場合も、データサイズが大きくなってパケット喪失率が大きくなると、最大値として予想される時間待っても、パケットは返ってこなことがわかる。

最適なサイズのデータを送受するのにかかる応答時間がわかったとすると、タイムアウトは次のように決定することができる。

T をそのデータサイズに対する平均の応答時間、f を回線によって定まる応答時間のバラツキ度、α を上乗せ値として、

$$timeout = T * f + \alpha$$

のようになる。バラツキは f で考慮されているので、この上乗せ値は応答時間の何割というような指定で良いと考える。

5.3 再送回数

再送はパケットが喪失したときにのみ行なわれるべきである。パケットが喪失したかどうかは十分長い時間待たなければわからないが、回線のエ

ラー率というものがわかっていればある程度予測することができる。実験の結果から、平常時には、ethernet や 64Kbps の専用回線では平均的な応答時間を越えて返ってこないパケットはほとんどなく、再送を行なうことで RPC が失敗することは回避される。

NFS では再送の回数を retrans という変数によってマウント時に設定することになっていて、Sun のデフォルトは３回である。信頼性のある回線では十分な値と思われる。また、9.6Kbps の回線のように信頼性の高くない場合は、再送回数が３回ぐらいでも安全に通信できるようなデータサイズを選択することが必要である。

第 6 章

プロトタイプの試作

6.1 データサイズとタイムアウトの決定

第 5 章で考えてきたように、RPC を最適に利用するためのネットワークの診断システムは、様々な要因を考慮しなくてはならない。ここでは、指定した相手ホストに対して送ることのできる最大データサイズと、そのときのタイムアウトを知ることを考える。

6.1.1 最大データサイズの決定

二分探索のアルゴリズムを用いてデータサイズを決定することにする。すなわち、あるデータサイズの RPC パケットを送ってみて、応答が返らなければデータサイズを小さくして送り、応答が返ればデータサイズを大きくして送る。

転送するデータの範囲は 500byte から 8000byte である。初期値として 250byte と 8250byte を指定しておくと、転送するデータサイズの初期値は 250byte と 8250byte の中間値である 4250byte となる。10 回の転送のうち 2 回応答が返ってこないとデータサイズを 4250byte と 250byte の中間値である 2250byte にして転送してみる。1 回だけ応答が返らず 9 回応答が返るか、10 回応答が返ってきたら、4250byte と 8250byte の中間値である 6250byte のデータサイズを持つ RPC パケットを転送する。以後、これを繰り返す。成功した時に、転送できなかったデータサイズとの差が 500byte 以下であったら、終了し、そのときのデータサイズを返す。500byte のデータが送れないときは RPC を相手ホストに対して行なうことはできないと判断する。これは、IP の最小パケットサイズが 512byte であることから決定した。

1 回の RPC では 3 回の再送を行う。RPC の応答が返ってこないということは UDP のパケットを 4 回続けて喪失したことになる。

6.1.2　タイムアウトの決定

RPC を行なう毎に応答時間を測定している。lowest layer に提供されているライブラリの中で、sendto システムコールを行なう直前から recvfrom システムコールの直後までの時間を測定する。

RPC に与えるタイムアウトは、データサイズを大きくする時は、その時のデータサイズを転送するのにかかった応答時間の最大値の 3 倍の値を使う。データサイズを小さくする時は、返ってきた応答パケットがあればその応答時間の最大値の 2 倍を、なければ、そのときに使っていたタイムアウトをそのまま使う。

返す値は、定まったデータサイズに対する応答時間の平均値の一割増しの値である。

6.2　結果

結果を付録 Bに示す。

ethernet や 64Kbps の専用回線の場合には 8000byte まで送ることができていて、その応答時間も安定していることがわかる。kluna–knight(sun1.5) の場合には、実際、2994byte までのデータしか送ることはできない。この場合の結果は、2875byte であるので、適当と思われる。

9600bps の場合はかなり不規則であることがうかがえる。例えば、ccut–saturn 間の通信では、十分と思われるタイムアウトで、5281byte のデータは送れないが、5265byte のデータは 9 回続けて送れて、成功している。

6.3　評価

実際に転送してみるデータサイズを 500byte から 8000byte に限定したが、これは、初期値の与え方で設定できるし、RPC に与えるタイムアウトの初期値も同様である。こういった変数をコマンド行から入力する方法も考えられる。ある程度、回線の様子がわかっている場合にはその方が良いであろう。たとえば、クライアントが kluna で、サーバが knight の場合は、同じ ethernet 上にあり、タイムアウトの初期値として 20000ms は長過ぎる。応答が返ってくるのであればいいが、この場合は、knight が Sun1.5 であり、最初の 10 回の転送は全て失敗するのであるから、タイムアウトが長いことは不都合である。タイムアウトの初期値の与え方を、コマンド行からの入力にするか、あるいは、回線のバンド幅によって決めるなどの対処があっても良かった。

応答時間の一割増しの値をタイムアウトとしたが、cabbage–colt、kluna–knight、ccut–saturn、jp-gate–saturn の通信では、実際にその値よりも長い応答時間で戻ってきているパケットがある。これは、バラツキを考慮していないためである。ローカルエリアネットワークのように応答時間の短い通信では、計算機のクロックの性能により誤差が出てくる場合も考えられるが、バラツキを考慮するなら、上乗せ値 α としては、一割程度の増分で良いと思われる。

ccut–saturn の通信の結果を見ると、5281byte で送れなかったのが、5265byte の 2 回目から送れるようになっている。この回線は、バラツキが大きかったのであるが、2 分探索のアルゴリズムでは、もし、急に、ネットワークが混雑し始めて応答時間が長くなったとすると、一度送ることができると判断したデータサイズよりも小さいパケットを送ることはしないので、結果は収束しないことになる。本当に、ネットワークの状況がそれほど頻繁に変化しているのならば、2 分探索のアルゴリズムをそのまま使うのではなく、応用して使わなければならない。

6.4 今後の課題

今後の拡張に関して次のようなことが挙げられる。

- 10 回に 9 回応答が返ればいいということにしたが、エラー率というものを正しく扱わなければならない。RPC としてのエラー率を指定し、その範囲内で応答の返ってくるようなデータサイズを探さなければならない。

- バラツキについて何も考慮していないが、必要不可欠な要因である。バラツキによっては、タイムアウトの設定に大きな違いがあるし、データサイズが小さい時の方が大きいときよりも、応答時間が長くなることもあり得る。応答時間の最大値も、また、データサイズと線形的な関係があったことをもっと考慮するべきである。さらに、バラツキの分散を知ることも必要なのかも知れない。

- 実時間で行なっているが、低速の回線では、結果が出されるまでにかなりの時間がかかる。実時間で計測することの長所は、その瞬間のネットワークの状況を知ることができることである。しかし、将来、NFS などのシステムでの利用を考えるとそのネットワークの全体的な状況を把握した方が良いということも考えられる。また、統計情報は多い方が良い値を導くことができるかもしれない。現在の状況を反映する範囲で、通常からの監視により、情報を集める方がよい。ネッ

トワークを診断するために、ネットワークのトラフィックを増やすこともなくなる。

- 効率のよい転送を考えなくてはならないのであるが、今回は転送効率について評価していない。返されたデータサイズとタイムアウトで転送した場合の転送効率も、値として返すべきであろう。

- UDP を用いた RPC ということで、8Kbyte までのデータしか扱わなかった。しかし、これは、拡張されるべきものと考える。ethernet のように信頼性が高く、かつ、高速の回線を利用して通信を行なう場合には、もっと大きなデータを転送した方が効率は良いのかも知れない。UDP のプロトコルとしては何も規定していないので、RPC として、この値を大きくとることができるのではないだろうか。現在は、RPC では 8Kbyte のデータを送ることができないが、おそらく、この拡張は可能であろう。

付録 A

データサイズと応答時間の関係

第 4 章で行なったデータサイズと応答時間の関係の計測の結果を以下に示す。

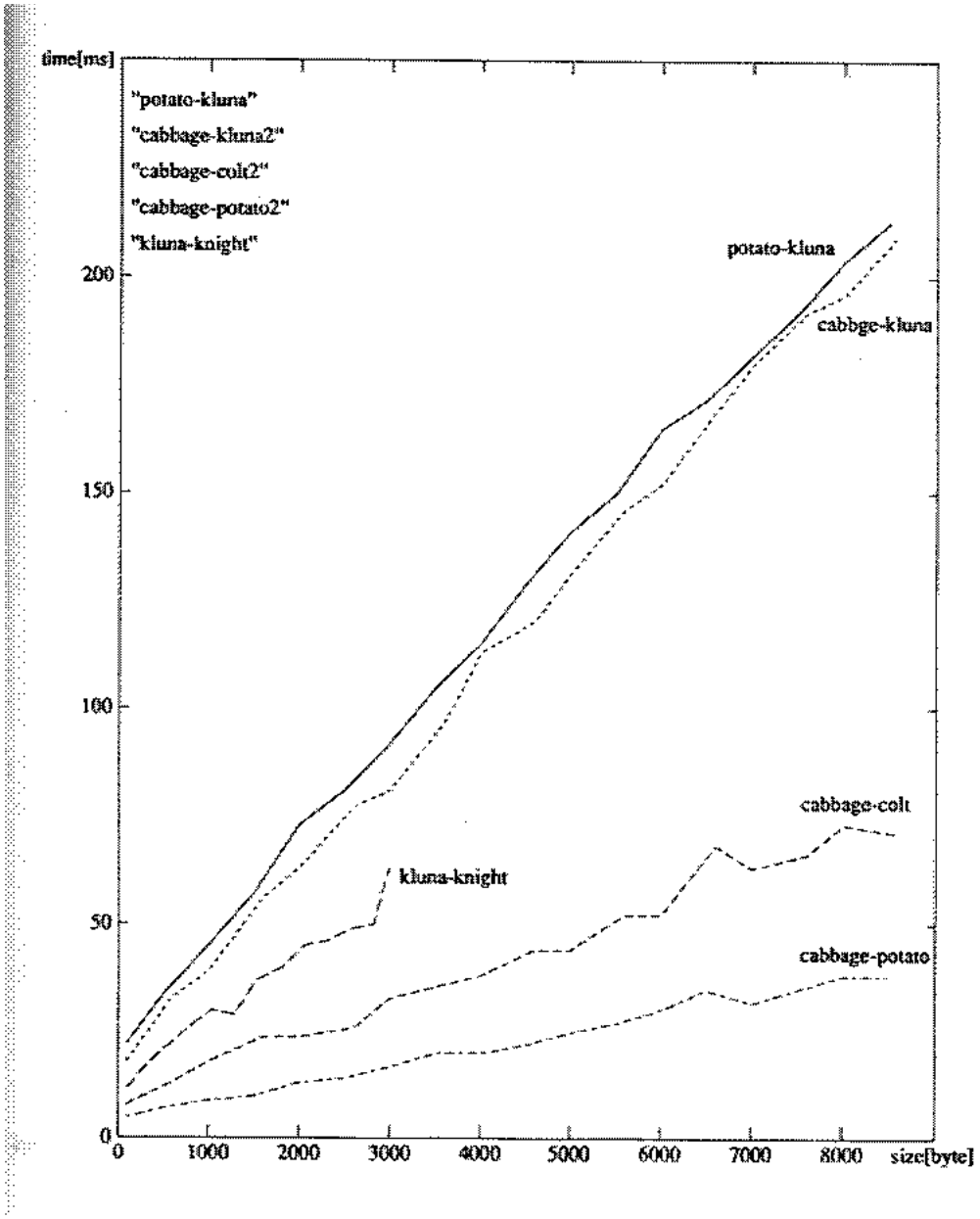


図 A.1: ethernet の場合

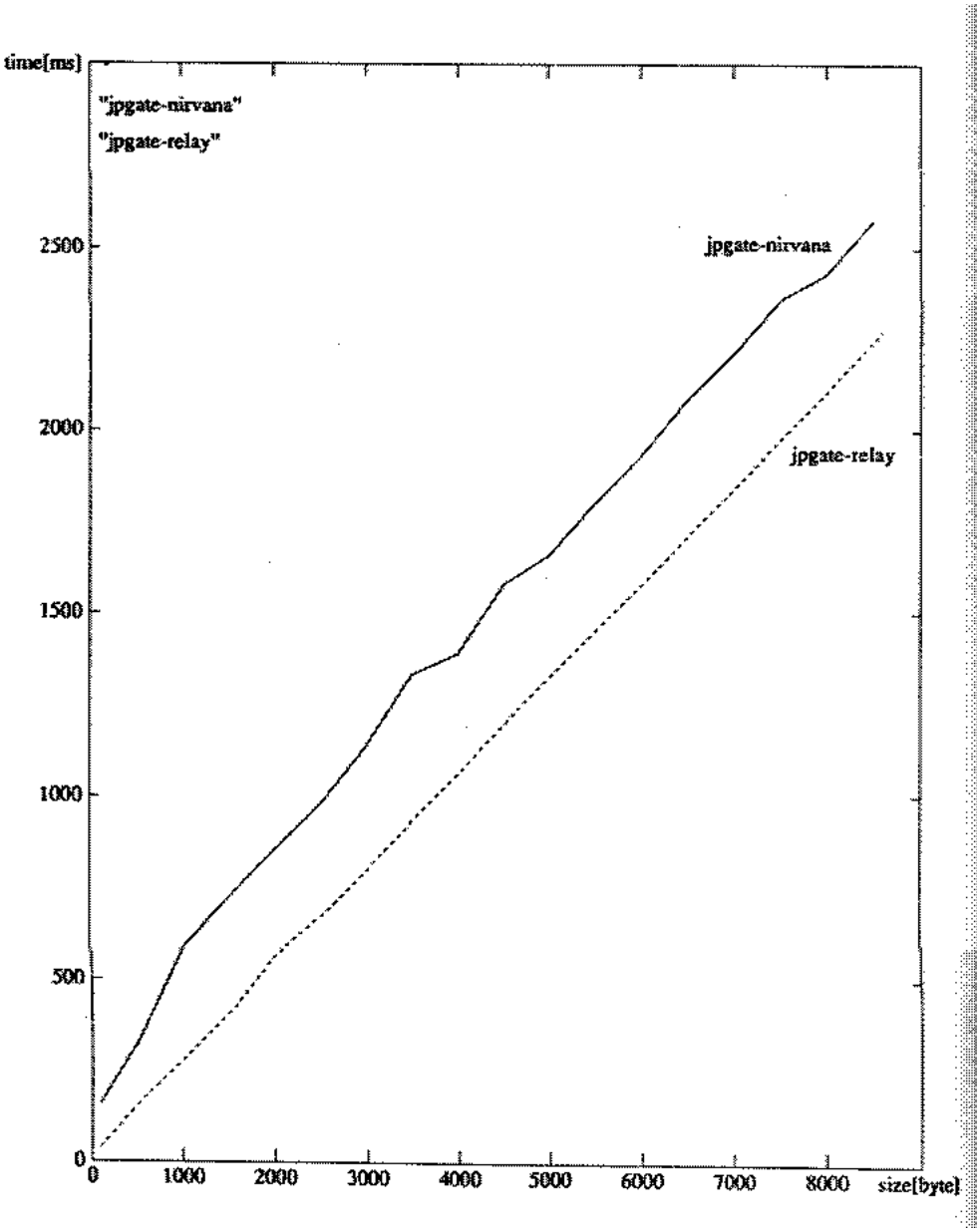


図 A.2: 64Kbps 専用回線の場合

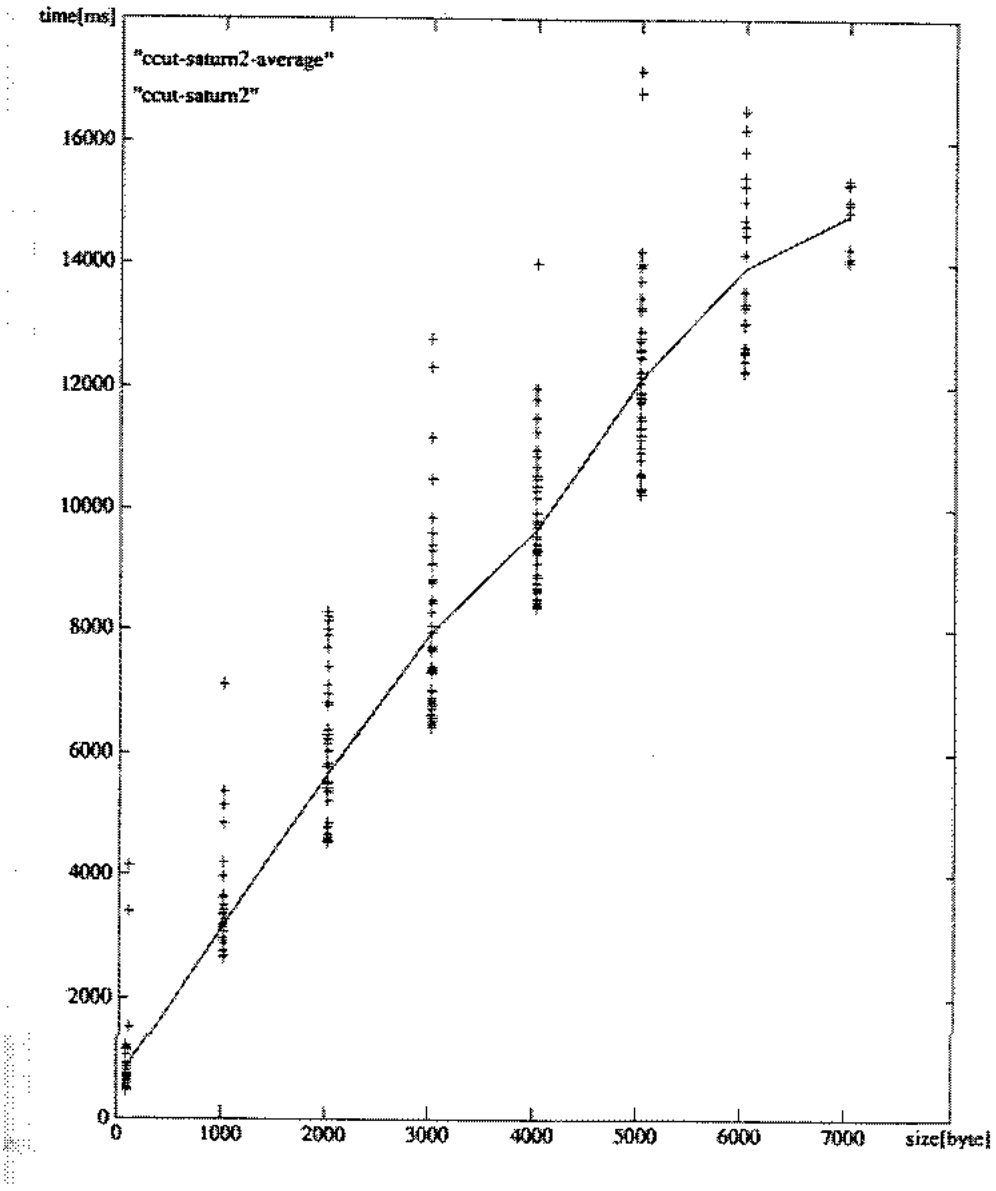


図 A.3: 9.6Kbps 国内回線の場合

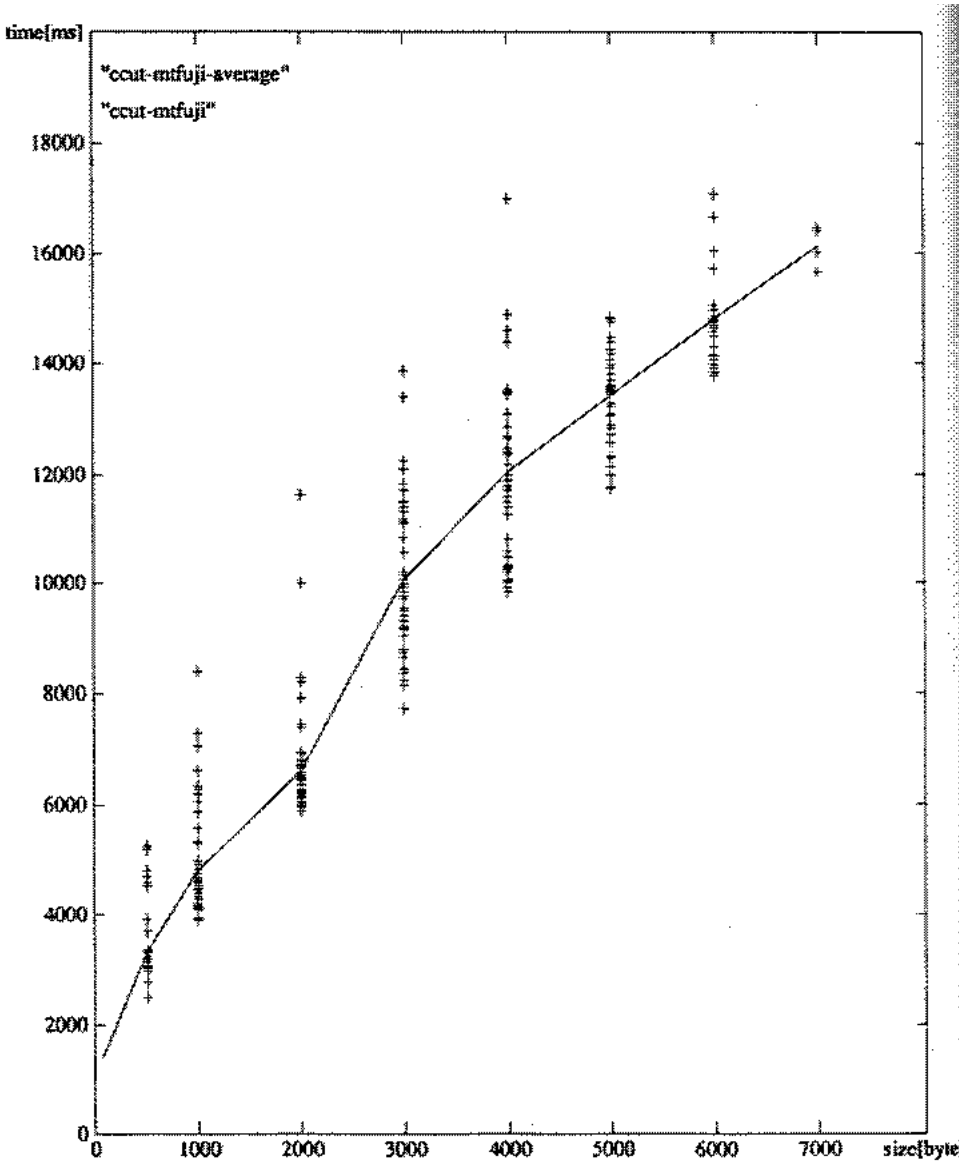


図 A.4: 9.6Kbps 衛星回線の場合

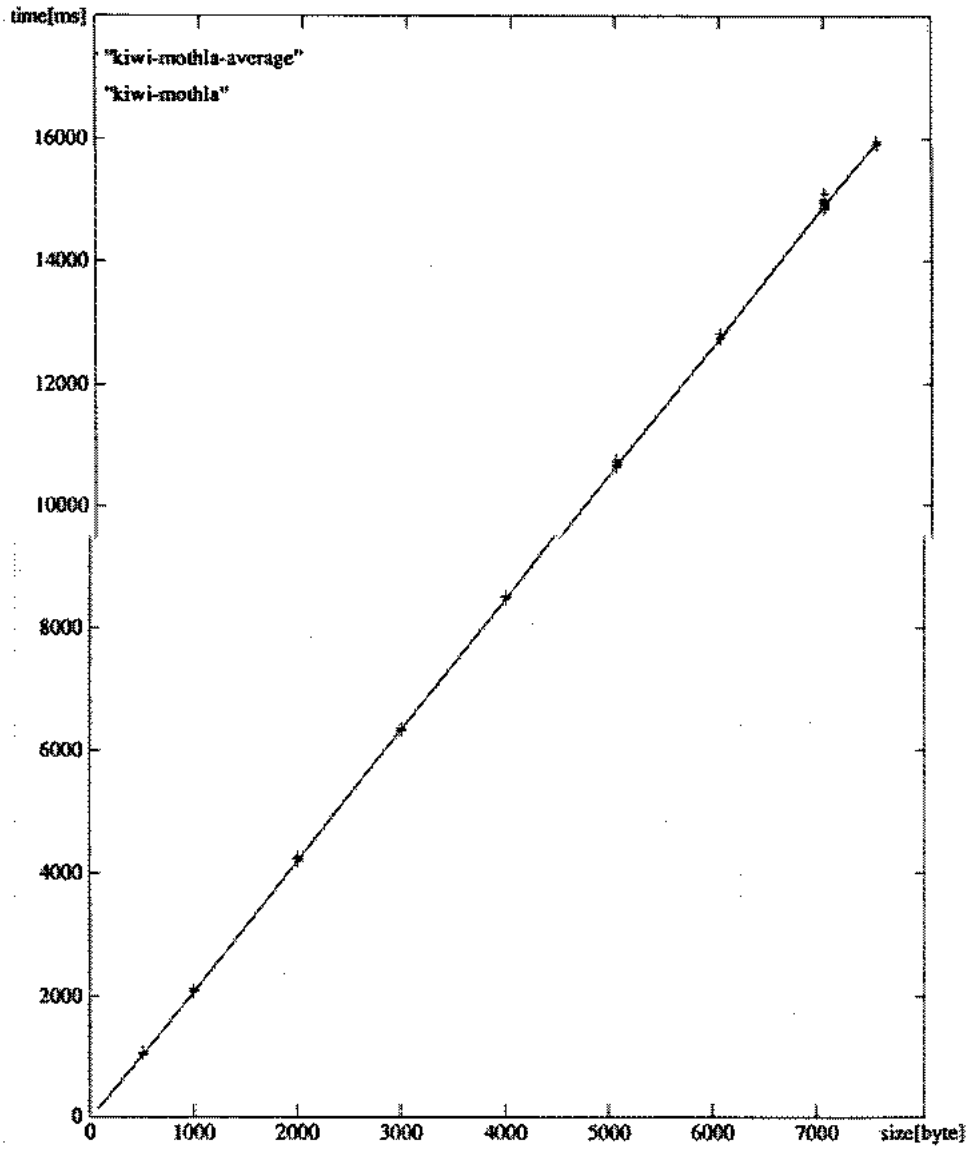


図 A.5: 9.6Kbps シリアル回線の場合

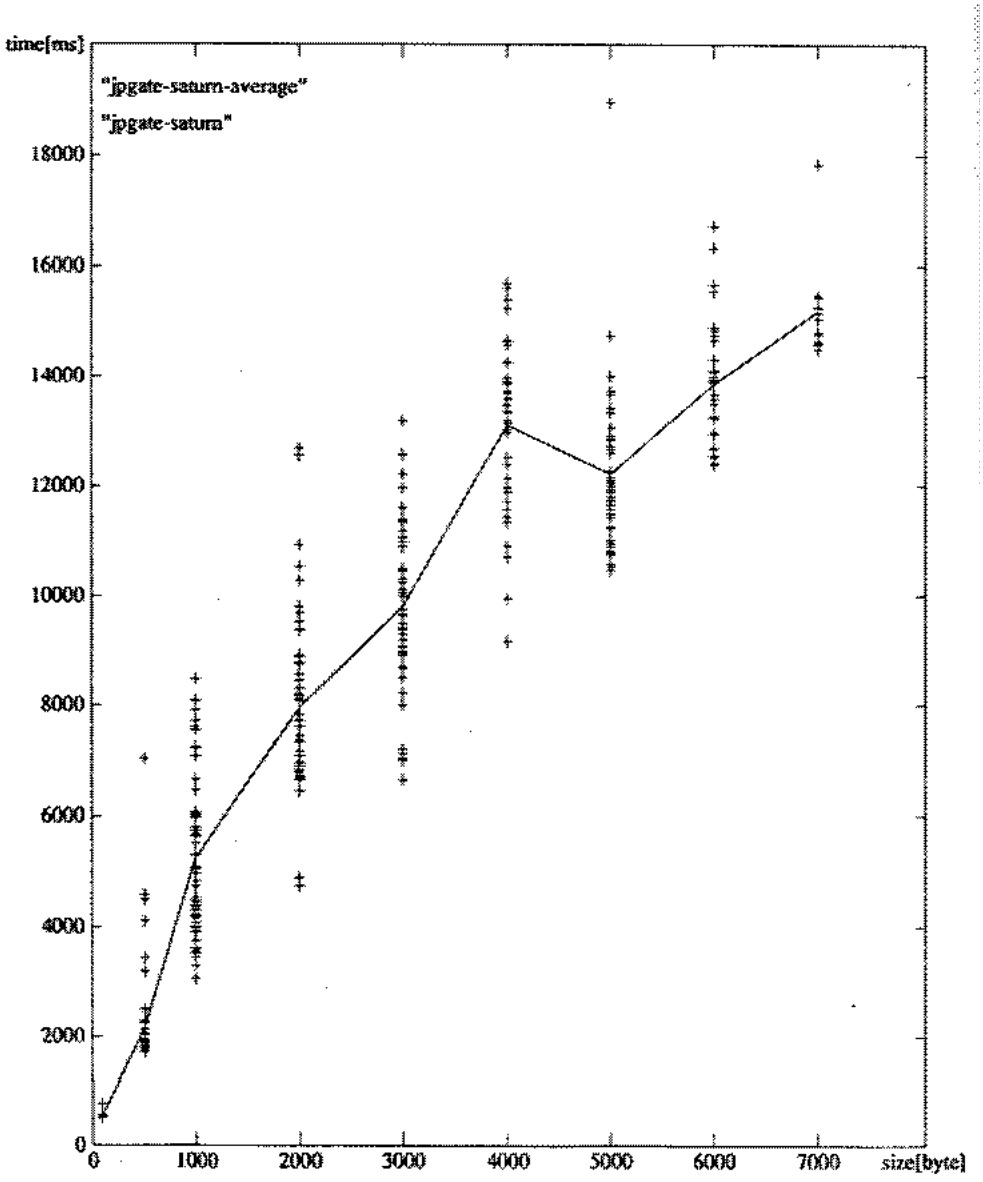


図 A.6: バンド幅の違う回線接続の場合

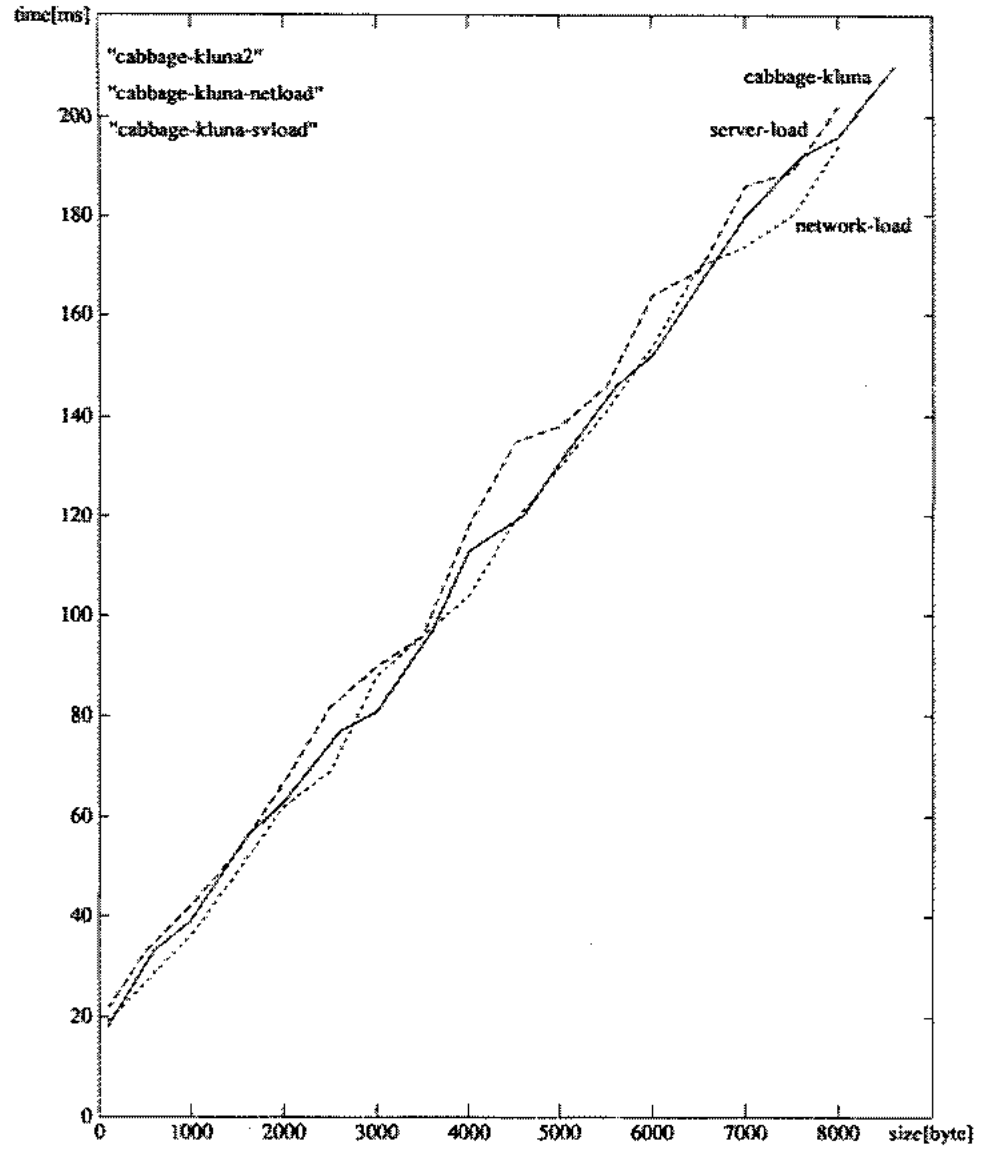


図 A.7: ethernet に負荷をかけた場合

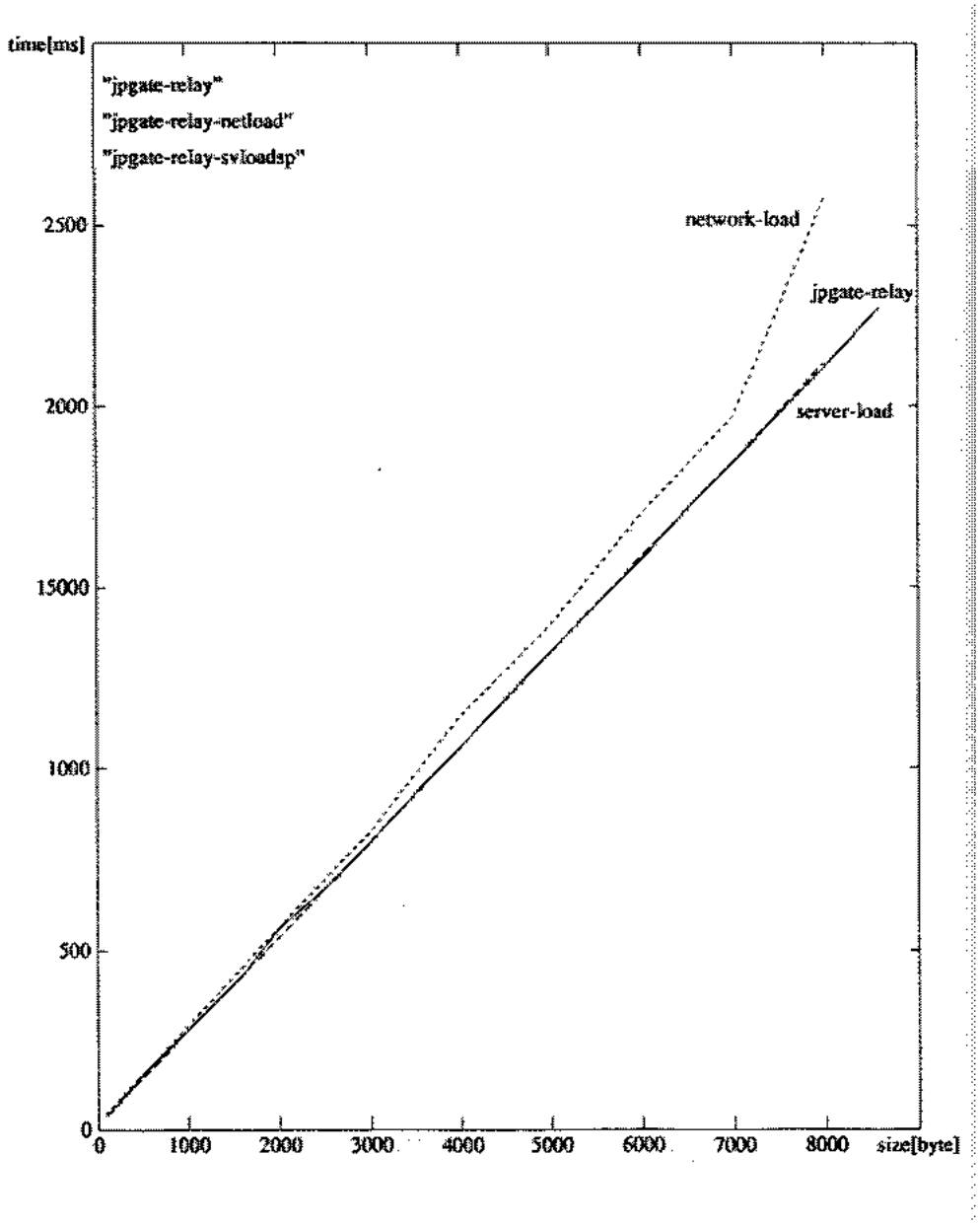


図 A.8: 64Kbps 専用回線に負荷をかけた場合

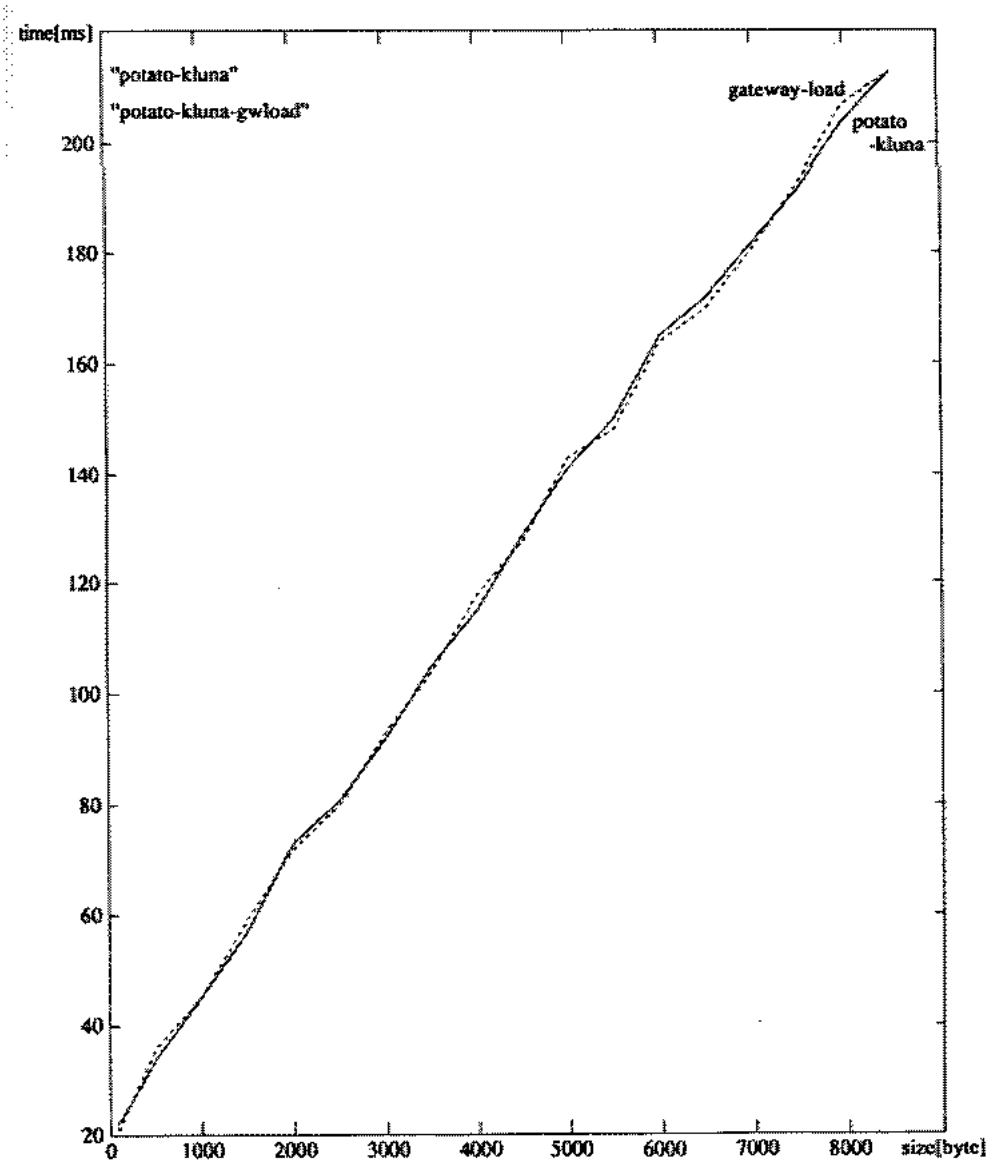


図 A.9: ゲートウェイに負荷をかけた場合 (ethernet での接続)

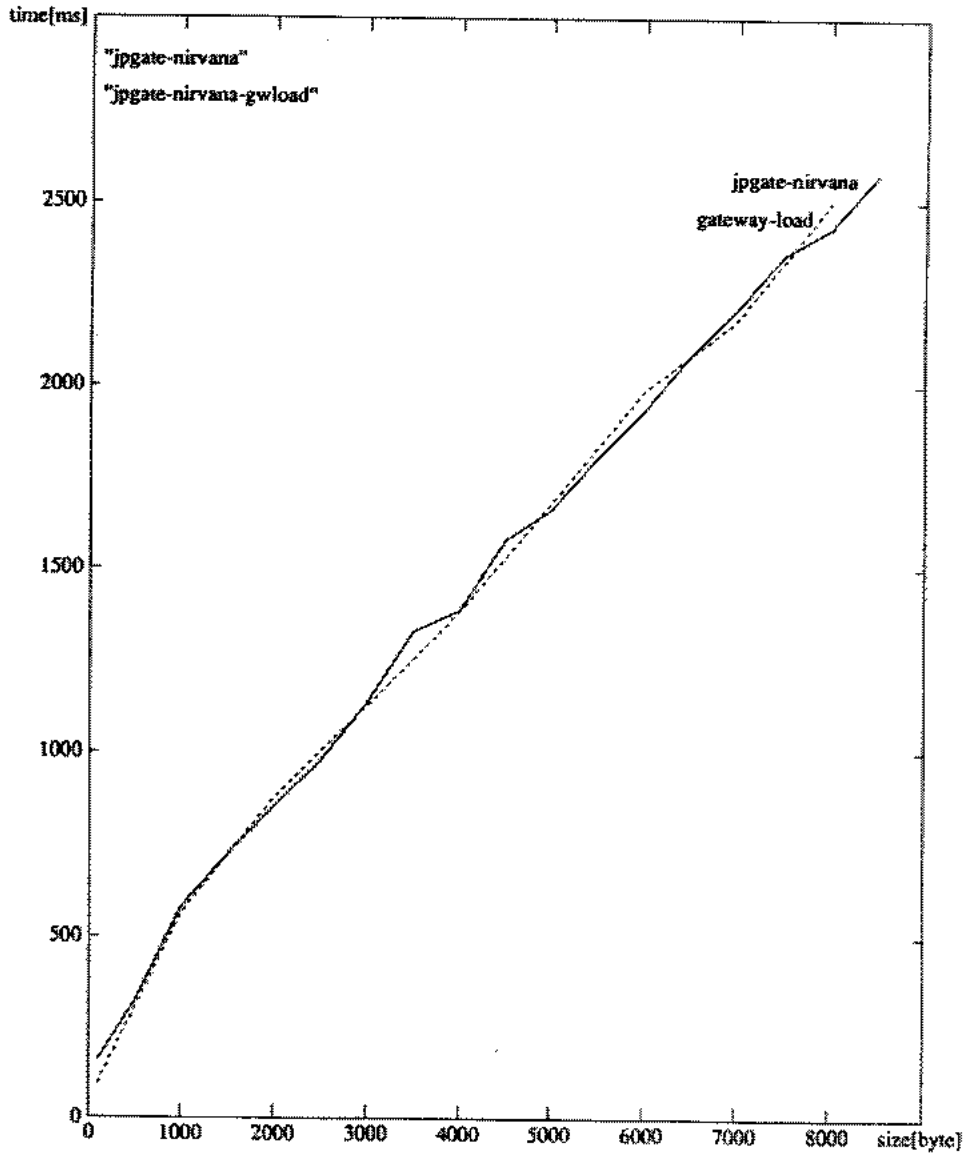


図 A.10: ゲートウェイに負荷をかけた場合 (64kbps 専用回線での接続)

付録 B

結果

```
yasuko*cabbage[43] getsizetime potato
check up !   to potato
4250 byte (20000.ms)
    29.  19.  20.  19.  19.  20.  19.  20.  29.  29.
6250 byte (87.ms)
    29.  30.  29.  30.  29.  30.  39.  39.  30.  29.
7250 byte (117.ms)
    39.  29.  29.  40.  39.  29.  29.  29.  29.  39.
7750 byte (120.ms)
    39.  39.  30.  39.  39.  50.  49.  39.  30.  39.
8000 byte (150.ms)
    39.  39.  39.  40.  39.  39.  40.  39.  30.  39.
max data size = 8000 byte   RTT = 38 ms   timeout = 41 ms

yasuko*cabbage[50] getsizetime kluna
check up !   to kluna
4250 byte (20000.ms)
    109.  109.  110.  110.  129.  109.  109.  119.  109.  109.
6250 byte (387.ms)
    169.  159.  169.  149.  150.  159.  150.  159.  149.  150.
7250 byte (507.ms)
    179.  170.  169.  169.  169.  170.  179.  179.  179.  189.
7750 byte (567.ms)
    189.  180.  199.  189.  249.  180.  189.  199.  189.  219.
8000 byte (747.ms)
    189.  190.  189.  190.  209.  199.  189.  189.  189.  189.
max data size = 8000 byte   RTT = 192 ms   timeout = 211 ms
```

```
yasuko*cabbage[52] getsizetime colt
check up !   to colt
4250 byte (20000.ms)
    49.  39.  39.  40.  39.  39.  39.  39.  39.  49.
6250 byte (147.ms)
    50.  59.  49.  49.  49.  50.  50.  59.  59.  59.
7250 byte (177.ms)
    60.  59.  60.  59.  59.  69.  69.  60.  69.  60.
7750 byte (207.ms)
    69.  60.  69.  69.  60.  60.  69.  69.  69.  69.
8000 byte (207.ms)
    79.  69.  60.  69.  69.  60.  69.  69.  69.  70.
max data size = 8000 byte   RTT = 68 ms   timeout = 74 ms

yasuko@potato[9] getsizetime kluna
check up !   to kluna
4250 byte (20000.ms)
    159.  120.  119.  120.  119.  119.  119.  139.  120.  119.
6250 byte (477.ms)
    199.  179.  159.  160.  159.  180.  200.  180.  200.  179.
7250 byte (600.ms)
    200.  199.  199.  199.  199.  179.  179.  180.  179.  219.
7750 byte (657.ms)
    219.  219.  199.  239.  180.  199.  200.  199.  199.  220.
8000 byte (717.ms)
    220.  239.  219.  219.  219.  219.  239.  200.  219.  219.
max data size = 8000 byte   RTT = 221 ms   timeout = 243 ms
```

```
yasuko*kluna[34] getsizetime knight
check up !   to knight
4250 byte (20000.ms)
    *<5>  *<5>
2250 byte (20000.ms)
    66.  49.  49.  49.  49.  49.  49.  49.  49.  49.
3250 byte (198.ms)
    *<5>  *<5>
2750 byte (198.ms)
    49.  49.  49.  49.  49.  49.  49.  49.  49.  49.
3000 byte (147.ms)
    *<5>  *<5>
2875 byte (147.ms)
    49.  49.  49.  49.  66.  66.  49.  49.  49.  49.
max data size = 2875 byte   RTT = 52 ms   timeout = 57 ms

yasuko*jp-gate[10] getsizetime relay.cc.u-tokyo.ac.jp
check up !   to relay.cc.u-tokyo.ac.jp
4250 byte (20000.ms)
    1130.  1129.  1129.  1120.  1130.  1129.  1119.  1130.  1129.  1119.
6250 byte (2260.ms)
    1649.  1650.  1649.  1650.  1649.  1650.  1649.  1649.  1660.  1649.
7250 byte (3320.ms)
    1920.  1900.  1910.  1909.  1920.  1900.  1910.  1899.  1909.  1909.
7750 byte (3840.ms)
    2030.  2039.  2040.  2030.  2040.  2139.  2030.  2029.  2030.  2039.
8000 byte (4278.ms)
    2100.  2099.  2099.  2110.  2100.  2100.  2100.  2090.  2100.  2099.
max data size = 8000 byte   RTT = 2099 ms   timeout = 2308 ms
```

```
yasuko*jp-gate[11] getsizetime nirvana.cs.titech.ac.jp
check up !   to nirvana.cs.titech.ac.jp
4250 byte (20000.ms)
    2709.  1660.  1539.  1690.  1599.  1610.  1470.  1529.  1530.  1800.
6250 byte (5418.ms)
    2050.  2129.  2160.  2109.  2210.  2239.  2099.  2088.  2169.  2290.
7250 byte (4580.ms)
    2300.  2280.  2240.  2240.  2350.  2620.  2380.  2620.  2590.  2729.
7750 byte (5458.ms)
    2809.  2930.  2910.  2410.  2500.  2800.  2770.  2480.  2630.  2369.
8000 byte (5860.ms)
    2439.  2438.  2450.  2410.  2430.  2760.  2580.  2639.  2700.  2720.
max data size = 8000 byte   RTT = 2556 ms   timeout = 2811 ms

yasuko*ccut[12] getsizetime saturn.center.osaka-u.ac.jp
check up !   to saturn.center.osaka-u.ac.jp
4250 byte (20000.ms)
    12560.  13220.  10860.  14260.  13400.  12300.  13140.  13100.  13000.  11520.
6250 byte (42780.ms)
    *<5>  *<5>
5250 byte (42780.ms)
    14700.  13400.  13960.  12980.  11840.  15280.  *<5>  17220.  14240.  13860.
5750 byte (51660.ms)
    *<5>  *<5>
5500 byte (51660.ms)
    *<5>  *<5>
5375 byte (51660.ms)
    14680.  *<5>  *<5>
5312 byte (29360.ms)
    *<5>  *<5>
5281 byte (29360.ms)
    *<5>  *<5>
5265 byte (29360.ms)
    *<5>  13920.  13160.  13180.  11560.  12580.  14820.  11420.  14960.  11700.
max data size = 5265 byte   RTT = 13033 ms   timeout = 14336 ms
```

```
yasuko*jp-gate[14] getsizetime saturn.center.osaka-u.ac.jp
check up !   to saturn.center.osaka-u.ac.jp
4250 byte (20000.ms)
    15229.  10359.  *<5>  *<5>
2250 byte (15229.ms)
    8098.  7150.  9860.  8859.  8420.  9710.  14478.  7710.  9920.  9609.
3250 byte (28956.ms)
    11930.  *<5>  10910.  12190.  10279.  10380.  10980.  8368.  7750.  9830.
3750 byte (24380.ms)
    8759.  8450.  10420.  9049.  8378.  8810.  9900.  8279.  8810.  10600.
4000 byte (21200.ms)
    8928.  9050.  10740.  8919.  9120.  10480.  9289.  9639.  9920.  8549.
max data size = 4000 byte   RTT = 9463 ms   timeout = 10409 ms
```