平成 11 年度

佐賀大学大学院工学系研究科情報科学専攻修士論文

IPv6 ネットワークの構築とルーティングヘッダを用いた経路制御アプリケーションの実現

Operation of an IPv6 network and a development of an application with an active routing function using routing headers.

論文提出者　大谷 誠 (98SM48)

指導教官　近藤 弘樹 教授
渡辺 健次 助教授

専攻主任　上原 健 教授

論文提出日　平成 12 年 2 月 1 日

目 次

図 目 次

表 目 次

第 1 章

はじめに

本章では、本研究のテーマである、次世代インターネットプロトコル IPv6 のネットワーク構築および、ルーティングヘッダを用いたアプリケーションの実現とその検証を行った背景を説明するとともに、研究の目的について述べる。

1.1 研究の背景

現在のインターネットの仕組みを支えている Internet Protocol version 4 (以下、IPv4) などの一連のインターネットプロトコル群は、1970 年代に開発された。それから約 30 年たった現在、インターネットは、音声やビデオ等のリアルタイム通信、World Wide Web、電子商取引など多くの技術が登場し始めた。またインターネットがここ数年で一般家庭にまで普及し、誰もがインターネットを使用できるような環境になり始めた。

これらの急速なインターネットの普及によって、もはや一世代前のプロトコルとも言える IPv4 では対応し難い問題がいくつか浮上してきた。例えばこの IPv4 では、通信相手の識別を行うために IPv4 アドレスという 32 ビットの整数値を用いる。32 ビットという大きさは 0 から $2^{32} \simeq 4 \times 10^9$ までの符号なしの整数を表すことができ、一見広大な空間のように感じられる。しかしながら IPv4 アドレスの割り当ては、ネットワークトポロジに対してランダムに行われていることや、アドレス・クラスという概念による大雑把な割り当ても影響し、経路制御表の爆発的な増加、アドレス空間の枯渇などの問題が起きてきた。その他にも上記に述べた音声やビデオ等のリアルタイムの効率的な配送、電子商取引などのセキュリティへの対応が難しくなりつつある。よって将来、インターネットに要求されるであろうサービスの実現が不可能になる恐れもある。このような上記の IPv4 の問題を解決し、新たなインターネットへの要求に対応するために、新たに開発されたプロトコルが次世代インターネットプロトコルである Internet Protocol version 6 (以下 IPv6) である。

この IPv6 は、IPv4 の弱点を補い、かつインターネットに新たな仕組みを導入することを目的とし、IETF(Internet Engineering Task Force) によって提案された。そして現在、IPv6 の仕様はほぼ確定し、各研究機関や企業において実用化に向けて活発に研

究、製品開発が進められている。また APNIC などが IPv6 アドレスの正式な割り当てを開始し、JPNIC においても 2000 年 1 月より割り当てが開始された。このような背景から IPv6 は今後、インターネットの主要プロトコルとなり、現在の IPv4 ネットワークが IPv6 へと移行していくことは、ほぼ間違いはないといえる。

1.2 本研究の目的

まず本研究では、IPv6 の仕様の検証、IPv4 からの移行技術の蓄積等を目的として、実際に IPv6 ネットワークを構築し運用を行った。

ここ数年の急速なインターネットの普及は目を見張るものがある。これによりインターネットは様々な品質やサービスを持つ回線が混在したネットワーク環境になりつつある。このような状況下において、性質の異なる通信経路が複数存在するため、ユーザやアプリケーションがデータの通信経路を能動的に選択できるメカニズムが必要となってくると考えられる。そこで本研究では、次世代インターネットプロトコルである IPv6 (Internet Protocol version 6) の拡張ヘッダとして定義されている “ルーティングヘッダ” を用いて、このようなアプリケーションを作成し、検証を行った。

本論文では、第 2 章で IPv6 の仕様と現状について述べる。次に第 3 章で本研究において構築した IPv6 ネットワークとその運用について述べる。次に第 4 章ではルーティングヘッダの概要とそれを用いた経路制御について述べ、第 5 章で実際に作成したルーティングヘッダを用いた経路制御アプリケーションについて述べ、第 6 章でこのアプリケーションの検証実験および考察について述べる。最後に第 7 章でまとめを述べる。

第 2 章

次世代インターネットプロトコル IP version 6

2.1 次世代インターネットの開発過程

次世代インターネットプロトコルが備えるべき機能を決定するために、1993 年 12 月に RFC1550 “IP: Next Generation (IPng) White Paper Solicitation” [1] が発行された。この RFC は、次世代インターネットプロトコルの選定過程で考慮すべき中心要素を、ネットワーク関係者に求めるものであった。この RFC に対して、セキュリティや大企業ユーザ、ケーブルテレビ業界などから多くの意見解答が寄せられた。

これらの意見を受けて RFC1726 “Technical Criteria for Choosing IP The Next Generation (IPng)”[2] において次世代インターネットプロトコルの評価作業において使用すべき基準が 17 項目定められた。そのいくつかを以下に述べる。

- 規模

  少なくとも 10^{12} 個のエンドシステムと 10^9 個のネットワークを識別し、個別にアドレスを指定できなければならない。

- 移行

  IPv4 から堅牢な移行計画を提示できなければならない。

- 安全な運用

  セキュリティ的に安全なネットワーク層がなければならない。

- 固有名

  全ての IP 層オブジェクトに対して大域的名称、偏在的名称、インターネット全体に対して、ユニークな名称を割り当てることができなければならない。

- 拡張性

  拡張可能でなければならず、将来のインターネットサービスに対するニーズを満たせるように成長できなければならない。また成長の過程において同一ネットワーク上にいくつものバージョンが共存できなければならない。

これらの基準をもとにし、1995 年 1 月に RFC1752 “The Recommendation for the IP Next Genereation Protocol”[3] として 3 つの提案が公開された。これには RFC1726 で提示された評価基準に基づいた評価も記述されている。この 3 つの提案はそれぞれ CATNIP (Common Architecture for the Next Generation Internet Protocol)、SIPP (Simple Internet Protocol Plus)、TUBA (TCP/UDP with Bigger Address) である。

この 3 案とも、いくつかの問題点を抱えていたが、SIPP が 128 ビットの IP アドレスを採用することや、その他の問題に対応することを条件に、この案が採用された。最終勧告はこの修正版 SIPP を中心とし、TUBA 案が提唱する自動設定機能と IPv4 からの移行上の考慮、本研究で用いている “ルーティングヘッダ” を盛り込んだ内容となり、これを IPv6 と呼ぶこととなった。現在 IPv6 の最新仕様は RFC2460 “Internet Protocol,Version 6 (IPv6) Specificaton”[4] として公開されている。

2.2 IPv6 の概要

現在、世界的に使用されている IPv4 を見直すきっかけとなったものはアドレス空間の問題によるものだったが、その他にもリアルタイム通信のサポートやセキュリティの強化といった内容も重要な意味を持つため、IPv6 ではアドレス空間の拡大だけでなく様々な機能拡張が行われている。

以下にその特徴を示す。

- ヘッダの簡略化

  IPv6 のヘッダは、IPv4 で有効活用できなかったフィールドが削除され、簡略化が行われている (図 2.1)。また経路上でのフラグメント機能の削除なども行われ、高速な転送が可能となっている。

- アドレス空間の拡大

  IPv4 のアドレス空間の枯渇に伴い、IPv6 ではアドレス長を 128 ビットに拡張した。これにより、アドレス空間の制約を受けずにアドレスを付加することができ、今後インターネットへの接続が予想される携帯機器なども容易にネットワークへ接続できると考えられる。

- 経路の集約を考慮したアドレス構造

  IPv6 のアドレス構造には、階層構造を意識した割り当てを行い、経路制御を効率的に行うことを目指している。現時点では、Aggregatable Global Unicast Address と呼ばれるアドレス体系が定義されている。

- セキュリティ機構
  IPv6では、パケットの認証や暗号化の機構が組み込まれており、インターネットの弱点といわれているセキュリティの強化がなされている。必要に応じてアプリケーションが通信セキュリティを確保でき、盗聴、改竄、なりすましなどを防止することが可能となる。

- Plug & Play 機能
  ホストをネットワーク接続する際、アドレス等の設定を自動で行う機能を標準で装備している。これによりホストをネットワークへ接続するだけですぐに通信を行うことが可能となる。

- リアルタイム通信への対応
  VoD (Video on Demand) やビデオ会議システムなど、リアルタイム性が要求されるトラフィックに対応するため、IPv6ではフローラベルやトラフィッククラスといった情報がヘッダに付加される。これらの情報によりIP層のみでフローの検出が可能となり、高速な処理が可能となる。

- 拡張ヘッダ
  IPv6の機能拡張は、それぞれ個別のヘッダーとして定義され、IPv6ヘッダと別に扱い、送信元から送信先までの経路中で処理を簡略化できる。

2.2.1 IPv6ヘッダ

IPv6のヘッダフォーマットは、上記に示したようにIPv4に比べて簡略化されている。以下にそのヘッダフォーマットを示す。

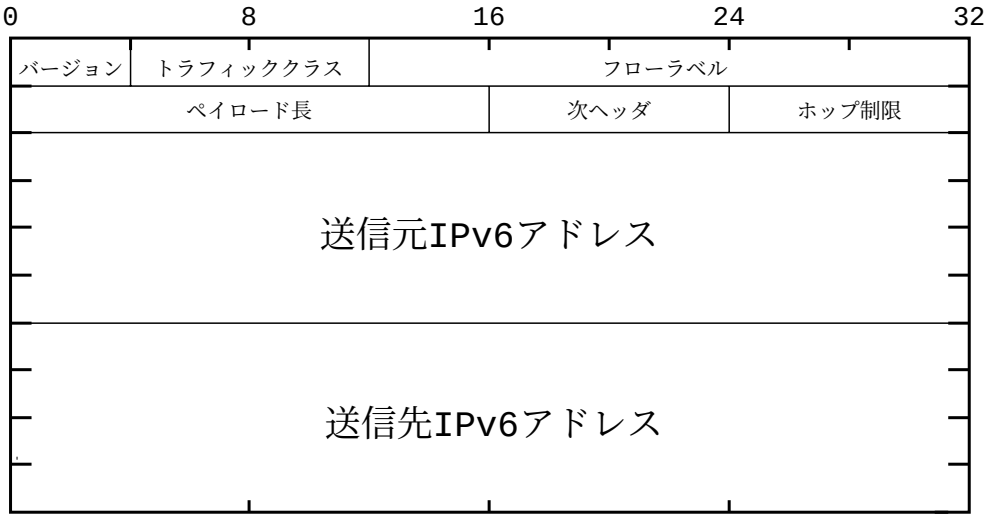


図 2.1: IPv6ヘッダフォーマット

- バージョン
  IP のバージョンを表すフィールドで、フィールド長は 4 ビット、値は 6 である。唯一このフィールドだけが IPv4 と場所及び用途が同じである。

- トラフィッククラス
  IP のパケットに対して “区分化サービス (Differentiated Service: Diffserv)” を行うためのフィールドであり、サイズは 8 ビットである。これは IPv4 の TOS(Type of Servive) と同様な機能を提供するために提案されたものである。しかしながら現状においてこのフィールドを用いた具体的なサービスは提案されていない。

  また FRC2460 においてトラフィッククラスに対する挙動は以下のように定義されている。

  - ノード中の IPv6 サービスを提供するサービスインターフェイスはトラフィッククラスのビット情報をそのまま上位プロトコルに渡さなければならない。
  - トラフィッククラスのビットを利用するノードは、その利用目的においてパケットの生成、転送、受取りの段階で値を変更しても良い。用途不明な場合や必要としない場合は、無変更、または無視しなければならない。
  - 上位プロトコルは、送信元が同じパケットのトラフィッククラスのビットの値が同じであると仮定しなければならない。

- フローラベル

  このフィールドは、通常時とは異なるサービス品質を要求する場合や、リアルタイム通信を行う際に、パケットに対する特別な処理を送信元で示す際に用いるものである。

  IPv6 仕様の第 1 版となる RFC1883[5] ではこのフィールドは 24 ビットだったが、トラフィッククラスフィールドの拡張のため 20 ビットとなった。IPv6 において特別な扱いをしなくてはならないリアルタイム通信などのサービスではこのフィールドを使ってフローの識別などを行う。フローラベルをサポートしないノードは 0 指定し、受信時は無視し、生成には乱数を用いる。フローの識別は送信元アドレスと 0 以外のフローラベルによって識別される。

  このフィールドの利用方法については、トラフィッククラスフィールドと同様に研究が続いている段階にある。よって今後変更される可能性もある。

- ペイロード長

  このフィールドは 16 ビットでオクテット単位で表したペイロード長を示している。このペイロード長は、IPv6 の基本ヘッダより後ろの部分を示しているため、拡張ヘッダや TCP や UDP などの上位層のプロトコルのヘッダサイズも含まれている。

- 次ヘッダ

  このフィールドは IPv6 ヘッダの直後に続くヘッダの種類を示すフィールドで、フィールド長は 8 ビットである。使用される値は IPv4 のフィールドと同じもので IPv6 固有なものなども追加されている。またこの次ヘッダに相当するものが拡張ヘッダの中にも存在し、図 2.2 のように、次に続くヘッダの識別子を順に記述する。

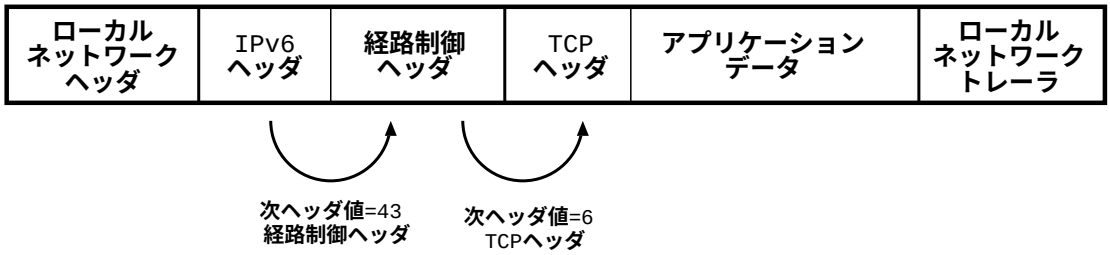


図 2.2: 次ヘッダフィールドによるヘッダの識別

- ホップ制限

  このフィールドは、パケットがノードを通過するごとに 1 減少し、この値が 0 になるとこのパケットは破棄される。

- 送信元 IPv6 アドレス

  このフィールドはパケットの送信元の IP アドレスを示しており、フィールド長は 128 ビットである。

- 送信先 IPv6 アドレス

  このフィールドは、パケットの受取手のアドレスを示している。ここで示されている IP アドレスが必ずしも最終的なアドレスではなく、拡張ヘッダであるルーティングヘッダが使用されている場合は、中継ノードを示していることがある。

2.2.2 拡張ヘッダ

IPv6 では、IPv4 ヘッダの中で用いられていたオプションのフィールドを、拡張ヘッダとして定義し直すことにより構造を単純化している。これによって不必要な処理に起因するオーバーヘッドによってパケットの処理速度の低下を招くことはない。また構造が単純化しているにも関わらず、IPv4 より複雑な要求にも対応できるように設計されている。

以下にこの拡張ヘッダについて示す。

- オプションヘッダ (中継点、終点)

  中継点オプションヘッダ、終点オプションヘッダは、それ自身がパラメータや通信を詳細に定義するためのオプションを含むヘッダである。

- ルーティングヘッダ

  このヘッダは、始点から終点への経路上のパケットが経由するノードのリストを含んでおり、パケットの通信経路を制御したいときに用いる。このルーティングヘッダを用いることで、アプリケーションからの能動的な経路制御が実現可能である。これについては第 4 章以降において詳しく説明する。

- 断片ヘッダ

  送信したいデータが巨大で、経路の MTU (Muximum Transmission Unit, 最大伝送単位) に収まらない場合、このヘッダを用いてパケットを分割することによりデータの送信可能にするためのヘッダである。

- 認証ヘッダ

  このヘッダは RFC2404 (Authentication Header)[6] で定義されており、現在提案されている認証ヘッダの機能は、パケット全体の完全性およびデータの認証を付加するものである。また繰り返し行われる攻撃に対しての保護機能も提供している。

- 暗号ペイロードヘッダ

  このヘッダは、RFC2402 (Encapsulationg Security Payload)[7] で定義されており機密性、認証、整合性などを実現するよう設計されている。

- 次ヘッダなし

  ヘッダの次ヘッダフィールドの値が 59 のとき、そのヘッダがデータの最後となり、この後に続くヘッダは何も存在しない。このようなことからこのヘッダは、次ヘッダなし (No Next Header) と呼ばれている。

第 3 章

IP version 6 実験ネットワークの構築とその運用

ここではわれわれが構築を行った IPv6 実験ネットワークの構成と、このネットワークの運用について述べる。

3.1 IPv6 実験ネットワークの構築

IPv6 で通信を行うためには、

- OS (オペレーティング・システム)
- アプリケーション
- ルータなどの通信機器

この 3 つが IPv6 に対応する必要がある。このため現在多くの組織が、各種 OS やルータのための IPv6 実装を公開している [8]。 以下に OS への実装の代表的なものを示す。

- Windows NT
  - Microsoft Research
  - Toolnet6 (日立)
- BSD 系 OS
  - KAME
  - INRIA Rocquencourt
  - NRL (US Naval Research Laboratory)
  - IPv6 daemon

- Linux
  - NRL をもとに kernel 2.1,2.2 に実装
- Mac
  - Mentat Inc.

本研究において IPv6 の実験ネットワークを構築、およびアプリケーションの開発を行うにあたり、BSD 系の OS である FreeBSD 上で動作する上記の KAME を用いた。この KAME と呼ばれる実装は、日本の 7 つの企業が共同した KAME Project[9] において、BSD 系の OS の IPv6 化を目的として公開しているものである。この実装はもとは Hydrangea と呼ばれていたもので、WIDE Project の v6 分科会によって開発されていたものである。

本研究において、この実装を採用した理由としては、

- 日本での動作実績が高い
- ソースコードの更新が早い
- AT 互換機上で実装が可能
- IPv6 対応アプリケーションが豊富
- IPv6 ルータとして動作可能

などの理由からである。

また、構築したネットワーク上で、Microsoft Research の実装も動作確認した。

3.1.1 ネットワーク構成

構築したネットワークは、ルータ 6 台 (v6rt1 〜 v6rt4、 v6rt6、v6rt7)、ホスト 5 台 (v6ht5、v6ht8 〜 v6ht11) の計 11 台の AT 互換機による構成となっている。また、v6ht8 はノートパソコンであり、ネットワークのほかの部分に接続が可能である。

図 3.1 にこのネットワークの構成図を示す。また、各ルータやホストの性能、使用 OS などは、表 3.1 のとおりである。

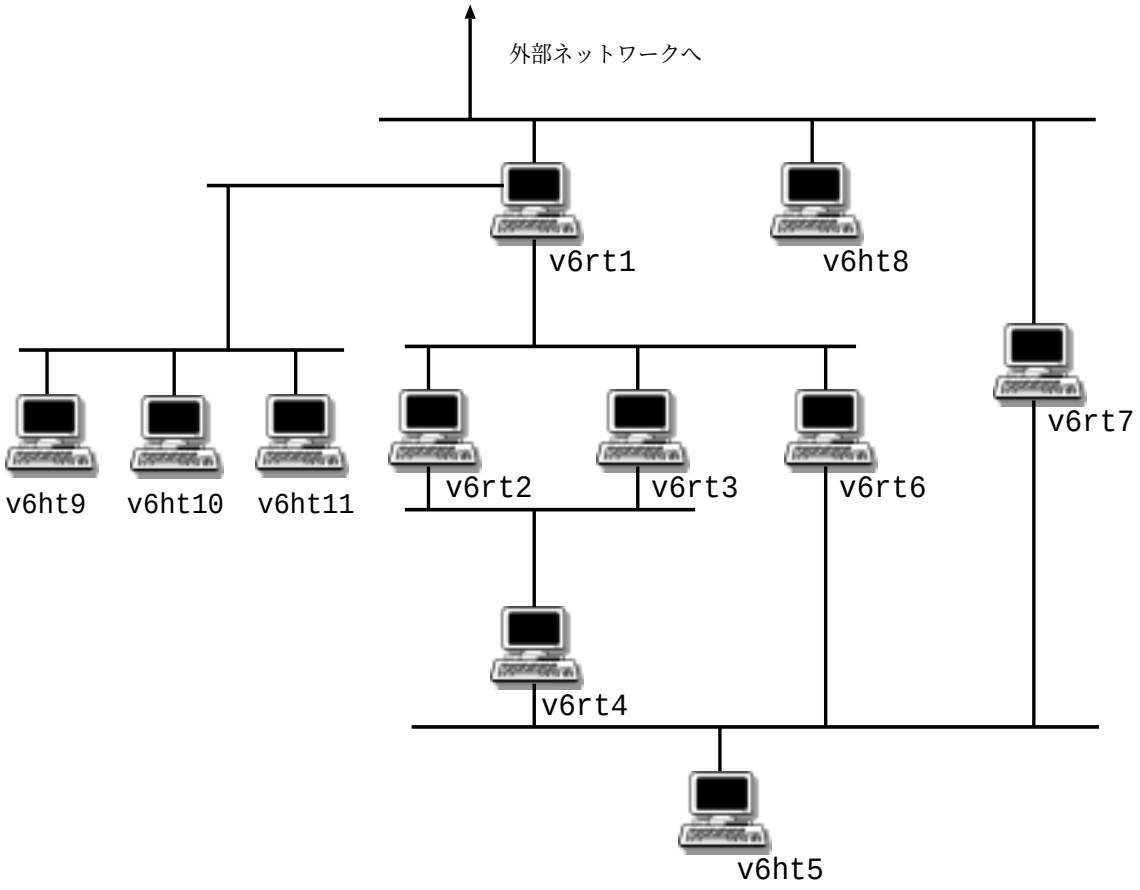


図 3.1: IPv6 実験ネットワーク構成図

マシン名	CPU	メモリ	OS
v6rt1	Pentium II 450MHz	256MB	FreeBSD 2.2.8 + KAME
v6rt2	Pentium II 450MHz	256MB	FreeBSD 2.2.8 + KAME
v6rt3	Pentium II 450MHz	256MB	FreeBSD 2.2.8 + KAME
v6rt4	Pentium II 450MHz	256MB	FreeBSD 2.2.8 + KAME
v6ht5	Pentium II 300MHz	128MB	FreeBSD 3.3 + KAME
v6rt6	Pentium II 450MHz	256MB	FreeBSD 2.2.8 + KAME
v6rt7	Pentium 100MHz	48MB	FreeBSD 2.2.8 + KAME
v6ht8	pentium 133MHz	40MB	FreeBSD 2.2.8 + KAME+ PAO
v6ht9	Pentium II 450MHz	256MB	Windows 2000 RC2 + Microsoft Research
v6ht10	Pentium II 450MHz	256MB	Windows 2000 RC2 + Microsoft Research
v6ht11	Pentium II 300MHz	128MB	Windows NT4 + Microsoft Research

表 3.1: IPv6 実験ネットワークのルータ及びホストの詳細

3.1.2 ネットワークの設定

本ネットワークにおいて、ネットワークの設定は、ルータとホストによって異なっている。以下にその違いを示す。

3.1.2.1 ルータにおけるネットワーク設定

ルータの場合、グローバルユニキャストアドレスの設定は、ルータが所属しているネットワークのプレフィックスを静的に設定し、そのプレフィックスと物理的なインターフェースのアドレスから動的に生成される。またエニーキャストアドレスも同様に、静的に設定したプレフィックスを用いて設定される。

一方、リンクローカルアドレスは、グローバルユニキャストアドレスと同様に物理的なインターフェースのアドレスと、リンクローカルアドレスのプレフィックスより動的に生成される。

またホストが経路情報を設定するために必要な情報を、経路上に配送するための “rtadvd” (router advertisement daemon) も同時に起動している。

経路情報は、KAME に附属する “hroute6d” と呼ばれる RIPv6 の daemon を起動することによって管理を行っている。

3.1.2.2 ホストにおけるネットワーク設定

ホストの場合は Plug&Play 機能により、ネットワークの設定は特に静的に行う必要はなくグローバルなアドレスはルータから rtadvd によって提供される情報をもとに動的に設定される。リンクローカルアドレスもルータと同様に動的に設定される。よって v6ht8 ような可搬性のあるホストは、このネットワークの、どの部分に接続してもすぐに通信が可能となる。

3.2 6bone への接続

ここでは、世界的規模で構築されている IPv6 の実験ネットワークにおいて、およびこの実験ネットワークへの接続について述べる。

3.2.1 6bone とは

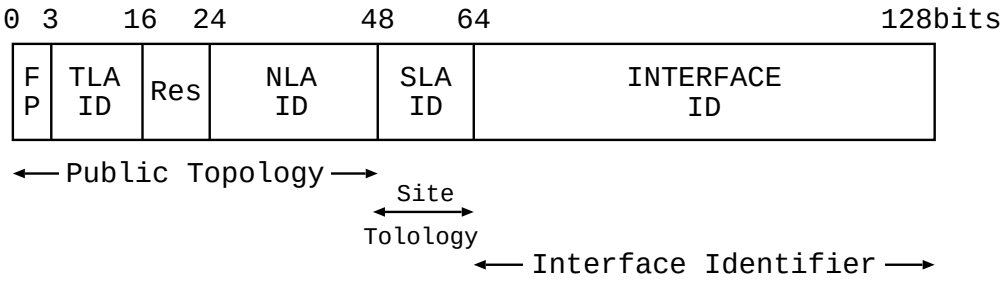
6bone とは世界的規模で構築、運営されている IPv6 のテストベットネットワークである。ここではテストベッドネットワーク用のアドレスを使用し、IPv6 ネットワークの運用、およびそれら問題点の発見などを行っている。6bone は実験を目的としたネットワークで、IPv6 の研究を行っている組織の接続を受け付けており、2000 年 1 月現在で 42ヶ国・500 組織以上が参加している。

研究組織が 6bone に接続を行う場合、現在は 6bone から直接アドレスを取得する方法と、既に 6bone に直接接続している組織からアドレスを取得する方法がある。前者は 6bone から pTLA と呼ばれる割り当てを受けるということであり、後者は pTLA を取得した組織からアドレス割り当てを受けるということである。

本研究では、pTLA を取得して 6bone に接続している WIDE Project の IPv6 テストベッドネットワーク、6bone JP への接続を行った。

3.2.2 6bone における IPv6 アドレス構造

6bone で用いられている IPv6 アドレスは、集約可能グローバルユニキャストアドレス (Aggregatable Global Unicast Address) と呼ばれるアドレスをもとにしている。図 3.2 に集約可能グローバルユニキャストアドレスの構造を示す。アドレスは、上位 64bit と下位 64bit に分けられ、下位 64bit は Interface Identifiier として使用する。Public Topology と Site Topology は，上位 64bit で以下のようになっている。



- FP
  Format Prefix (001)
- TLA ID
  Top-Level Aggregation IDentifier
- RES
  REServed for future use
- NLA ID
  Next-Level Aggregation IDentifier
- SLA ID
  Site-Level Aggregation IDentifier
- INTERFACE ID
  INTERFACE IDentifier

図 3.2: 集約可能グローバルユニキャストアドレス

上記の FP と TLA の部分をあわせて、“3FFE” としたテストベッドネットワーク用のアドレスが決められ、この “3FFE” のネットワークを 6bone と呼ぶ。

また RES としている 8bit を TLA と想定し、それを pTLA と呼んでおり、図 3.3 のようなアドレス構造をしている。

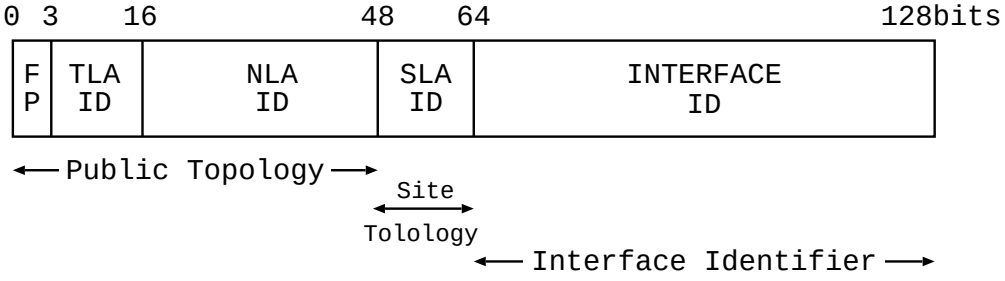


図 3.3: 6bone における IPv6 アドレス構造

3.2.3 6bone JP における IPv6 アドレス構造

WIDE Project のテストベッドネットワークである 6bone JP では、6bone で NLA としている 24bit を 8bit と 16bit に分け、それぞれを NLA1、NLA2 と呼ぶ。アドレス構造は図 3.4 のようになる。

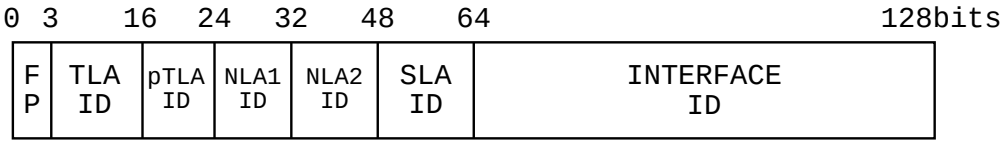


図 3.4: 6bone JP における IPv6 アドレス構造

6bone JP に接続するには、NLA1 を取得して接続する方法と、NLA1 を取得した組織から NLA2 を取得する方法がある。 我々は、NLA1 を取得している NTT 情報流通プラットフォーム研究所の協力をもとに、NLA2 のアドレス空間を取得した。

図3.5に示すようにNTT情報流通プラットフォーム研究所はNLA1として “3ffe:503::/32” を持っており、本ネットワークはNLA2として “3ffe:503:1050::/48” のアドレスブロックを割り当てられている。

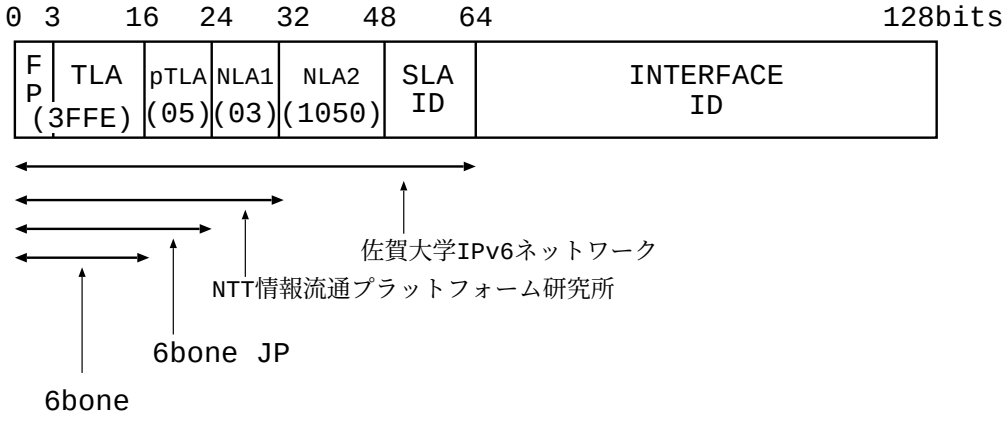


図 3.5: 6bone JP における IPv6 アドレス構造 (NTT)

ここで SLA はサイト内で集約的な経路制御を行うために使用し、INTERFACE ID は、ネットワークインターフェイスの MAC アドレス等から動的に生成されるものである。

3.2.4 トンネリング

実際に IPv6 のアドレスを割り当てられたとしても、接続先までのネットワークが IPv6 に対応していなければ、直接 IPv6 で接続することはできない。この問題に対応するために、IPv6 ではトンネリングによる接続が提案されている。

トンネリングとは、あるネットワーク層のプロトコルを通すために異なるプロトコルを利用する技術である。トンネリングを用いれば、IPv6 パケットの前に IPv4 ヘッダを付加することにより、IPv4 ネットワークを利用して IPv6 パケットを送受信できる。その結果、ネットワーク的に孤立している IPv6 ホスト同士が IPv6 を用いて通信することが可能となる。上記に述べた、本ネットワークと NTT 情報流通プラットフォーム研究所との接続形態はこのトンネリングによるものである。図 3.6 にこの接続におけるネットワークの構成図を示す。

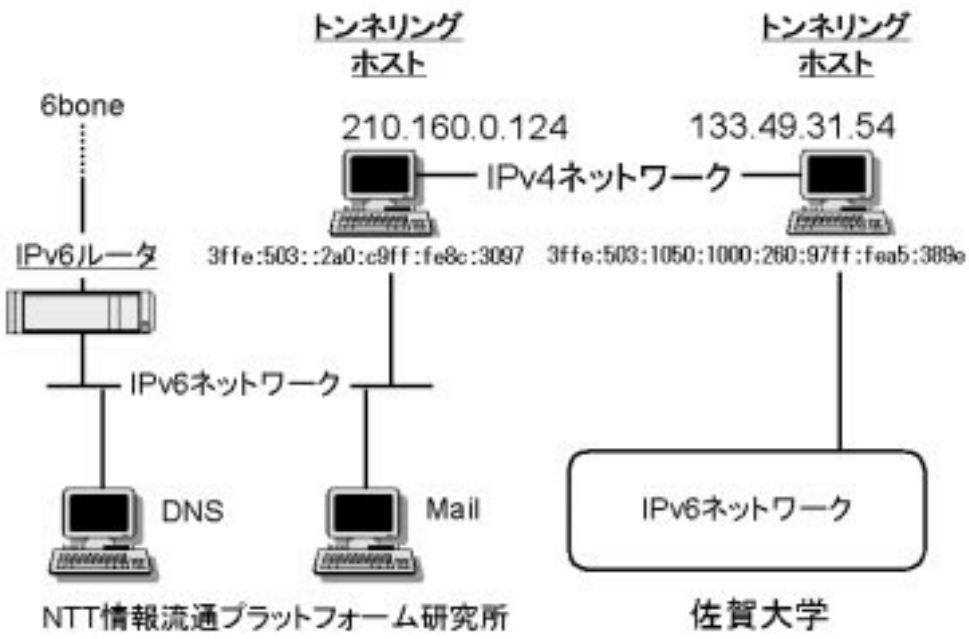


図 3.6: トンネリングによる接続の構成図

3.2.5 マルチホーム環境

マルチホームな環境においての能動的な経路制御の研究を目的として、同じく 6bone JP より NLA1 を取得しているインテック・システム研究所の協力で新たに NLA2 のアドレス空間を取得した。インテック・システム研究所のは NLA1 として “3ffe:508::/32”、本ネットワークにおいては NLA2 として “3ffe:508:9::/48” のアドレスブロックを割り当てられている。接続はトンネリングによる接続形態である。

3.2.6 プロバイダの運用

外部組織のアドレス割当を行うことを目的として、構築したネットワークを用いプロバイダの運用も行っている。このプロバイダ用のアドレス空間は上記のものとは異なり、新たに取得したものである。このアドレス空間は、上記の NTT 情報流通プラットホーム研究所が 6bone から取得した pTLA で、このアドレス空間からプロバイダ運用のため NLA1 相当の “3ffe:1800:d800::/48” アドレスブロックを取得した。

3.2.7 階層的なネットワーク構成

上記のように本研究で行ったネットワークは、複数の IPv6 のアドレスを取得している。また取得した IPv6 アドレスの、外部組織への割当も可能している。本ネットワー

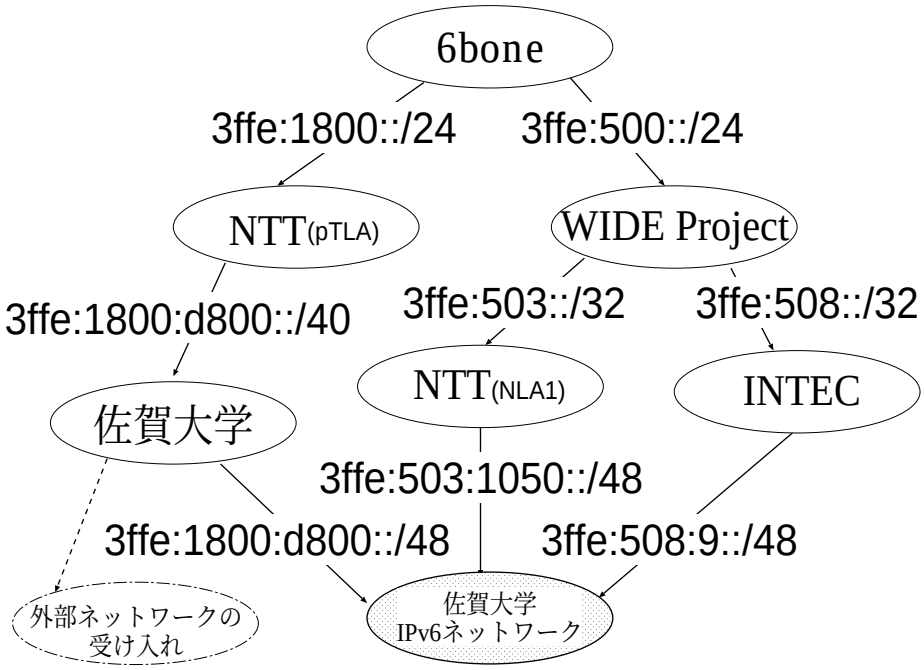


図 3.7: IPv6 ネットワーク構成図

クの IPv6 アドレスの割当構成を図 3.7 に示す。

この図のように IPv6 アドレス割当は、階層構造となる。これは集約可能グローバルユニキャストアドレスに起因するものである。

3.2.8 トランスレータ

IPv4 から IPv6 への移行において、インターネットに接続されているノードが一斉に IPv6 に対応することはまず不可能であり、実際は、かなり長い期間をかけて移行が進むことになる。従って、IPv6 と IPv4 の環境が混在してしまうことになる。

このような環境下で移行を円滑に進めるためには、IPv6 と IPv4 両者のホストが相互に通信が行えることが望まれる。これは我々が構築したネットワークのように、IPv6 ホストとは IPv6 で、IPv4 ホストとは IPv4 を使用するという “デュアルスタック” と呼ばれる実装を行うことで解決できる。我々のネットワークでは、通信先ホストの IP アドレス構造により、IPv6 および IPv4 を判断し、自動的にプロトコルを選択することにより通信が行なわれている。

しかし、これは IPv6 ホストが IPv4 ホストと通信を行うために、IPv6 および IPv4 アドレスの両方を取得する必要があることを意味している。これでは、IPv4 のアドレス枯渇の問題を根本的に解決することにはならず、移行期の後半に IPv4 アドレスが枯渇してしまう可能性も考えられる。

そこで現在、KAME Project では、IPv6 で通信可能なホストから IPv4 ネットワークに IPv6 を使って接続するためのトランスレータを公開している。これはカーネルと協

調して動く TCP relay daemon である。トランスレータを起動しているノードに、接続先が IPv4 互換アドレスであるパケットが届いた場合、接続先の IPv4 アドレスに対して TCP コネクションを確立し、TCP/IPv6 から TCP/IPv4、またその逆への変換を行うことにより通信を可能とする。

このトランスレータは faithd と呼ばれ、現在、telnet、rlogin、ftp などといったアプリケーションをサポートしている。本研究で構築したネットワークにおいてもこのトランスレータを起動し、IPv6 ネットワークからの IPv4 ネットワークへ接続を可能にしている。

3.2.9 IPv4 ノードから IPv6 ノードへの接続

上記で述べたトランスレータは IPv6 ノードから IPv4 ノードへ接続のみが可能であり、IPv4 ノードからの IPv6 ノードへの接続をサポートしていない。

そこで本研究において、この接続をサポートするシステム (コネクションリレープログラム) を作成した。IPv6 ノードから IPv4 ノードへ接続する場合は、IPv4 互換アドレスを用いて IPv6 ノードとの接続が可能である。しかしながら、IPv4 ノードから IPv6 ノードへ接続する場合、IPv4 アドレスを用いて IPv6 アドレスをマッピングする必要がある。

本研究で作成したシステムでは、IPv6 ノードへの接続を行うための IPv4 アドレスを 1 つ定義し、この IPv4 アドレスに対してどの IPv6 アドレスをマッピングするかを、WWW の CGI を用いて設定するという方法を用いた。図 3.8 に WWW のインターフェイス画面を示す。図のように、接続したい IPv6 ノードのホスト名または、IPv6 アドレスを入力することにより、設定が完了する。IPv4 ノードは、IPv6 アドレスをマッピングした IPv4 アドレスに接続を行うことにより、IPv6 ノードとの通信が可能となる。このシステムは、IPv6 ノードと同様に IPv4 ノードに接続も可能である。

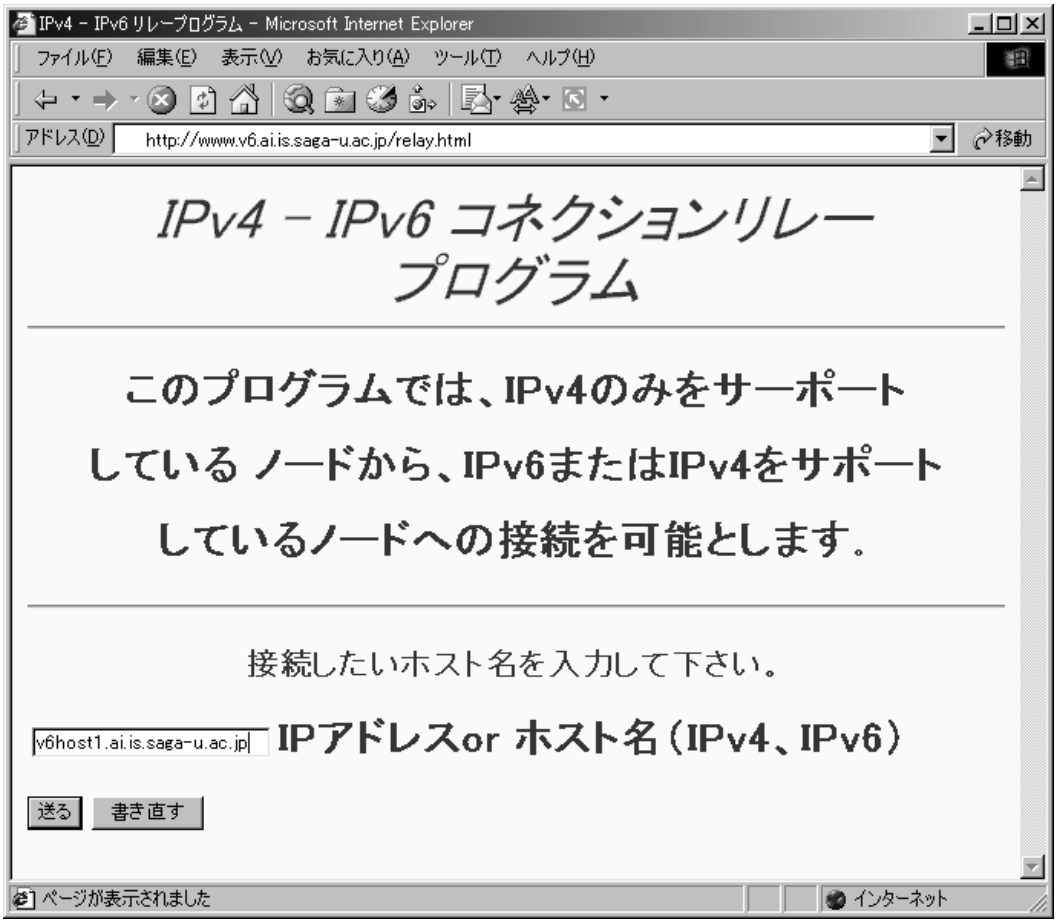


図 3.8: コネクションリレープログラムの接続設定インターフェイス

またこのプログラムは、並行処理を行うことにより同時に複数のコネクションを管理することが可能である。これにより複数のユーザが、同時に異なる IPv6 ノードへ接続することを可能にしている。

図 3.9 にこのシステムの概要を示す。

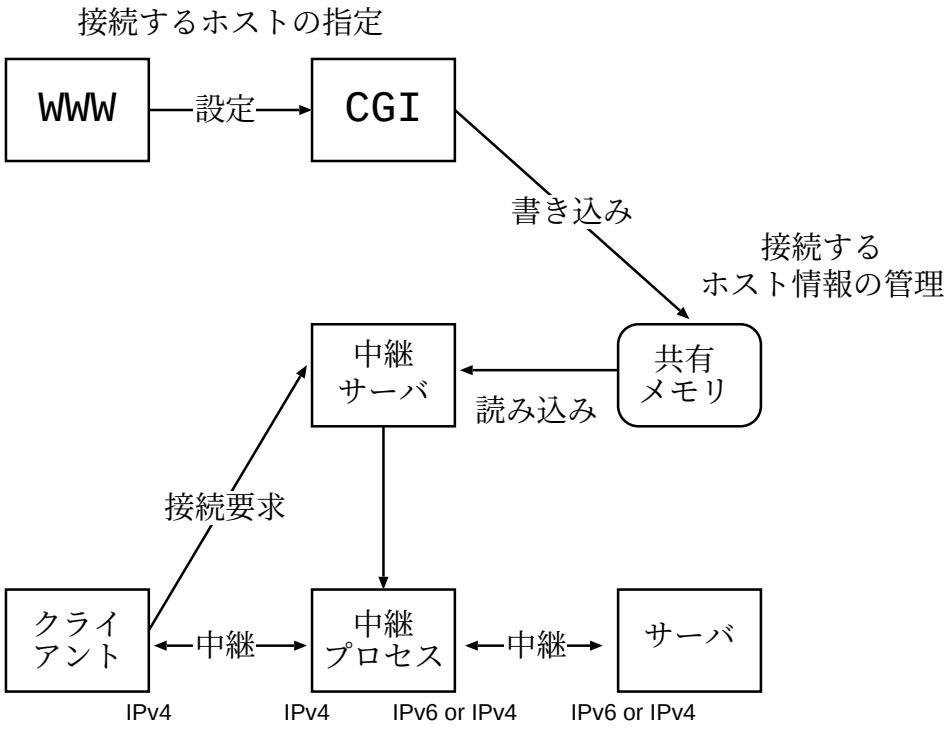


図 3.9: リレープログラム

図のように、ユーザが WWW を用いて接続したい IPv6 ノードを設定し、次に接続中継用のアドレスに接続を行うと、そのコネクションを設定した IPv6 ノードへ中継する。IPv6 ノードの情報は、共有メモリによって管理されている。以上の方法により IPv4 ノードから IPv6 ノードへの接続を行う。

第 4 章

ルーティングヘッダを用いた能動的な経路制御

本章では能動的な経路制御の必要性と、これを実現するために用いたルーティングヘッダについて述べる。

4.1 能動的な経路制御の必要性

近年の急速なインターネットの普及により、これからのインターネットは様々な品質やサービスの回線が混在したネットワーク環境となることが予想される。またユーザからのインターネットに対するセキュリティやマルチメディア、ローコストなどといった新たな要求も生じ始めている。そのほかにも、ネットワークの冗長性の向上や負荷分散を目的としてマルチホームなネットワーク環境を構築する組織も登場している。このような状況下では、性質の異なる通信経路が複数存在するため、ユーザやアプリケーションがデータの通信経路を能動的に選択できるメカニズムが必要となってくる。

4.2 IPv6 におけるルーティングヘッダ

次世代インターネットプロトコルである IPv6 (Internet Protocol version 6) では、アプリケーションからデータの経路を選択可能にする、ルーティングヘッダと呼ばれる拡張ヘッダが定義されている。

第 2 章でも示したように、このルーティングヘッダは、始点から終点への経路上の、パケットが経由する中継ノードのリストを含んでいる。アプリケーションがこのヘッダを用いて経由するルータを指定することで、経路選択機能が実現できる。以下にこのルーティングヘッダのフォーマットを示す。

4.2.1 ルーティングヘッダのフォーマット

- 次ヘッダ (Next Header)

  最初の次ヘッダフィールドには、ルーティングヘッダの直後にくる拡張ヘッダ、または、上位層プロトコルのヘッダの番号が格納される。

- 拡張ヘッダ長 (Hdr Ext Len)

  このフィールドには、拡張ヘッダ長が格納される。8 オクテットを単位としたときの拡張ヘッダ長を表し、かつ拡張ヘッダの最初の 8 オクテットを含まない。

- ルーティングタイプ (Routing Type)

  5 オクテット目以降のフォーマットを指定する識別子が定義される。現時点では、0 以外の値は定義されていない。

- 残りセグメント (Segments Left)

  ソースルーティングを処理してきて、拡張ヘッダ内にあるアドレスリストに示されているノードのうち、まだ通過していないノードがいくつあるかを表す。このフィールドは、ルータでソースルーティングを処理を行うたびに減らされる。残りセグメントの値が 0 になったら、何の意味ももたなくなる。

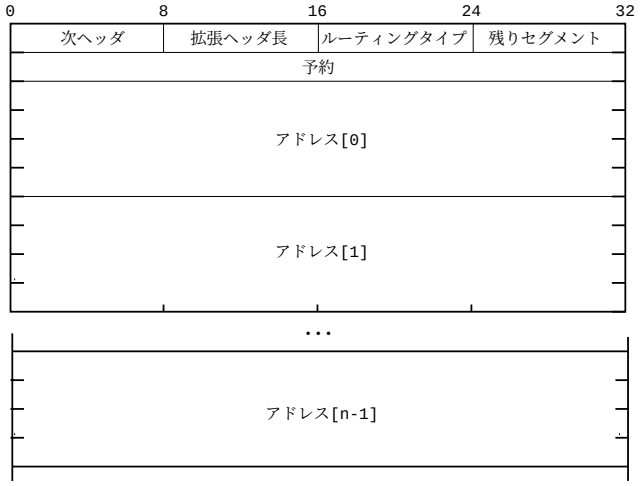


図 4.1: ルーティングヘッダのフォーマット

ルーティングヘッダの 5 オクテット目以降は、ルーティングタイプごとに定義された書式にしたがって用意される。上で述べたように、現在定義されているルーティングタイプは 0 だけなので、実際には図 4.1 はタイプ 0 の書式を表している。

- 予約 (Reserved)

  最初の 4 オクテットは予約フィールドであり、すべて 0 を設定しておく。ルーティングヘッダの処理では、単純に無視する。

- アドレス

  経由するルータのアドレスを順に記述するためのフィールドである。

4.2.2 従来のルーティングヘッダとの変更点

RFC1883で公開されていたルーティングヘッダ (図 4.2) から、厳密/非厳密ビットマスクが削除された。これは厳密 (strict) 経路制御の必要性が疑問視されたこと、及びこのマスクにより経由できるノードに制限 (23 ノード) が生じるという 2 つの理由からである。これによりルーティングは、すべて非厳密ルーティング (指定したホストを順に通過するのみ) として扱われる。

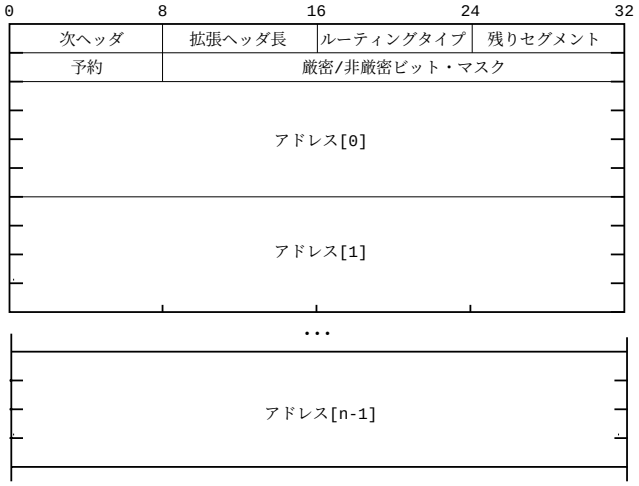


図 4.2: ルーティングヘッダのフォーマット (変更前)

4.2.3 ルーティングヘッダの処理

複数の途中経路が指定された場合は、ヘッダを書き換えながら処理が進められる。表 4.1 は始点ホスト S から終点ホスト D に到達するまでに、途中経路として A、B、C の 3 つのホストを経由する場合の処理において、ヘッダがどのように書き換えられていくかを示したものである。

ホップ	IPv6 ヘッダ		ルーティングヘッダ				
	始点	終点	Hdr Ext Len	Segments Left	Address[1]	Address[2]	Address[3]
S → A	S	A	6	3	B	C	D
A → B	S	B	6	2	A	C	D
B → C	S	C	6	1	A	B	D
C → D	S	D	6	0	A	B	C

表 4.1: ソースルーティング処理でのヘッダの書き換え

最初の行は、始点ホスト S で用意される IP データグラムの各ヘッダ値である。始点ホスト S では、通常の IP データグラム転送処理を行う。A に到達するまでのノードでは、ルーティングヘッダに関する処理は不要である。

ノード A に到達すると以下の処理を行う。

1. 拡張ヘッダの残りセグメント (Segments Left) の値をチェックする。この値は 0 ではないので、ソースルーティング処理を行う。
2. 拡張ヘッダの値から、ルーティングヘッダに格納されているアドレスの個数を得る。これを n とする。拡張ヘッダ長は 8 オクテット単位としており、IPv6 アドレスは 16 オクテットから構成されるため、アドレスの個数は

   $$n = HdrExtLen/2$$

   で得られる。表 1 の例では、n は 3 である。残りセグメントの値が n よりも大きければ IP データグラムを破棄し、ICMP Parameter Problem(Code 0) メッセージを生成する。
3. 残りセグメントの値を 1 つ減らす。
4. 次に、

   $$i = n - Segments\_Left$$

   を計算し、次に到着すべきアドレスを得る。A では、$i = 1$ となる。
5. ルーティングヘッダの Address[i] と、IPv6 ヘッダの終点アドレスを入れ換える。A では、Address[1] と終点アドレスを入れ換える。
6. 次に IPv6 ヘッダのホップリミットフィールドの値が 1 であれば、ホップリミットの定義にしたがって IP データグラムを破棄し、ICMP Time Exceeded(Hop Limit Exceeded) メッセージを生成して始点ホスト S に返送する。ホップリミットフィールドの値が 1 でなければ、ホップリミットを 1 減らす。
7. 最後に新しい送信先に転送するためのモジュールにパケットを渡す。

この処理をルーティングヘッダで指定された途中で順次実行していけば、ソースルーティングが実現できる。

4.3 IPv4におけるソースルーティング

IPv6 のルーティングヘッダとほぼ同等な機能を提供する IPv4 にも存在した。これはソースルーティングオプションと呼ばれるオプションである。しかしながらこのオプションは、IPv4 の必須の機能ではないことも影響し、実際にはあまり使用されてはいない。

IPv4 におけるソースルーティングオプションにも、初期の IPv6 ルーティングヘッダと同じように厳密な経路制御 (図 4.3) と非厳密な経路制御 (図 4.4) が存在する。しかし初期の IPv6 ルーティングヘッダと違い、IPv4 のソースルーティングオプションは厳密、非厳密が別のオプションとして定義される。以下にこのオプションのフォーマットを示す。

図 4.3: ソースルーティングオプションにおける厳密な経路制御のフォーマット

図 4.4: ソースルーティングオプションにおける非厳密な経路制御のフォーマット

- タイプ (Option type)

  オプションのタイプを示している。

- オプション長 (Option Length)

  ソースルーティングオプションの長さを示している。

- ポインタ (Pointer)

  次に処理されるべきアドレスに対するポインタである。IPv6 のソースルーティングヘッダにおける残りセグメント (Segments Left) と同様の働きをしている。

第 5 章

能動的な経路制御が可能なアプリケーションの実現

5.1 FTPへの経路制御の実装

ルーティングヘッダを用いた経路制御可能なアプリケーションとして、本研究ではFTP (File Transfar Protocol) 上に、この経路制御機構を実現した。FTP では、通信を行う場合にコントロールコネクションとデータコネクションの 2 つのコネクションを用いる。またこの 2 つのコネクションに対して、それぞれ送信および受信のアクションが存在するため、ルーティングヘッダによる多様な経路が設定可能となる。

5.2 データ送信時における経路設定

コントロールコネクションとデータコネクションにおけるデータ送信の経路を指定するために、FTP クライアントの内部コマンドとして、“putroute” というコマンドを定義した。このコマンドは、2 つのコネクションにおけるデータの送信経路を設定する場合に用いる内部コマンドである。またこのコマンドで、設定した経路の参照や、一時的なルーティングヘッダによる経路制御の有効、無効などの設定を行うことが可能なものとした。

表 5.1 に “putroute” の使用法の一覧を示す。

コマンド			動作および状態			
内部コマンド	引数 1	引数 2	動作	経路制御	コントロール	データ
putroute	-	-	表示	-	○	○
putroute	経路	-	設定	ON	○	○
putroute	ctl	-	表示	-	○	-
putroute	data	-	表示	-	-	○
putroute	off	-	設定	OFF	○	○
putroute	on	-	設定	ON	○	○
putroute	ctl	経路	設定	ON	○	-
putroute	ctl	off	設定	OFF	○	-
putroute	ctl	on	設定	ON	○	-
putroute	data	経路	設定	ON	-	○
putroute	data	off	設定	OFF	-	○
putroute	data	on	設定	ON	-	○

表 5.1: ルーティングヘッダを実装した FTP の内部コマンド putroute

“putroute” における引数の “ctl” および “data” はそれぞれコントロールコネクション、データコネクションに対するアクションを示している。またこれらの指定が無い場合は、2つのコネクションの両方を指定したものとする。“経路” は経由するノードを指定する。この “経路” には経由させたいノードの名前を “@” を区切りとして順に記述する。例えば v6router1、v6router2、v6router4を経由する場合は、“@v6router1@v6router2@v6router4” と記述する。“on” および “off” はそれぞれ、設定した経路を一時的に有効、無効にするものである。また経路の指定および “on”、“off” の指定を行わない場合は、設定している経路の指定を表示する。その他に、内部コマンド “help” に対する引数として “putroute” を指定すると、このコマンドに対するヘルプの出力を行う。

5.3 データ受信時における経路設定

データ受信時における経路指定も、受信時と同様である。しかしながら受信時における経路制御の指定は、FTP のクライアントとサーバと強調して行う必要がある。以下にクライアントとサーバへの経路制御機構の実装について述べる。

5.3.1 クライアント上への実装

データ受信時における経路設定も送信時と同様に “getroute” という内部コマンドを定義することにより実現する。コマンドの引数やその意味も受信時の場合と同様である。表 5.3.1 にその使用法の一覧を示す。

コマンド			動作および状態			
内部コマンド	引数 1	引数 2	動作	経路制御	コントロール	データ
getroute	-	-	表示	-	○	○
getroute	経路	-	設定	ON	○	○
getroute	ctl	-	表示	-	○	-
getroute	data	-	表示	-	-	○
getroute	off	-	設定	OFF	○	○
getroute	on	-	設定	ON	○	○
getroute	ctl	経路	設定	ON	○	-
getroute	ctl	off	設定	OFF	○	-
getroute	ctl	on	設定	ON	○	-
getroute	data	経路	設定	ON	-	○
getroute	data	off	設定	OFF	-	○
getroute	data	on	設定	ON	-	○

表 5.2: ルーティングヘッダを実装した FTP の内部コマンド getroute

このように、受信時におけるコマンドも送信時と同様であるが、受信時における経路情報は、FTP サーバへ送信する必要がある。

クライアントがサーバへ情報を送信する際に、FTP においてはアクセス制御コマンドまたは転送パラメータコードを用いる。アクセス制御コマンドは、その名の通り、アクセスの制御に関するコマンドで、ユーザの情報やパスワードの情報をやり取りするためのコマンドである。また転送パラメータコードは、それぞれが通信などに関するデフォルト値を持っており、そのデフォルト値を変更したい場合に用いるコマンドである。

本研究においては、この経路制御の機構を組み込むために、“CROUTE” および “DROUTE” の 2 つの転送パラメータコードを定義した。“CROUTE” は受信時におけるコントロールコネクションの経路に関するパラメータであり、“DROUTE” は受信の経路に関するパラメータである。表 5.3.1 にこのコマンドの使用法の一覧を示す。

コード　引数	動作内容
CROUTE　SHOW	コントロールコネクションの経路情報表示
CROUTE　ON	コントロールコネクションの経路制御 ON
CROUTE　OFF	コントロールコネクションの経路制御 OFF
CROUTE　経路	コントロールコネクションの経路設定
DROUTE　SHOW	データコネクションの経路情報表示
DROUTE　ON	データコネクションの経路制御 ON
DROUTE　OFF	データコネクションの経路制御 OFF
DROUTE　経路	データコネクションの経路設定

表 5.3: 経路設定用転送パラメータコード (CROUTE,DROUTE)

引数 “SHOW” は設定されている経路情報の、“ON” および “OFF” は一時的な経路の指定の、有効、無効を指定するものである。“経路” は、内部コマンド おいて指定する形式と同様に “@” を用いて記述する。

また既存のパラメータコードとして定義されている “HELP” の引数として、この 2 つのコードを渡すことにより、ヘルプ情報の要求が可能となる。

5.3.2　サーバ上への実装

FTP サーバ上では、クライアントから送信されたパラメータをもとに、経路の設定およびその応答、設定された経路情報などを、クライアントに送信する必要がある。このときに用いるコードは応答コードと呼ばれる。

応答コードは、3 桁のコード および空白、それに続く 1 行の文章によって構成される。この最初の 3 桁のコードの各桁には、それぞれ意味を持っている。

経路設定に関するパラメータに対する応答としては、そのパラメータの成功および失敗の 2 パターンを定義しており、成功においてはコードとして “200” および、設定した経路情報をクライアントに送信する。コード “200” は肯定的な完了応答を示すもので、コマンドが成功した場合に送信する既存のコードである。失敗においてはコードとして “501” および、設定に失敗した経路情報をクライアントに送信する。コード “501” は永久否定的応答を示すもので、指定したコマンドの引数やパラメータにエラーがある場合に送信する既存のコードである。

第 6 章

経路制御アプリケーションの検証実験とその考察

ここでは、経路制御アプリケーションの動作確認および、ルーティングヘッダが通信に与える影響についての検証実験について述べる。

6.1 アプリケーションの経路指定の確認

作成したアプリケーションが正確に経路の指定を行っていることを、以下の方法により確認した。

まず帯域の異なる経路 (10Mbps、100Mbps) が存在するネットワークを構築した。次にこの 2 つの通信経路をアプリケーションから指定し、それぞれの通信においてプロトコルアナライザによるパケットのモニタリングや通信速度の違いを計測した。図 6.1 に実験を行ったネットワーク構成を示す。

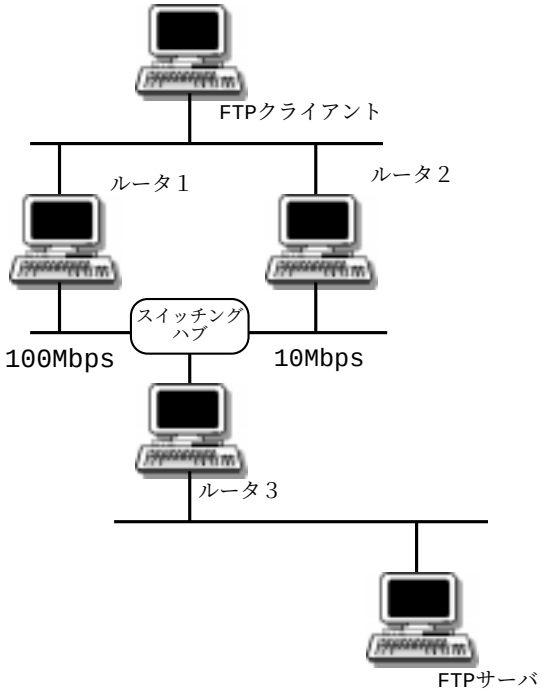


図 6.1: 検証実験におけるネットワーク

10Mbps および 100Mbps の経路を、図のルータ 1 あるいはルータ 2 を経由させることにより選択する。実験結果として、データ転送におけるスループットとして、それぞれ 10 回平均で 802.5KB/s、2.389MB/sが得られた。よってこのスループットの違いにより経路選択が正しく行われていると言える。

次にプロトコルアナライザによって、ヘッダ上に指定した経路を示すルーティングヘッダが付加されているかの確認を行った。図 6.2 に 10Mbps の経路を指定した場合の FTP クライアントが受信したパケットの 1 つを示す。(経由するノードとしてルータ 3 およびルータ 2 を指定)

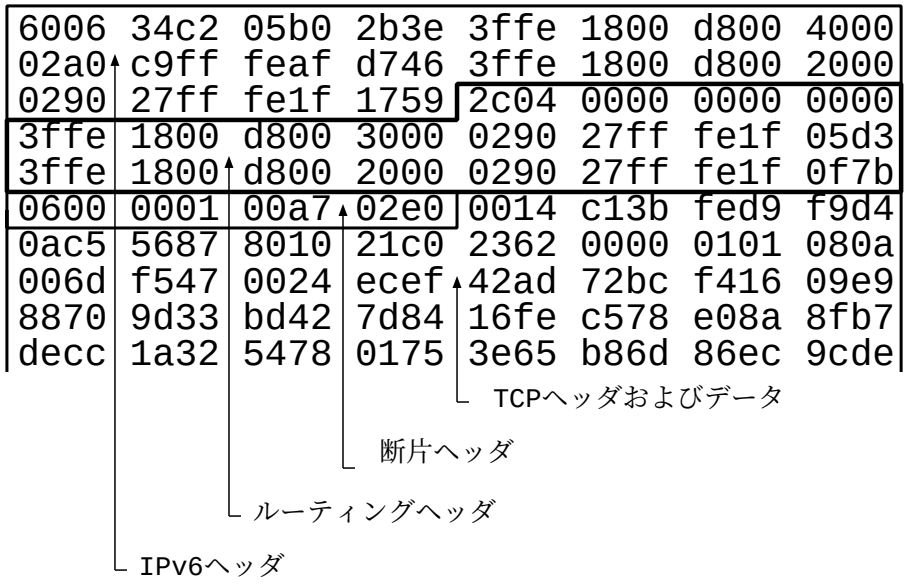


図 6.2: ルーティングヘッダが付加された IPv6 パケット

図のように、IPv6 ヘッダ、ルーティングヘッダ、断片ヘッダ、TCP ヘッダおよびデータが順に並んでいることが確認できる。

ここでルーティングヘッダの部分のみを見ると、以下のようになっている。

```
                                2c04 0000 0000 0000
3ffe 1800 d800 3000 0290 27ff fe1f 05d3
3ffe 1800 d800 2000 0290 27ff fe1f 0f7b
```

まず次ヘッダとして、断片ヘッダの識別である “2c” が指定されている。またその次に、拡張ヘッダ長として”04” が指定されている。これは、8 オクテット単位で記述されるため、経由ノードが 2 つ指定されていることが分かる。そしてその次の 48 ビットの “0” は、それぞれ経路制御タイプ、中継点残数、予約を示しており、正しく指定されている。 そして残りの 256 ビットの部分は、中継ノードであるルータ 3 およびルータ 2 のアドレスを示している。

これより正しくルーティングヘッダが付加されていると確認できた。

6.2 ルーティングヘッダが通信に与える影響の検証実験

経路を指定してデータを送信する場合、ルーティングヘッダが拡張ヘッダとして IPv6 ヘッダに付加され、各ルータやエンドノードで処理される。このため、ルーティングヘッダを用いない場合と比べ、若干のオーバーヘッドが生じる可能性が考えられる。

そこで、今回作成したアプリケーションを用いて、

- 通信帯域
- 経路指定数
- ルーティングヘッダの有無
- FTP におけるデータの送信、受信における経路指定のパターン
  (コントロールコネクションのみの経路指定 など)

などの違いによる送信経路、転送時間、パケット数等の計測を行った。

計測結果として、図 6.3 に 100Mbps の経路を指定してデータを受信した際の、経路指定数の違いによる転送時間を示す (試行回数 10 回)。

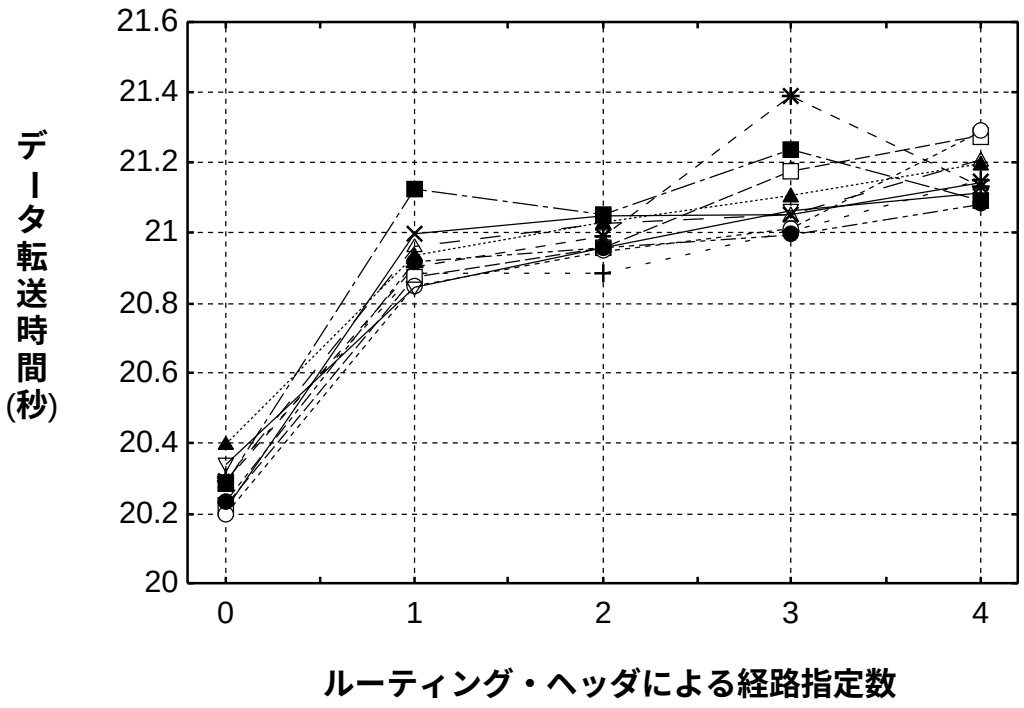


図 6.3: 経路指定数の違いにおける転送時間 (100Mbps でのデータ受信)

この計測結果からは、ルーティングヘッダの有無が、転送時間に平均 約 0.65 秒、経由するルータが 1 つ増加するごとに平均 約 0.08 秒程度の影響を与えることが分かる。

また図 6.4 に同じく 100Mbps の経路を指定したデータを送信した際の結果を示す。

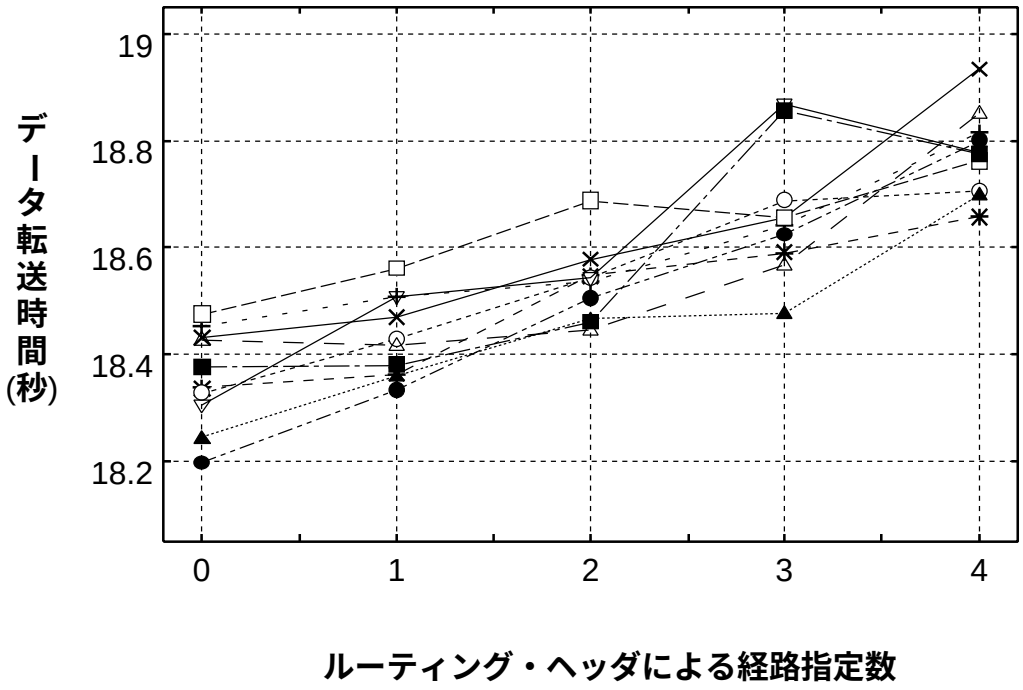


図 6.4: 経路指定数の違いにおける転送時間 (100Mbps でのデータ送信)

100Mbps の経路を指定した際のデータ送信においても、少し特性が異なるが受信の時と同様の結果が得られた。

これらの検証実験により、100Mbps の経路においては、十分許容範囲であるといえる。

また同様に、10Mbps の経路を指定した場合の検証実験も同様に行った。受信における結果を図 6.5 に、送信時における結果を図 6.6 に示す。

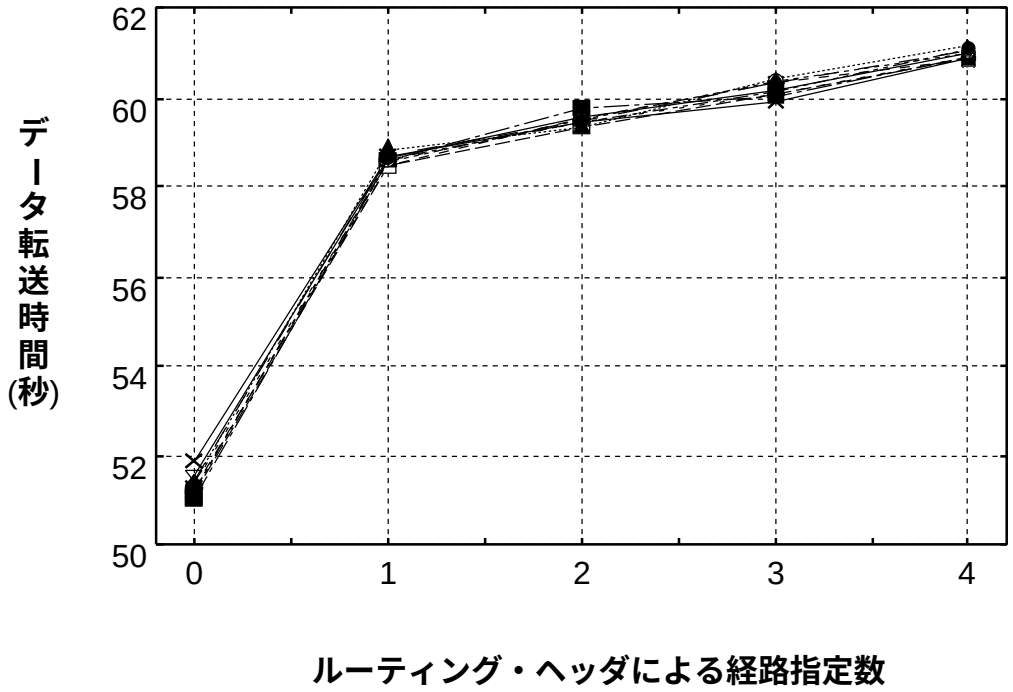


図 6.5: 経路指定数の違いにおける転送時間 (10Mbps でのデータ受信)

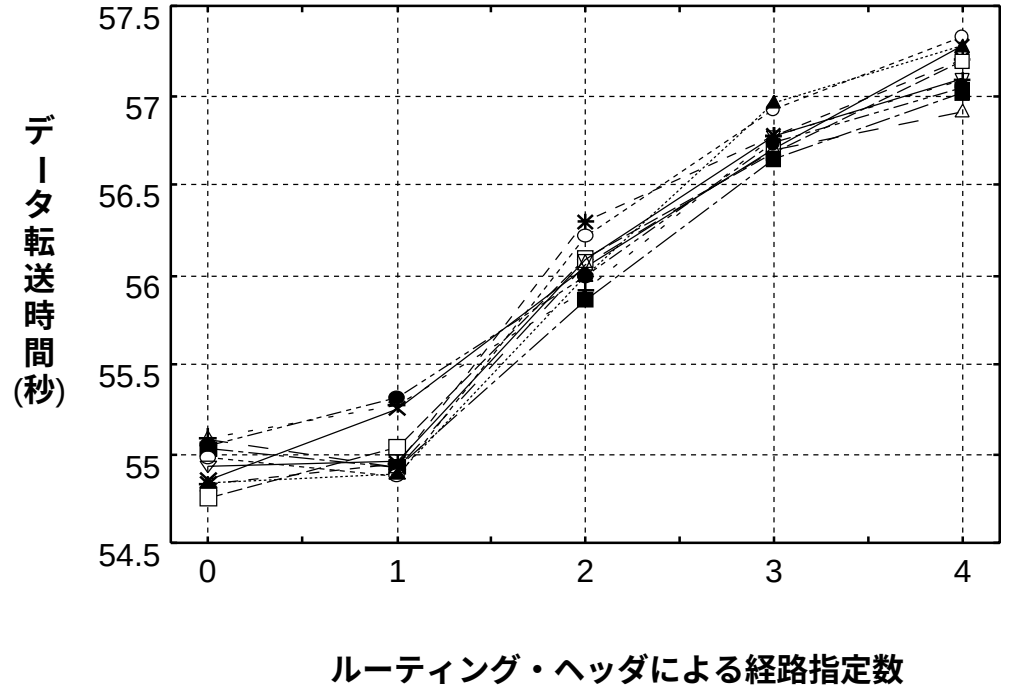


図 6.6: 経路指定数の違いにおける転送時間 (10Mbps でのデータ送信)

この計測結果からは、データの受信において、ルーティングヘッダの有無が、転送時間に平均 約9秒、経由するルータが1つ増加するごとに平均 約0.77秒のを与えることが分かる。データの送信においても特性が異なるが、同様の結果が得られている。

この結果は、10Mbpsのネットワークの転送においては影響が大きいように思われるが、これからのネットワークの通信速度の高速化などを考えると許容できると考えられる。

6.3 経路の決定方法に関する考察

このようにアプリケーションからの能動的な経路方法の手法としてルーティングヘッダを用いることは有効な手段であると言えるが、現状ではアプリケーションを使用するユーザが経路とその特性を十分把握しておく必要がある。そのため動的な経路決定メカニズムが必要となると考える。

よってこの問題を解決するため、ルーティングヘッダを考慮にいれて、以下の6つの手法を提案する。

- ユーザ主導型
- サーバ主導型
- ルーティングサーバ主導型
- 既存ルーティング依存型
- 経路探索型
- アドレス選択型

以下にそれぞれの経路決定方法の特徴について述べる。

- **ユーザ主導型**
  - アプリケーションを使うユーザや、クライアントがデータの経路を決定する方法である。
  - ユーザやクライアントが能動的に経路を選択することが可能。
  - ユーザやクライアントがあらかじめ送信経路および、アプリケーションの性質を把握しておく必要がある。
- **サーバ主導型**
  - サーバアプリケーションのデータ特性によってサーバが経路を決定する方法である。

- サーバが経路を指定するため、アプリケーションの特性に応じた的確な経路の選択が可能。
- 個々のサーバそれぞれが動的な経路変化に対応することが困難。

- **ルーティングサーバ主導型**
  - ユーザやサーバアプリケーションが、QoS(Quality of Service) に応じた経路を管理するサーバ (ルーティングサーバ) に問い合わせることにより経路を決定する方法である。
  - ユーザやサーバが経路を決定する場合に、必要とする QoS を指定しそれに応じた経路を取得するため、ユーザやサーバの経路決定コストが比較的少ない。
  - 動的な経路変化に対応するためには、経路探索メカニズムを組み込む必要がある。
  - 一般的な QoS のポリシーサーバと異なり、クライアントやサーバからも経路の問い合わせが起こるため、ルーティングサーバの配置なども考慮に入れる必要がある。

- **既存ルーティング依存型**
  - ルーティングヘッダを付加せずにルーティングプロトコル等に経路制御を任せデータを送信する方法である。
  - ルーティングヘッダを付加することのオーバーヘッドが大きなデータ (受信データの確認応答など) に有効。
  - ルーティングプロトコルによる経路決定が破綻しているときにデータの送信が困難。

- **経路探索型**
  - 送信されてきたデータのルーティングヘッダの情報から返信データのための経路を探索し、送信データと逆の経路でデータを送信する方法である。
  - 送信データと受信データの性質が異なる場合に最適な経路を選択できない。
  - 送信データの逆の経路がファイアウォールなどで防がれている場合データが配送できない。
  - 送信データの指定が曖昧な場合、探索経路が曖昧になってしまう可能性がある。

- **アドレス選択型**
  - ノードが複数のアドレスを取得し、そのアドレスごとに経路を定義し、そのアドレスを選択することにより経路を決定する方法である。
  - アドレスを選択するため、既存のアプリケーションに変更を行う必要がない。
  - アドレスごとに経路を設定しなければならず、経路情報の把握が困難。

上に述べた経路の選択方法をデータの送信、受信において選択することになるが、それぞれの組合せについては今後検討する予定である。

第 7 章

おわりに

近年のインターネットの急激な普及により現在のインターネットプロトコルである IPv4 の問題点が表面化してきた。この問題点を解決し、新たな機能を追加するために次世代インターネットプロトコル IPv6が提案された。我々は、この IPv6 の研究や IPv4 からの移行技術の構築を目的として、実際にネットワークを構築、運用を行った。

またユーザがアプリケーションに対して要求する通信品質に応じた能動的な経路選択を可能にするため、次世代インターネットプロトコル IPv6 におけるルーティングヘッダに着目し、実験ネットワーク環境下で FTP 上に実装し、実際にその効果を検証した。

今後は他のアプリケーションへのルーティングヘッダの実装や、ルーティングヘッダを用いて経路制御を行う場合の、経路の決定方法の考察を行って行く予定である。

謝辞

本研究を行うにあたって、御指導下さいました近藤弘樹 教授、渡辺健次 助教授を始めとする本研究室の皆様に心から御礼申し上げます。

参考文献

[1] S. Bradner and A. Mankin,IP: Next Generation (IPng) White Paper Solicitation,,Request for Comments 1550,(Dec. 1993)

[2] C. Partridge and F. Kastenholz,Technical Criteria for Choosing IP The Next Generation (IPng), Request for Comments 1726,(Dec. 1994)

[3] S. Bradner and A. Mankin,The Recommendation for the IP Next Generation Protocol,Request for Comments 1752,(Jan. 1995)

[4] S. Deering and R. Hinden,Internet Protocol, Version 6 (IPv6) Specification,Request for Comments 2460,(Dec. 1998)

[5] S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification,Request for Comments 1883,(Dec. 1995)

[6] C. Madson and R. Glenn,The Use of HMAC-SHA-1-96 within ESP and AH,Request for Comments 2404,(Nov. 1998)

[7] S. Kent and R. Atkinson,IP Authentication Header,Request for Comments 2402,(Nov. 1998)

[8] IPng Implementations,
http://playground.sun.com/ipng/ipng-implementations.html

[9] KAME Project,http://www.kame.net

[10] Christian Huitema, 村井純 監修,WIDEプロジェクト IPv6 分科会監訳、松島英樹訳,IPv6 次世代インターネットプロトコル, プレンティスホール出版,(Jan. 1997)

[11] Mark A.Miller, トップスタジオ訳, 宇夫陽次郎 監修, 次世代インターネットプロトコルの仕様と事例 IPv6 入門, 翔泳社,(Jun. 1999)

[12] W. Stevens and M. Thomas,Advanced Sockets API for IPv6,Request for Comments2292,(Feb. 1998)

[13] W. Stevens and M. Thomas,Advanced Sockets API for IPv6,draft-ietf-ipngwg-2292bis-01.txt,(Dec. 1999)

[14] Gilligan, et. al.,Basic Socket Interface Extensions for IPv6,Request for Comments2553,(Mar. 1999)

[15] 大谷誠:“次世代インターネット・プロトコル IPv6 の実装と相互通信性の検証”, 佐賀大学理工学部情報科学科卒業論文,(Mar. 1998)

[16] 大谷誠, 田中久治, 渡辺健次, 近藤弘樹: “IPv6 のネットワーク構築とその運用”, 情報処理学会九州支部研究会,(Mar. 1999)

[17] 大谷誠, 渡辺健次, 近藤弘樹: “IPv6 におけるルーティングヘッダを用いたアプリケーションの実現”, 電気関係学会九州支部第 52 回連合大会, (Oct. 1999)

[18] 大谷誠, 津田伸秀, 渡辺健次, 近藤弘樹: “IPv6 におけるルーティングヘッダを用いたアプリケーションの実現とその検証”, 情報処理学会第 60 回全国大会,(Mar. 2000)

付録 A

Advanced Sockets API for IPv6 Routing Header Option

Source routing in IPv6 is accomplished by specifying a Routing header as an extension header. There can be different types of Routing headers, but IPv6 currently defines only the Type 0 Routing header [RFC-2460]. This type supports up to 127 intermediate nodes (limited by the length field in the extension header). With this maximum number of intermediate nodes, a source, and a destination, there are 128 hops.

Source routing with IPv4 sockets API (the IP_OPTIONS socket option) requires the application to build the source route in the format that appears as the IPv4 header option, requiring intimate knowledge of the IPv4 options format. This IPv6 API, however, defines eight functions that the application calls to build and examine a Routing header, and the ability to use sticky options or ancillary data to communicate this information between the application and the kernel using the IPV6_RTHDR option.

Three functions build a Routing header:

```
  inet6_rth_space()    - return #bytes required for Routing header
  inet6_rth_init()     - initialize buffer data for Routing header
  inet6_rth_add()      - add one IPv6 address to the Routing header
```

Three functions deal with a returned Routing header:

```
  inet6_rth_reverse()  - reverse a Routing header
  inet6_rth_segments() - return #segments in a Routing header
  inet6_rth_getaddr()  - fetch one address from a Routing header
```

The function prototypes for these functions are defined as a result of including the <netinet/in.h>.

To receive a Routing header the application must enable the IPV6_RECVRTHDR socket option:

```
    int  on = 1;
```

```
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

To send a Routing header the application specifies it either as ancillary data in a call to sendmsg() or using setsockopt().

The application can remove any sticky Routing header by calling

setsockopt() for IPV6_RTHDR with a zero option length.

When using ancillary data a Routing header is passed between the application and the kernel as follows: The cmsg_level member has a value of IPPROTO_IPV6 and the cmsg_type member has a value of IPV6_RTHDR. The contents of the cmsg_data[] member is implementation dependent and should not be accessed directly by the application, but should be accessed using the six functions that we are about to describe.

The following constant is defined as a result of including the <netinet/in.h>:

```
    #define IPV6_RTHDR_TYPE_0    0 /* IPv6 Routing header type 0 */
```

When a Routing header is specified, the destination address specified for connect(), sendto(), or sendmsg() is the final destination address of the datagram. The Routing header then contains the addresses of all the intermediate nodes.

A.1 inet6_rth_space

```
    size_t inet6_rth_space(int type, int segments);
```

This function returns the number of bytes required to hold a Routing header of the specified type containing the specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 0 and 127, inclusive. The return value is just the space for the Routing header. When the application uses ancillary data it must pass the returned length to CMSG_LEN() to determine how much memory is needed for the ancillary data object (including the cmsghdr structure).

If the return value is 0, then either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.

(Note: This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, since all the ancillary data objects must be specified to sendmsg() as a single msg_control buffer.)

A.2 inet6_rth_init

```
void *inet6_rth_init(void *bp, int bp_len, int type, int segments);
```

This function initializes the buffer pointed to by bp to contain a Routing header of the specified type and sets ip6r0_len based on the segments parameter. bp_len is only used to verify that the buffer is large enough. The ip6r0_segleft field is set to zero; inet6_rth_add() will increment it.

When the application uses ancillary data the application must initialize any cmsghdr fields.

The caller must allocate the buffer and its size can be determined by calling inet6_rth_space().

Upon success the return value is the pointer to the buffer (bp), and this is then used as the first argument to the next two functions. Upon an error the return value is NULL.

A.3 inet6_rth_add

```
int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

This function adds the IPv6 address pointed to by addr to the end of the Routing header being constructed.

If successful, the segleft member of the Routing Header is updated to account for the new address in the Routing header and the return value of the function is 0. Upon an error the return value of the function is -1.

A.4 inet6_rth_reverse

```
int inet6_rth_reverse(const void *in, void *out)
```

This function takes a Routing header extension header (pointed to by the first argument) and writes a new Routing header that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

The return value of the function is 0 on success, or -1 upon an error.

A.5 inet6_rth_segments

```
int inet6_rth_segments(const void *bp);
```

This function returns the number of segments (addresses) contained in the Routing header described by bp. On success the return value is zero or greater. The return value of the function is -1 upon an error.

A.6 inet6_rth_getaddr

```
struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

This function returns a pointer to the IPv6 address specified by index (which must have a value between 0 and one less than the value returned by inet6_rth_segments()) in the Routing header described by bp. An application should first call inet6_rth_segments() to obtain the number of segments in the Routing header.

Upon an error the return value of the function is NULL.

A.7 Examples using the inet6_rth_XXX() functions

Here we show an example for both sending Routing headers and processing and reversing a received Routing header.

A.7.1 Sending a Routing Header

As an example of these Routing header functions defined in this document, we go through the function calls for the example on p. 17 of [RFC-2460]. The source is S, the destination is D, and the three intermediate nodes are I1, I2, and I3.

```
          S -----> I1 -----> I2 -----> I3 -----> D

   src:    *    S         S         S         S   S
   dst:    D   I1        I2        I3         D   D
   A[1]:  I1   I2        I1        I1        I1  I1
   A[2]:  I2   I3        I3        I2        I2  I2
   A[3]:  I3    D         D         D        I3  I3
   #seg:   3    3         2         1         0   3
```

src and dst are the source and destination IPv6 addresses in the IPv6 header. A[1], A[2], and A[3] are the three addresses in the Routing header. #seg is the Segments Left field in the Routing header.

The six values in the column beneath node S are the values in the Routing header specified by the sending application using sendmsg() of setsockopt(). The function calls by the sender would look like:

```
void  *extptr;
int   extlen;
struct msghdr  msg;
```

```
    struct cmsghdr  *cmsgptr;
    int   cmsglen;
    struct sockaddr_in6  I1, I2, I3, D;

    extlen = inet6_rth_space(IPV6_RTHDR_TYPE_0, 3);
    cmsglen = CMSG_SPACE(extlen);
    cmsgptr = malloc(cmsglen);
    cmsgptr->cmsg_len = CMSG_LEN(extlen);
    cmsgptr->cmsg_level = IPPROTO_IPV6;
    cmsgptr->cmsg_type = IPV6_RTHDR;

    optptr = CMSG_DATA(cmsgptr);
    optptr = inet6_rth_init(optptr, optlen, IPV6_RTHDR_TYPE_0, 3);

    inet6_rth_add(optptr, &I1.sin6_addr);
    inet6_rth_add(optptr, &I2.sin6_addr);
    inet6_rth_add(optptr, &I3.sin6_addr);

    msg.msg_control = cmsgptr;
    msg.msg_controllen = cmsglen;

    /* finish filling in msg{}, msg_name = D */
    /* call sendmsg() */
```

We also assume that the source address for the socket is not specified (i.e., the asterisk in the figure).

The four columns of six values that are then shown between the five nodes are the values of the fields in the packet while the packet is in transit between the two nodes. Notice that before the packet is sent by the source node S, the source address is chosen (replacing the asterisk), I1 becomes the destination address of the datagram, the two addresses A[2] and A[3] are "shifted up", and D is moved to A[3].

The columns of values that are shown beneath the destination node are the values returned by recvmsg(), assuming the application has enabled both the IPV6_RECVPKTINFO and IPV6_RECVRTHDR socket options. The source address is S (contained in the sockaddr_in6 structure pointed to by the msg_name member), the destination address is D (returned as an ancillary data object in an in6_pktinfo structure), and the ancillary data object specifying the Routing header will contain three addresses (I1, I2, and I3). The number of segments in the Routing header is known from the Hdr Ext Len field in the Routing

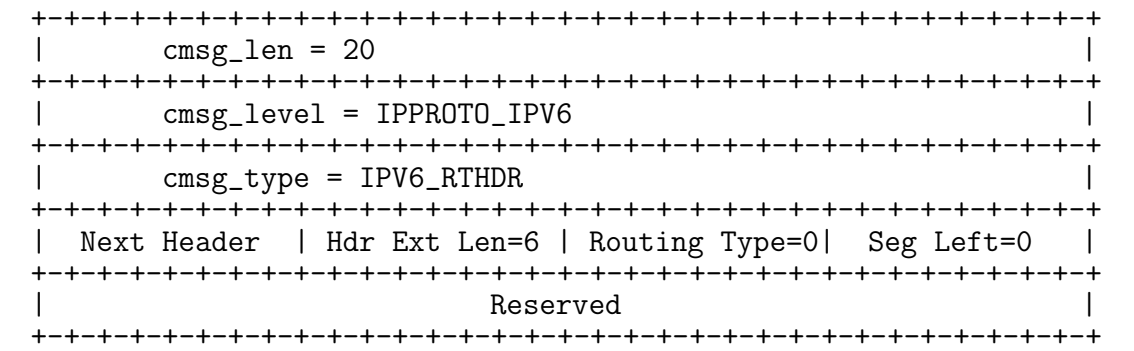
header (a value of 6, indicating 3 addresses).

The return value from inet6_rth_segments() will be 3 and inet6_rth_getaddr(0) will return I1, inet6_rth_getaddr(1) will return I2, and inet6_rth_getaddr(2) will return I3,

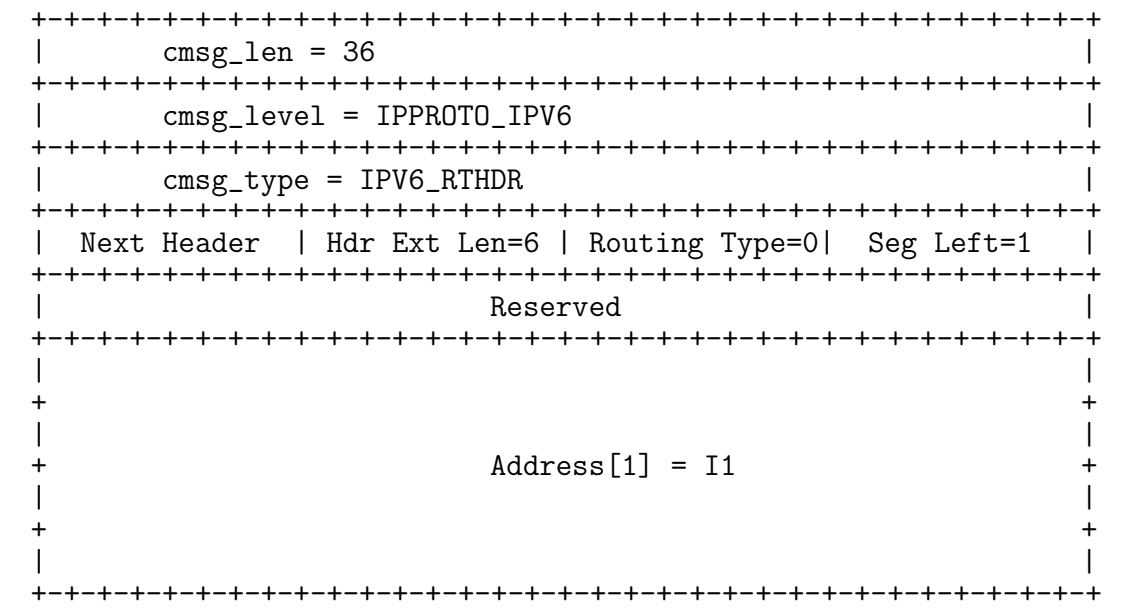
If the receiving application then calls inet6_rth_reverse(), the order of the three addresses will become I3, I2, and I1.

We can also show what an implementation might store in the ancillary data object as the Routing header is being built by the sending process. If we assume a 32-bit architecture where sizeof(struct cmsghdr) equals 12, with a desired alignment of 4-byte boundaries, then the call to inet6_rth_space(3) returns 68: 12 bytes for the cmsghdr structure and 56 bytes for the Routing header (8 + 3*16).

The call to inet6_rth_init() initializes the ancillary data object to contain a Type 0 Routing header:

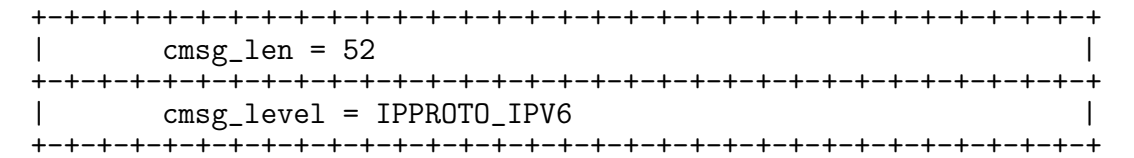


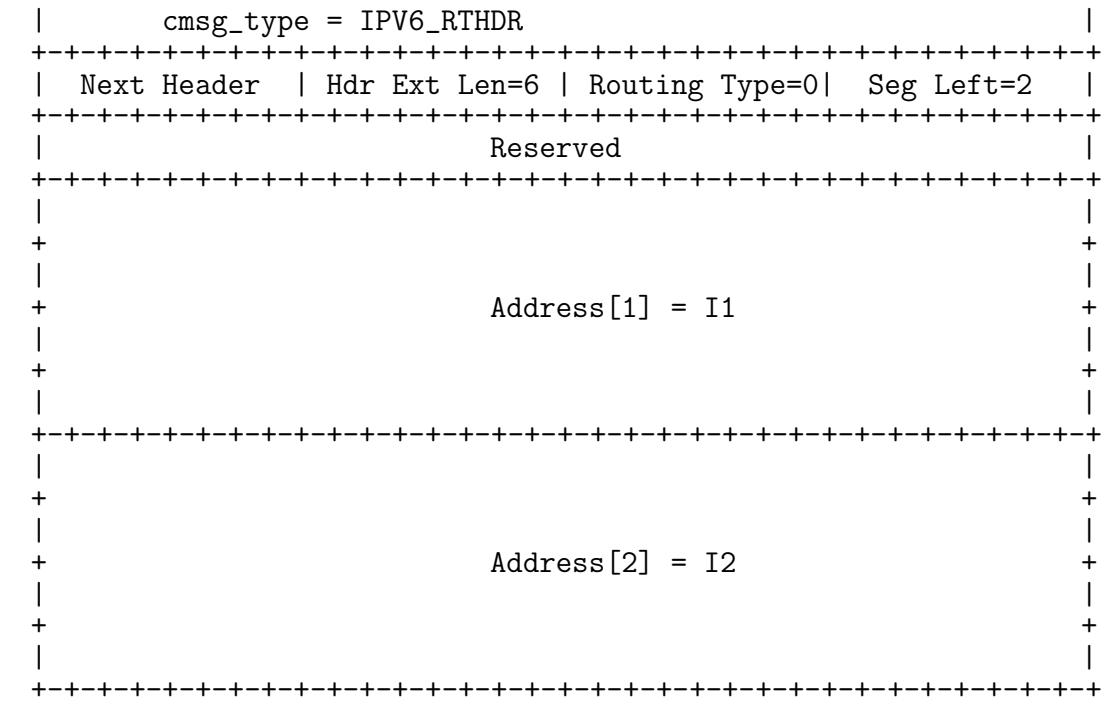
The first call to inet6_rth_add() adds I1 to the list.



cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

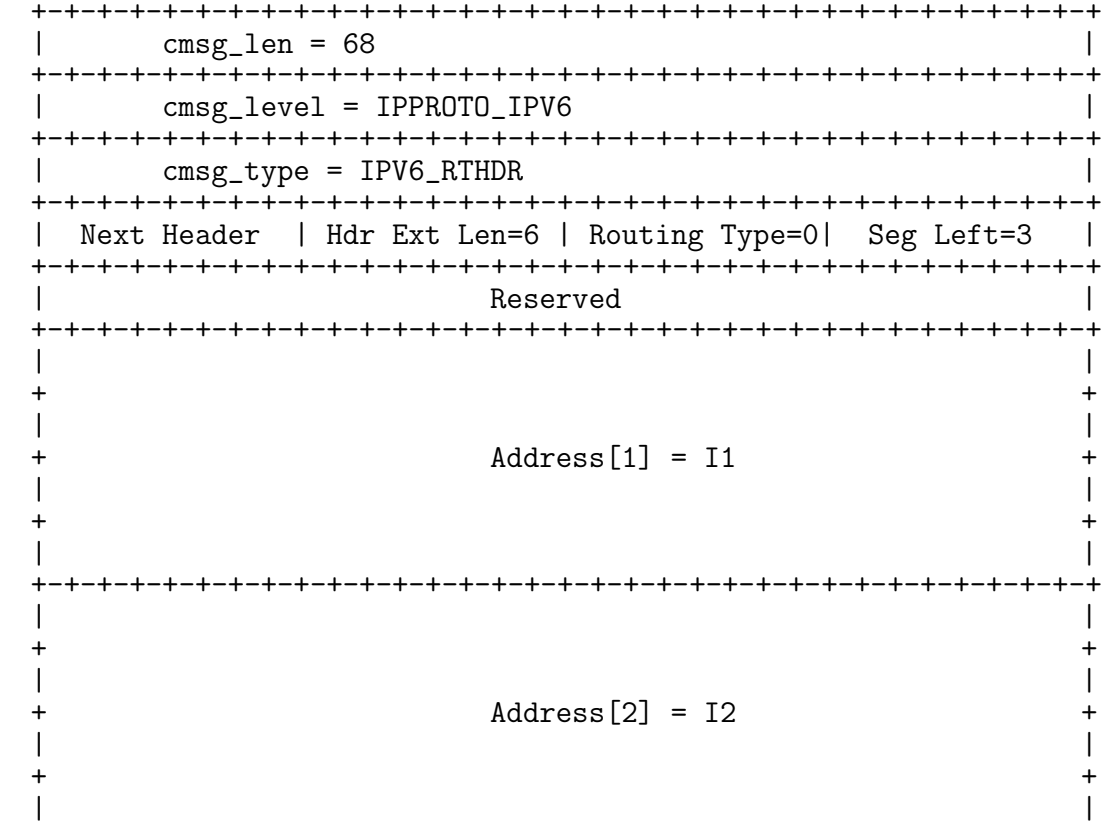
The next call to inet6_rth_add() adds I2 to the list.

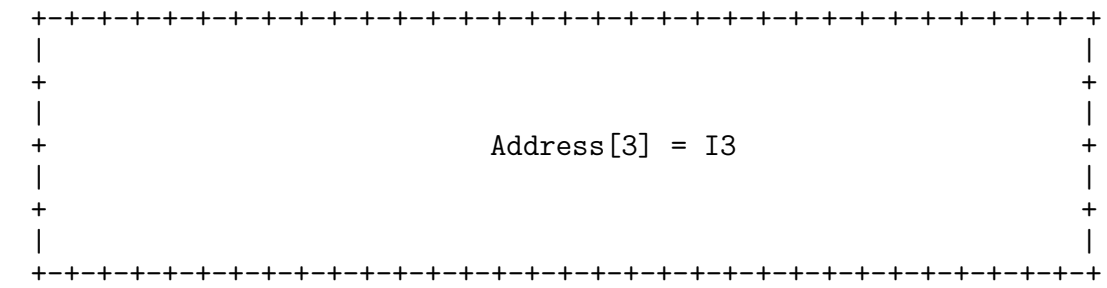




cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

The last call to inet6_rth_add() adds I3 to the list.





cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

A.7.2 Receiving Routing Headers

This example assumes that the application has enabled IPV6_RECVRTHDR socket option. The application prints and reverses a source route and uses that to echo the received data.

```
    struct sockaddr_in6     addr;
    struct msghdr           msg;
    struct iovec            iov;
    struct cmsghdr          *cmsgptr;
    size_t                  cmsgspace;
    void                    *optptr;
    int                     optlen;

    int                     segments;
    int                     i;
    char                    databuf[8192];

    segments = 100;        /* Enough */
    optlen = inet6_rth_space(IPV6_RTHDR_TYPE_0, segments);
    cmsgspace = CMSG_SPACE(optlen);
    cmsgptr = malloc(cmsgspace);
    if (cmsgptr == NULL) {
            perror("malloc");
            exit(1);
    }
    optptr = CMSG_DATA(cmsgptr);

    msg.msg_control = (char *)cmsgptr;
    msg.msg_controllen = cmsgspace;
    msg.msg_name = (struct sockaddr *)&addr;
    msg.msg_namelen = sizeof (addr);
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    iov.iov_base = databuf;
    iov.iov_len = sizeof (databuf);
    msg.msg_flags = 0;
    if (recvmsg(s, &msg, 0) == -1) {
            perror("recvmsg");
            return;
```

```
    }
    if (msg.msg_controllen != 0 &&
        cmsgptr->cmsg_level == IPPROTO_IPV6 &&
        cmsgptr->cmsg_type == IPV6_RTHDR) {
            struct in6_addr *in6;
            char asciiname[INET6_ADDRSTRLEN];
            struct ip6_rthdr0 *rthdr;

            rthdr = (struct ip6_rthdr0 *)optptr;
            segments = inet6_rth_segments(optptr);
            printf("route (%d segments, %d left): ",
                segments, rthdr->ip6r0_segleft);
            for (i = 0; i < segments; i++) {
                    in6 = inet6_rth_getaddr(optptr, i);
                    if (in6 == NULL)
                            printf("<NULL> ");
                    else
                            printf("%s ", inet_ntop(AF_INET6,
                                (void *)in6->s6_addr,
                                asciiname, INET6_ADDRSTRLEN));
            }
            if (inet6_rth_reverse(optptr, optptr) == -1) {
                    printf("reverse failed");
                    return;
            }
    }
    iov.iov_base = databuf;
    iov.iov_len = strlen(databuf);
    if (sendmsg(s, &msg, 0) == -1)
         perror("sendmsg");
    if (cmsgptr != NULL)
            free(cmsgptr);
```

Note: The above example is a simple illustration. It skips some error checks involving the MSG_TRUNC and MSG_CTRUNC flags.