

IPv6 とインターネットの将来

Technical White Paper



901 San Antonio Road
Palo Alto, CA 94303
1 (800) 786.7638
Sun Microsystems, Inc.
1.512.434.1511

Copyright 1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303 U.S.A

All Rights Reserved.

本書および本書に記載された製品（関連する文書を含み、以下同様とします）は著作権により保護されており、その使用、複製、再頒布および逆コンパイルを制限するライセンスのもとにおいて頒布されます。サン・マイクロシステムズ株式会社または同社に対する実施許諾者の書面による事前の許可なく、本書および本書に記載された製品のいかなる部分も、いかなる方法によっても複製することが禁じられます。

フォントを含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。本書に記載された製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。

UNIX は、X/Open Company Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph(c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

Sun、Sun Microsystems、The Network Is The Computer、Solaris、Sun Enterprise、NFS、ONC、Java、および Write Once, Run Anywhere は、米国およびその他の国における米国 Sun Microsystems, Inc. の商標または登録商標です。

Sun のロゴマーク、および Solaris は、米国 Sun Microsystems, Inc. の登録商標です。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems, Inc. が開発したアーキテクチャに基づくものです。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行なわれないものとします。

本書および本書に記載された製品が、外国為替および外国貿易管理法（外為法）に定められる戦略物資等（貨物または役務）に該当する場合、それを輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による同意を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

本書「IPv6 とインターネットの将来」は、1999 年 11 月に Sun Microsystems, Inc. が作成したドキュメント（英文タイトル：IPv6 and the Future of the Internet）を日本語化したものです。



Please
Recycle



目次

第 1 章	はじめに	1
	インターネットの成長とインターネットプロトコル	1
	IP のバージョンに依存しないテクノロジー	3
	IPv6 : 次世代のプロトコル	4
	Solaris™ オペレーティング環境 : Web の基盤	4
第 2 章	IPv4 と IPv6 の比較 : 概要	5
	IPv4 のバックグラウンド	5
	一時的な対策としての NAT	6
	長期的なソリューションとしての IPv6	7
	IPv6 を使用した将来の計画	7
第 3 章	IPv6 アプリケーション・プログラム・インタフェース	9
	IETF 仕様	9
	基本ソケット API	10
	高度なソケット機能	11
	Sun リモートプロシージャ呼び出し	12

第 4 章	クライアントコードの IPv6 へのポータビリティ	13
	ネームサービスと IPv4 から IPv6 への移行	13
	IPv4 と IPv6 の相互運用性	14
	IPv4 と IPv6 の命名 (Naming) API のちがい	15
第 5 章	サーバコードの IPv6 へのポータビリティ	23
第 6 章	アプリケーションのアップグレードを支援するツール	31
	Socket Code Scrubber for C/C++	31
	Socket Code Scrubber の使用例	32
第 7 章	まとめ	33
第 8 章	付録	35
	IPv4 から IPv6 へのソケットの変更	35
	IPv4 と IPv6 のちがい	37
第 9 章	参考文献	39

はじめに

TCP/IP プロトコルスイートはインターネットの基盤を提供します。今日、企業や個人の間で行われているすべてのデータ通信は、Web ブラウザ、電子メール、ファイル転送、リモートログインなどを含めて、事実上、これらのインターネットプロトコルに基づいています。今後数年間にわたって、インターネットは、現在のさまざまな制限を超えてインターネットの成長を可能にする重要な転換期に直面するはずですが、この転換期には、インターネットプロトコル (IP) のより新しく、より信頼性の高いバージョン、つまり IPv6 への移行も含まれます。さまざまな変更が TCP/IP プロトコルスイートの IP 部分に加えられます。

インターネットの成長とインターネットプロトコル

アカウントの数を見るだけでも、インターネットは信じられない速度で成長を続けています。DNS (Domain Name Servers - すべての常設ホストシステムを定義するインターネットサービス) に関する Internet Software Consortium の年次調査によれば、DNS の中で 5,600 万を超えるホストが識別されることが明らかになっています。前年の 3,600 万、1993 年 7 月の 200 万未満という数字と比較すれば、この調査結果はインターネットの急速な成長を裏付けています。最新の数字は、インターネットが年率で 65% を超える成長を遂げていることを示しており、しかもこの数字には DNS によって公式に発表されているシステムしか含まれていません。つまり、プライベートノードはこの調査の対象には含まれていないということです。

この成長と、この成長から予測されるより多くのアドレスに対する将来的な需要が、インターネットプロトコルの新バージョンに対するニーズを高める主な要因になっています。インターネットプロトコルでは、一意なインターネットアドレスの総数は 40 億の範囲に制限されています。ただし、これはあくまでも理論的な値であり、実際にはいくつかの要因によって億単位のアドレス数に制限されます。現在、約 1 億台のコンピュータがインターネッ

トに接続されていると推定されています。いつインターネットのアドレス数が不足するようになるのかは明らかではありませんが、2002年から2012年までのどこかでそれが起こるといのが大方の予想です。

IPアドレスが不足するという問題は時を迫ることに悪化の一途をたどっています。今日、インターネットアドレスを要求する最も一般的なデバイスは、价格的に数百ドルから数千ドルの範囲にあるPCです。将来、インターネットに接続するデバイスの数と種類はさらに増加すると予測されます。PDA、ポケベル、電話、自動車、あるいはその他多くの新製品がそれらに内蔵されたコンピュータを持つと考えられます。これらのデバイスの多くはPCよりもかなり安価であり、これらのデバイスがネットワークを構築しようとするれば、それらの多くがインターネット接続を要求することは確実です。これにより、さらに多くのIPアドレスに対する需要が劇的に増大することはまちがいありません。

また、IPv4ネットワークアドレスを取得することもだんだんと困難になりつつあります。使用可能なインターネットアドレスが少なくなるにつれ、いくつかの企業ではその上位にあるISPに追加のアドレスに対する要求を申し立てざるをえなくなります。しかし場合によっては、申し立ての理由に詳細なビジネスプランを加えなければならないこともあり、企業によってはこのような情報の開示に二の足を踏んでしまう可能性もあります。申し立ての正当性や、要求して使用可能にするノードの数によっては、要求そのものが拒否されることもあります。米国以外の国や地域では、新しいアドレスの取得がさらに困難になる可能性があります。実際、一部の国や地域では、米国よりもはるかに急速にIPアドレスが不足しつつあります。

インターネットは現在、インターネットプロトコルのバージョン4 (IPv4) に基づいています。この標準はJon Postelによって策定され、1981年9月1日に公刊されました。その当時、インターネットに接続されているコンピュータは約1,000台にすぎず、事実上、それらすべては大学や米国政府機関に設置されたタイムシェアリングベースの大型システムでした。個人ユーザ向けに設計されたコンピュータは、ほとんど存在していませんでした。

1981年当時、IPv4標準で使用可能な40億というアドレス数は非常に巨大なアドレス空間と考えられており、IPアドレスの割り当てに慎重になる必要など皆無でした。その結果、初期のアドレス割り当ては、その後の割り当てとは比較にならないほど大盤振る舞いで配布されました。そのため、早い時期にインターネットを採用した団体は必要以上に多くのアドレスを取得しています。このことは、インターネットアドレス空間の一部が浪費されていて、再割り当てできなかったということを意味しています。さらに、潜在的なアドレス空間の一部は、アドレスのブロックを使ってネットワークをより保守しやすく、またルーティングを容易にするために使用されており、それがさらに使用可能なアドレスを消費する原因となっています。

このようなIPv4の制限を克服するため、業界はインターネットプロトコルのバージョン4 (IPv4) からバージョン6 (IPv6) への移行の必要性を認識しました。IPv6では、IPv4の32

ビットに対して 128 ビットの一意なインターネットアドレスに基づくアドレス方式を使用します。

128 ビットがどれほど多くのアドレスを表現できるかを理解するため、それぞれのページに 100 個のノードの記述を含み、使用可能なすべてのインターネットアドレスを印刷した 1 冊の本を想像してみましょう。それぞれのページの用紙の厚さが 0.1 ミリで、用紙の両面にアドレスを印刷したとすると、使用可能なすべての IPv4 アドレスを記述した本の厚さは 2,000 メートルになります。これだけでも相当な厚さですが、使用可能なすべての IPv6 アドレスを記述した本の厚さは実に 2×10^{16} 光年に達します。現在、あのハッブル望遠鏡でさえ 2×10^{13} 光年を超える範囲を観測することはできません。IPv6 が近い将来の需要を十分に満たすことができるのは明らかです。

IP のバージョンに依存しないテクノロジー

IETF (Internet Engineering Task Force) が IPv6 の設計に着手したとき、一定の期間にわたって IPv4 ノードと IPv6 ノードの両方がネットワーク上で共存することが想定されていました。IPv4 から IPv6 への移行を支援するため、IPv6 コードインタフェースは IPv4 および IPv6 ノードの両方とデータをやり取りできるように設計されました。このことは、単一のプログラミングインタフェースを介して両方のプロトコルをサポートすることによって実現されています。特に、IPv6 の新しいソケットインタフェース API は、IPv4 および IPv6 両方のコミュニケーションを透過的にサポートします。

この新しいインタフェースを使用することで、アップグレードされたアプリケーションは、IPv4 を使って既存の IPv4 クライアントおよびサーバと自動的にデータをやり取りできるようになり、同時に IPv4 プロトコルに対応したアドレスだけで構成された IPv6 システムともデータをやり取りできるようになります。

アプリケーションを新しい IPv6 インタフェースにアップグレードすると、それらのアプリケーションは IPv6 プロトコルを使って他の IPv6 ノードとの接続を開始できるようになります。この移行は、企業が IPv6 ネットワークを展開すると同時に開始されます。この移行で最も重要なのは、アプリケーションがどのプロトコルを使用するかを指定する必要がないということです。新しいインタフェースが使用され、システムが IPv6 に対応して構成されていれば (IPv6 アドレスと構成を設定して)、IPv6 ソケットインタフェースは自動的に適切なプロトコルを選択して、所定のノードとのデータのやり取りにそのプロトコルを使用するようになります。

IPv6 : 次世代のプロトコル

17 年以上にわたり、Sun Microsystems は最新のネットワークシステムを開発するためにさまざまな努力を重ねてきました。ネットワーク機能を組み込んだ低価格ワークステーション

を最初に開発して製造したのも Sun でした。そして今日に至るまで、Sun の製品はネットワーク環境での使用を前提に設計されています。Sun はこれまで一貫して、Ethernet、TCP/IP などの主要な業界標準に基づいた信頼性が高く、柔軟性に富み、かつ強力なネットワークテクノロジーを提供してきました。そして、Sun の業界標準に対する前向きな姿勢は IPv6 のサポートにも引き継がれています。

Sun では、策定の初期段階から IPv6 にかかわり、Internet Architecture Board など、IETF の主要な作業グループのほとんどすべてに代表者を派遣しています。Sun の技術者たちは、IETF の地区責任者や、IP Next Generation (IPNG) Working Group、Next Generation Transition (ngtrans) Working Group などの IETF の作業グループの共同座長を務めており、彼らはさまざまな IETF 標準仕様書の共著者でもあります。

Sun はインターネットコミュニティに IPv6 のプロトタイプを提供した最初のベンダです。また、Sun の技術者たちは「6bone」、つまり史上初の IPv6 バックボーンネットワークの構築に従事し、その結果、Sun は「6bone」にはじめて接続されたサイトの 1 つになりました。

Sun では、IPv6 環境を開発する上で自らの専門技術を最大限に利用し、お客様を支援して、IPv4 から IPv6 への容易かつシームレスなアップグレードを実現することをお約束します。

Solaris™ オペレーティング環境：Web の基盤

情報へのシームレスなアクセスというビジョンと、17 年以上にわたる一貫した姿勢に支えられ、Sun はオープン・ネットワーク・コンピューティングの世界のリーダーと認められています。また、Sun では、電話回線と同等の信頼性でネットワーク接続を提供するために必要なコアテクノロジー、つまり WebTone への投資を続けています。

当初からソフトウェアに組み込まれていた TCP/IP および NFS™ 標準に基づくネットワーク機能により、Solaris™ オペレーティング環境は定評ある相互運用性 (インタオペラビリティ) を提供し、ネットワークングをより容易にするさまざまな最先端の機能を含んでいます。また、Sun では、一貫して Solaris オペレーティング環境のパフォーマンスの向上にも努めており、新しいアプリケーションが使用できるようになると同時に、それらのアプリケーションをサポートするために必要なプロトコルを Solaris オペレーティング環境に組み込んでいます。

数百にもおよぶすぐれた開発ツール、CORBA や Java™ テクノロジーなどの強力なインフラストラクチャテクノロジーに基づいた開発製品、さらに先進的なクライアント/サーバおよびネットワーク管理ツールが用意されている Solaris オペレーティング環境は、IPv6 アプリケーションを含めたネットワーク・アプリケーション・ソフトウェアを開発する上で理想的な環境といえます。

IPv4 と IPv6 の比較：概要

インターネットの成長の大部分は、インターネットプロトコルの IPv4 仕様を使って達成されました。このプロトコルのバージョンは、インターネットの途方もない成長という素晴らしい成果を成し遂げた一方で、その物理的な限界に急速に近づきつつあります。これから数年間、インターネットが予想通りに成長し、しかも数百万のネットワークノードおよびそれらのユーザにアップグレードが必要であるとすると、どうしても新しいプロトコルが必要になります。ただし、IPv6 の詳細を論じる前に、このような移行の必要性を正しく理解しておく必要があります。

IPv4のバックグラウンド

1981年秋、IETF (Internet Engineering Task Force) はIPv4仕様書 (RFC 791) を公刊しました。IPv4仕様書が公開された当時、インターネットは約1,000のシステムから構成されるコミュニティでした。IPv4仕様書では、すべてのIPアドレスが4つのグループの8ビットの数値から構成される32ビットの数値によって表現されることを要求していました。これにより、アドレスの総数は40億を超えることになりましたが、その階層的な割り当て方式により、実際に使用可能になるアドレス数は数億個にすぎなくなります。

IPv4の公開以来、インターネット人口は予想をはるかに上回る速度で増加し、たちまち1億台のコンピュータを超えることになりました。使用可能なアドレス数が減少するにつれ、IPv4アドレスを取得することはだんだん困難になります。さらに、現在の成長のペースは今後数年間もそのまま持続すると予想されています。

最終的には、インターネットのアドレスを割り当てようにも割り当てべきアドレスがなくなってしまうという事態になります。さらに、早ければ2002年にはこのような事態に至る可能性がある、という悲観的な見方もあります。

初期の IP 割り当てでは、一部の企業や研究機関のアドレスを非常に大きなブロック単位で確保していました。これらの「クラス A」および「クラス B」のネットワーク割り当ては、現在のような成長が予想できなかったときに発行されました。初期の段階でインターネットを採用した一部の団体が内部使用を目的として使用可能なアドレスを現在も保持している場合もありますが、未発行のアドレスのプールは日増しに小さくなっています。初期の段階で一部の大規模な企業内ネットワークに分配されたアドレスは、現在、他のユーザに再発行することはできません。

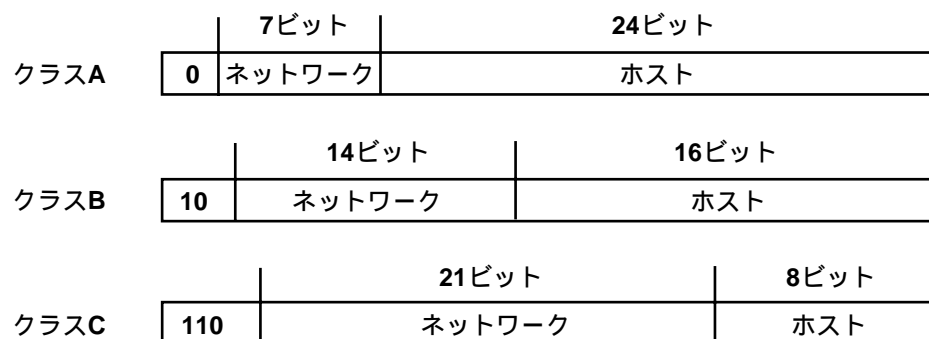


図 2-1 32 ビット IPv4 プロトコルでのネットワークおよびホスト識別子のマッピング

一時的な対策としての NAT

比較的短期的な対策として、NAT (Network Address Translation) は追加アドレス空間の圧力を緩和します。ローカルな使用のために予約されている一部の特殊なアドレス (10.*.*、や 192.168.*.* など) を使用することで、NAT はネットワーク管理者がユーザの巨大なコミュニティをファイアウォールや NAT ボックスの背後に隠すことを可能にします。インターネットのその他の部分からは、そのコミュニティ内のユーザのリクエストがすべて1つの NAT ボックスから送信されているように見えます。また、リクエストに対する応答が NAT ボックスに送り返されると、その情報は適切なローカルユーザに転送されます。複数の異なる企業内ネットワークで同一のローカルアドレスを再使用することができるため、NAT は新しい一意のインターネットアドレスに対する需要を小さくします。

残念ながら、NAT は一時的な対策にすぎず、永続的なソリューションではありません。NAT は、クライアントシステムの大規模なコミュニティの需要を満たしますが、それぞれが一意の永続的なアドレスを要求するインターネットサーバの需要を満たしません。同じ理由から、NAT はピア・ツー・ピア・コミュニケーションでは動作しません。また、NAT はインターネットのエンド・ツー・エンド・モデルに違反し、インターネットセキュリティの IPsec モデルという土台を突き崩します。

長期的なソリューションとしてのIPv6

IPv6 プロトコルは、IP アドレスの不足という差し迫った問題を解決すると同時に、IP アドレスの管理を容易にします。IPv6 プロトコルでは、8つのグループの16ビットの数値から構成されるアドレス方式を使用し、128ビットのネットワークアドレスを定義します。このアドレス方式により、全地球表面の1平方メートルあたり、約 6×10^{23} 個のアドレスを発行することができます。その結果、IPv6 プロトコルは将来の長期間にわたる存続が期待されます。

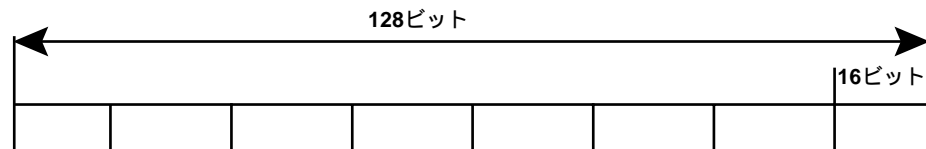


図 2-2 128 ビットの IPv6 アドレスフォーマット。それぞれの 16 ビットセグメントは 4 つの 16 進数によって識別され、それぞれのセグメントはコロンで区切られます。2 つ連続するコロンは、アドレスの中ですべてがゼロの 1 つまたは複数のセグメントを表現するために使用されます。

アドレスのプールがこのように大規模になるため、アドレスの一部を利用して、ネットワークの管理とルーティングを容易にすることが可能になります。新しいアドレス方式により、自動的なアドレスの設定と再設定が可能になります(サーバはすべてのクライアントにアクセスすることなくネットワークアドレスを振り直すことができようになり、モバイルクライアントはネットワーク内を自由に移動できるようになります)。プライベートアドレスを使用する必要がなくなるため、NAT サーバはもはや必要なくなります。それぞれのコンピューティングデバイスは、アドレス不足を心配することなく、そのデバイス独自の一意のアドレスを持つことができるようになります。

IPv6を使用した将来の計画

IPv6 仕様には、IPv6 コミュニケーションを可能にするいくつかの API が含まれていますが、大部分の API は IP のバージョンに依存しません。これらの API を使用することで、開発者は、IPv4 および IPv6 両方によるコミュニケーションをサポートする単一のセグメントのコードを書くことができます。コミュニケーションのターゲットとなるシステムの名前と現在のノードの設定に基づいて、API は、ターゲット IP アドレスと、それが IPv4 または IPv6 いずれのプロトコルを使用しているかを自動的に決定します。IP のバージョンに依存しない API を使用することで、開発者は IPv6 コミュニケーションを透過的に実行できるようになります。

特に短期的に最も頻繁に使用される API は「基本ソケット API (Basic Socket API)」です。アプリケーションをこのインタフェースに移植すると、ソケットコードが、現在の設定に基づいて、IPv4 または IPv6 のどちらを使ってデータをやり取りするかを決定します。その結

果、アプリケーションは、プロトコルのどちらのバージョンを選択するかを自動的に決定できるようにします。このため、さしあたりはIPv4の使用が可能になると同時に、将来IPv6を利用するために新しいコードを書く必要もありません。「一度コードを書くだけで両方のプロトコルに対応できる」というわけです。

ネームサーバはIPv4およびIPv6環境とそれらの協調を管理します。IPv6をサポートするようにネームサーバをアップグレードすると、環境の機能と設定に基づいて、リモートシステムの適切なアドレス（IPv4またはIPv6）が返されるようになります。ネームサーバを使ってIPv4およびIPv6に関する情報を管理すると、開発者にとって、コードの開発と管理が非常に容易になります。ネームサーバによる情報の管理の詳細については、「第4章 クライアントコードのIPv6へのポーティング」を参照してください。

Sun では、開発者を支援して現在のアプリケーションをIPv4からIPv6に変換するため、充実したツールのセットを用意しています。特に、Sunの「ソケット・コード・スクラバ」を使用すると、大きなコードのセグメントを検索して、IPv4ソケットに依存するルーチンを特定することができます。このツールは、標準的なIPv4ソケットルーチンに対する参照と主要なデータ構造体の識別子を検索します。ソケット・コード・スクラバを使用することで、開発者は変更を加える必要のある箇所をすばやく特定して、新しいインタフェースを使用するためにコードをアップグレードするプロセスを簡略化することができます。このプロセスの詳細については、「第6章 アプリケーションのアップグレードを支援するツール」を参照してください。

新たに開発を始める場合は、IPのバージョンに依存しないAPIを必ず使用してください。これらのインタフェースはIPv4とIPv6の両方をサポートするため、これらのAPIの使用により、IPv6標準へのシームレスなアップグレードパスが保証されます。この時点でIPv4インタフェースを使い続けると、アプリケーションにはじめから陳腐化を組み込むことになりません。

IPv6はインターネットの将来を担っています。問題は、IPv6がインターネットにとって将来の標準プロトコルであるかどうかということではなく、むしろ、IPv6への移行がいつ始まるかということです。開発者は、新しい開発に着手したり、現在のIPv4アプリケーションの変換を効率的に計画することで、いまずぐ将来への準備を整えてください。

IPv6 アプリケーション・プログラム・インタフェース

開発者は一般に新しいインタフェースを介して新しい機能にアクセスします。2つのバージョンの間でパラメータ（ノードアドレスなど）を変更する必要があるため、APIそのものは変わらざるをえません。ただし、IPv6を開発したエンジニアたちは、このプロセスを非常に単純なアップグレードにしてくれました。

IETF仕様

IETFでは、IPv6環境への移行に関するいくつかのRFC仕様書を公開しています。最も重要なものは、すべてのIPv6アプリケーションの90%から95%で使用される基本的なソケットコミュニケーションを規定しています。また、IETFでは現在、アドバンスドIPv6 APIに関するRFCの策定に取り組んでいます。アドバンスドIPv6 APIは追加のソケット機能と拡張機能を提供し、プログラマが制御できるレベルを大幅に拡張することになります。IPv6アプリケーションの10%未満がこの拡張機能を使用すると予想されています。

RFC 2553はIPv6の主要な仕様であり、基本的なソケットコミュニケーションを取り扱い、既存のソケットインタフェースに対するIPv6の拡張機能を規定しています。RFC 2553を使用するアプリケーションは下層にある特定のプロトコルから分離されることになるため、RFC 2553は基本的なAPIであるといえます。IPv4およびIPv6コミュニケーションの両方がサポートされるため、この仕様は現状のIPv4仕様に対する上位互換の拡張であると見なすことができます。一般に、IPv4呼び出しとIPv6呼び出しの間には1対1の対応関係が存在します。その結果、IPv6インタフェースを使用するように既存のIPv4ソケットベースのアプリケーションをアップグレードする作業は非常に簡単です。

基本ソケットAPI

かなりの期間にわたって、インターネット内にIPv4ホストとIPv6ホストの両方が混在し続けることはまずまちがいありません。このため、IPv6の基本ソケットAPIではIPv4とIPv6の両方をサポートします。このアプローチは、デュアル・スタック・インタフェースと呼ばれます。アプリケーションをIPv6ソケットインタフェースにアップグレードするだけで、IPv4システムおよびIPv6システムの両方とデータをやり取りするためにそれ以上のコードを書く必要はなくなります。新しいソケットインタフェースへの呼び出しを行うと、データ構造体が検査され、IPv6を使ってこのノードとデータをやり取りできるかどうか判断されます。それが不可能であると、ソケットは自動的にIPv4プロトコル接続を使って接続を確立します。現在使用されているすべてのインターネットソフトウェアはIPv4を使用しているため、デュアルスタックIPv6アプリケーションはIPv4を使って、現在使用されているすべてのソフトウェアとデータをやり取りすることができます。しかも、IPv4アプリケーションに対して何らかのコードを追加する必要はありません。

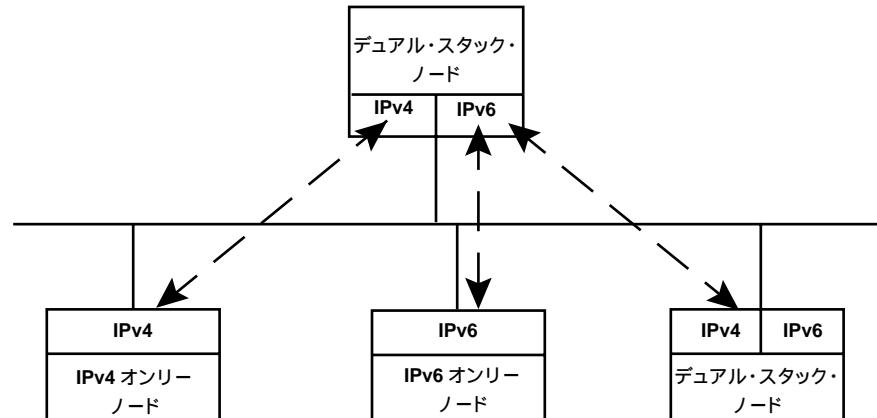


図 3-1 デュアル・スタック・ソケット・インタフェースは、IPv6 プロトコルが使用可能である場合、デフォルトの設定で IPv6 プロトコルを使用します。また、そうでない場合は、自動的に IPv4 プロトコルを使ってデータのやり取りを行います。

表3-1に、クライアントとサーバの機能に基づいたIPv4およびIPv6コミュニケーションの詳細な区分を示します。

アプリケーションのタイプ (ノードのタイプ)	IPv6-unawareサーバ (IPv4-only ノード)	IPv6-unawareサーバ (IPv6-enabled ノード)	IPv6-awareサーバ (IPv6-only ノード)	IPv6-awareサーバ (IPv6-enabled ノード)
IPv6-unaware クライアント (IPv4-only ノード)	IPv4	IPv4	x	IPv4
IPv6-unaware クライアント (IPv6-enabled ノード)	IPv4	IPv4	x	IPv4
IPv6-aware クライアント (IPv6-only ノード)	x	x	IPv6	IPv6
IPv6-aware クライアント (IPv6-enabled ノード)	IPv4	(IPv4)	IPv6	IPv6

表 3-1 IPv4 と IPv6 の相互運用性

この表の「x」印は、そのIPレイヤで対応するサーバとクライアントがデータをやり取りできないことを意味します。このようなコミュニケーションを可能にするには、プロキシのような他のツールを使用することができます。この表では、デュアル・スタック・インタフェースが対応するネームサービスの中にIPv4およびIPv6両方のアドレスを持っていることを前提としています。IPv6-unaware ノードはIPv4スタックのみを持ちます。IPv6-aware ノードはデュアルスタックと、IPv6が設定した少なくとも1つのインタフェースを持ちます。IPv6-onlyシステムはIPv6スタックのみを実装し、ネーム・サービス・データベース内にただ1つのアドレスを持ちます。

IPv6ソケット拡張機能を使ってIPv4ソケットアプリケーションを変換するのは容易なプロセスです。一般に、IPv4ソケット呼び出しをIPv6ソケット呼び出しにアップグレードするための直接的な変換方法が存在します。実際には、2つまたは3つのIPv4呼び出しを単一のIPv6呼び出しに置き換えることができるようなケースもありますが、必ずしもこのようなショートカットを使用する必要はありません。ソケット呼び出しの先頭の引数を変更し、さらにいくつかのパラメータをアップデートする必要はありますが、一般にそれは非常に単純なプロセスです。呼び出しのフォーマットとデータ構造体はよく似ており、変更が必要な箇所を特定するために使用できるツールも用意されています。

高度なソケット機能

すでに述べたように、IPv6のソケット仕様はRFC 2553で定式化され、インターネットコミュニケーションの主要な標準になることが期待されています。この仕様は完全なものであると見なされ、事実上IPv6に対するあらゆるニーズを処理することができます。RFC 2553は十分に定義されており、大多数のシステムベンダがRFC 2553のサポートを公約しています。

一方、現在の仕様を超えるいくつかの機能を追加するのが望ましいというコンセンサスも徐々に成立しつつあります。RFC 2292は、現在の仕様にいくつかの拡張機能を追加するように設計されており、開発者がIPv6 オンリーコミュニケーションをより詳細に制御することを可能にします。これらの付加的な制御要件は、全ソケットアプリケーションの約5%から10%でのみ使用されると見込まれています。たとえば、RFC 2553で定義されていないソースルーティング、ロー・ソケット・アクセス、インタフェース識別、機能拡張ヘッダなどの、高度な機能によって可能になるオペレーションがすべて考慮されています。

これらの高度な機能は、現在IETFによって評価されています。Sunでは、他のベンダとともに、これらすべての高度な機能の確定に協力しています。Sunでは、この仕様に関するより詳細な情報を今後も開発者のコミュニティに向けて発信し続ける予定です。

Sunリモートプロシージャ呼び出し

複数のアプリケーションがデータをやり取りするためのもう1つの方法は、リモートプロシージャ呼び出し(RPC)を介した方法です。Solarisオペレーティング環境に付属するRPC機能はプロトコルに依存しない形式で構築されています。このインタフェースを使用すると、アプリケーション開発者はこれらの呼び出しを使って、IPv6クライアントと透過的にデータをやり取りすることができます。

SunのRPC機能には、IPv6プロトコルを使ってデータをやり取りするための別の方法も用意されています。標準的な呼び出しを使用することで、アプリケーションは既存のRPC経路を使って、IPv4またはIPv6ノードのいずれかとデータをやり取りすることができます。SunRPCがIPv6を使ってデータをやり取りできるようにするための変更はすべてSunのライブラリに収められているため、アプリケーション開発者がアプリケーションのコードを書き直す必要はありません。最新のライブラリへの標準的なダイナミックリンクを使用することで、アプリケーションは必要に応じてIPv6を透過的に使用するようになります。

クライアントコードの IPv6 へのポーティング

インターネットを可能にする主要なテクノロジーの1つにネームサービスがあります。DNS (Domain Name Service)、NIS および NIS+ (Network Information Service) などのネーム・サーバ・テクノロジーは、論理参照 (コンピュータの名前など) を物理参照 (アドレス、名前など) に変換するために使用されます。IPv6 では、これらの名前からアドレスへの変換ルーチンを使って、IPv4 ベースのプロトコルから IPv6 ベースのプロトコルへの円滑な移行を支援します。

ネームサービスとIPv4からIPv6への移行

IPv4 から IPv6 への移行を検討し始めたとき、IETF はその計画が複雑な事業になる可能性があることをすぐに認識しました。このプロセスの簡略化を支援するため、IETF では多くの企業がより容易にこの移行を実現できるように1つの文書を作成しました。その結果成立した仕様が RFC 1933 です。

IETF では、純粋な IPv4 環境から IPv6 に対応した環境へのアップグレードを行うためにいくつかの方法を推奨しています。組織のニーズにしたがって、この移行は、環境内にあるクライアントの変換、環境内にあるサーバの変換、または環境間のルータの変換のいずれかから開始することができます。どこから移行を開始するかに関係なく、IPv6 に対応したネームサーバは移行の初期の段階から重要な要件の1つとなります。

ネームサーバは2つの IP 環境に接続し、どのプロトコルスタックがアクセスされるかを決定します。IPv6 に対応したネームサーバでは、IPv4 アドレスと IPv6 アドレスの両方に対応した名前からアドレスへの変換が格納されます。ネットワーク名からアドレスへの変換がリクエストされると、ネームサーバは、リクエストを行ったノード、ターゲットノード、および呼び出しパラメータに基づいて適切なアドレスを返します。リクエストノードとデスティ

ネーションノードの両方が IPv6 コミュニケーションに対応している場合（しかも呼び出しパラメータがそれを許可する場合）ネームサーバは IPv6 アドレスを返します。このアドレスを使用して、ソケット・インタフェース・コードは IPv6 プロトコルを使ったデータのやり取りを行います。環境全体が IPv6 に対応している場合、ネームサーバは通常使用するコミュニケーションプロトコルを自動的に IPv4 から IPv6 に変換します。

ノード名	ノードタイプ	IPv4 アドレス	IPv6 アドレス
earth	デュアルスタック	129.146.86.2	fe80::a00:20ff:fea1:6a83
venus	IPv4 オンリー	129.146.86.14	
mars	IPv6 オンリー		fe80::a00:20ff:fe87:8dde

表 4-1 あるネームサーバに対するデータ構造体がどのように類似しているかを示す例。ノードタイプは明示的に格納されませんが、その他の列の内容から容易に推定することができます。

IPv4 から IPv6 への移行ができるかぎり容易であることを保証するため、Sun では、Solaris 8 オペレーティング環境とともに出荷されるネームサーバをアップグレードして、IPv6 に対するサポートを組み込んでいます。これらのネームサーバは、IPv4 アドレスと IPv6 アドレス両方の情報をそのデータベースに格納し、それによって IPv4 環境と IPv6 環境の両方をサポートします。その結果、Solaris オペレーティング環境は IPv6 プロトコルへのアップグレードを開始するための出発点として最適な環境となります。

IPv4 と IPv6 の相互運用性

Sun が提供するツールを使用することで、IPv4 環境から IPv6 環境への変換は容易なものとなります。いくつかの IPv6 対応ネームサーバをインストールすれば、その後は、ネットワーク上のどこからでも IPv6 環境へのアップグレードを開始することができます。たとえば、IP ソケット呼び出しを IPv4 ソケット・オンリー・インタフェースからデュアル・スタック IP ソケット・インタフェースにアップグレードすることで、ノードは IPv4 オンリーシステムからデュアル・スタック・システムに移行されます。このようなデュアルスタック配列の例を 15 ページの図 4-1 に示します。

デュアルスタック IPv4/IPv6 インタフェースを使用するようにアプリケーションをアップグレードした後は、次のステップとして IPv6 情報の設定を行います。これを行うためには、ノードの IPv6 アドレスなどのパラメータを設定して、ノードが確実に IPv6 ネットワークに接続されるようにする必要があります。IPv6 が有効になると、そのノードを IPv6 ネームサーバに登録することができます。

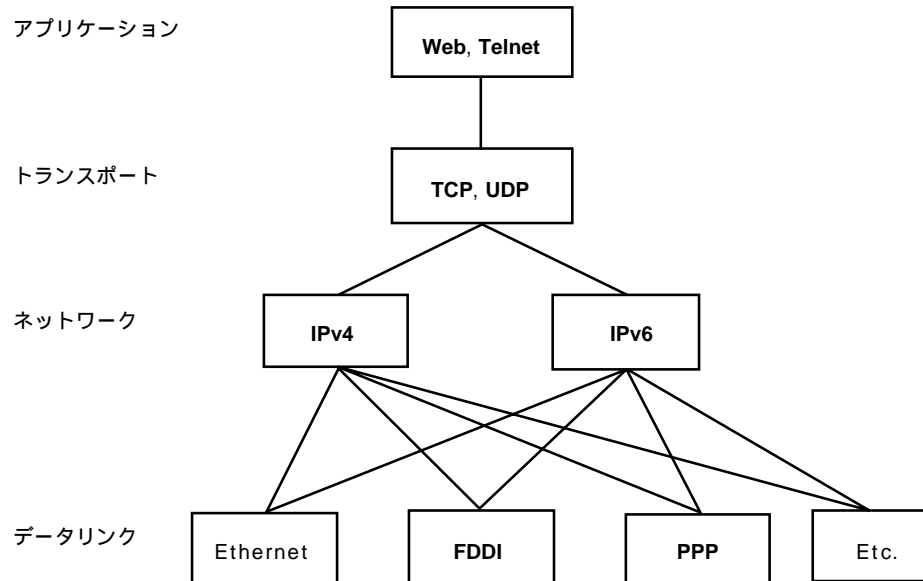


図 4-1 デュアル・スタック・プロトコル

ノードがネームサーバに登録されると、そのネームサーバはデスティネーションの IPv6 アドレスを返すようになり、パケットは IPv6 プロトコルを使ってルーティングされるようになります。どのデュアル・スタック・ノードも、ネーム・サーバ・データベースの中に IPv4 アドレスと IPv6 アドレスの両方を持ちます。IPv4 アドレスだけが格納されている場合、ネームサーバは自動的に IPv4 アドレスを返します。IPv6 アドレスだけ、または IPv4 アドレスと IPv6 アドレスの両方が格納されている場合、ネームサーバは IPv6 アドレスを返します。返されたアドレスに基づいて、アプリケーションは適切なプロトコルを自動的に選択し、そのノードに接続します。

IPv4 と IPv6 の API のちがい

IPv4 ルーチンから IPv6 ベースのルーチンへの変換を行うプロセスは非常に単純です。以下に、このプロセスを具体的に説明するコード例を示します。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
int
myconnect(char *hostname, int port)
```

```

{
struct sockaddr_in sin;
struct hostent *hp;
int sock;
/*
 * ソケットをオープンする。
 */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
perror("socket");
return (-1);
}

/*
 * ホストアドレスを取得する。
 */
hp = gethostbyname(hostname);
if (hp == NULL || hp->h_addrtype != AF_INET || hp->h_length != 4) {
(void) fprintf(stderr, "Unknown host _\"%s\"_n", hostname);
(void) close(sock);
return (-1);
}

sin.sin_family = AF_INET;
sin.sin_port = htons(port);
(void) memcpy((void *)&sin.sin_addr, (void *)hp->h_addr,
hp->h_length);
/*
 * ホストに接続する。
 */
if (connect(sock, (struct sockaddr *)&sin, sizeof(sin)) == -1) {
perror("connect");
(void) close(sock);
return (-1);
}
return (sock);
}

main(int argc, char *argv[])
{
int sock;
char buf[BUFSIZ];
int cc;
switch (argc) {
case 2:
sock = myconnect(argv[1], IPPORT_ECHO);

```

```

break;
case 3:
sock = myconnect(argv[1], atoi(argv[2]));
break;
default:
(void) fprintf(stderr,
"Usage: %s <hostname> <port>_n", argv[0]);
exit(1);
}
if (sock == -1)
exit(1);
if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
perror("write");
exit(1);
}
cc = read(sock, buf, sizeof (buf));
if (cc == -1) {
perror("read");
exit(1);
}
(void) printf("Read <%s>_n", buf);
return (0);
}

```

以上のコード例では、IPv4標準のgethostbyname() ルーチンを使用して、IPv4 ノード名をIPv4 アドレスに変換し、接続をオープンしています。この呼び出しを以下のコード例に示す呼び出しと比較してください。以下の呼び出しでは、IPv6 のgetipnodebyname() ルーチンを使用して、IPv6 ノード名をIPv6 アドレスに変換しています。

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
int
myconnect2(char *hostname, int port)
{
struct sockaddr_in6sin;
struct hostent*hp;
int sock, errnum;
/*
* ソケットをオープンする。
*/
sock = socket(AF_INET6, SOCK_STREAM, 0);
if (sock == -1) {

```

```

perror("socket");
return (-1);
}

/*
 * ホストアドレスを取得する。IPv4 にマップされた IPv6 アドレスであれば OK。
 */
hp = getipnodebyname(hostname, AF_INET6, AI_DEFAULT, &errnum);
if (hp == NULL) {
(void) fprintf(stderr, "Unknown host _\"%s\"_n", hostname);
(void) close(sock);
freehostent(hp);
return (-1);
}

/* すべての sockaddr_in6 フィールドがゼロであることを確認する */
bzero(&sin, sizeof (sin));
sin.sin6_family = hp->h_addrtype;
sin.sin6_flowinfo = 0;
sin.sin6_port = htons(port);
(void) memcpy((void *)&sin.sin6_addr, (void *)hp->h_addr,
hp->h_length);
freehostent(hp);

/*
 * ホストに接続する。
 */
if (connect(sock, (struct sockaddr *)&sin, sizeof (sin)) == -1) {
perror("connect");
(void) close(sock);
main(int argc, char *argv[])
{
int sock;
char buf[BUFSIZ];
int cc;
switch (argc) {
case 2:
sock = myconnect2(argv[1], IPPORT_ECHO);
break;
case 3:
sock = myconnect2(argv[1], atoi(argv[2]));
break;
default:
(void) fprintf(stderr,
"Usage: %s <hostname> <port>_n", argv[0]);
exit(1);
}
}

```

```

if (sock == -1)
exit(1);
if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
perror("write");
exit(1);
}
cc = read(sock, buf, sizeof (buf));
if (cc == -1) {
perror("read");
exit(1);
}
(void) printf("Read <%s>_n", buf);
return (0);
}

```

以上のコード例は、IPv4 と IPv6 の両方を処理するようにポーティングを行うときに `myconnect()` に加える必要のある最小限の変更を示しています。IPv4 ノードとデータをやり取りするときは、`sockaddr_in6` データ構造体に格納されている IPv4 にマップされたアドレスが使用されます。

異なる方法でポートを行うには、以下のコード例に示すように新しい API 関数である `getaddrinfo(3N)` と `getnameinfo(3N)` を使用します。このコード例はアドレスファミリーに関する知識を必要としません。またもう1つの利点として、ネームサーバから返されたすべてのアドレスを試して、`connect()` が正常に実行されるアドレスを検出するため、このコード例の動作ははるかに安定しています。

```

int
myconnect3(char *hostname, char *servicename)
{
struct addrinfo*res, *aip;
struct addrinfohints;
int sock = -1;
int error;

/*
 * ホストアドレスを取得する。どのタイプのアドレスでも OK。
 */
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
(void) fprintf(stderr,
"getaddrinfo: %s for host %s service %s_n",
gai_strerror(error), hostname, servicename);

```

```

return (-1);
}
for (aip = res; aip != NULL; aip = aip->ai_next) {
/*
* ソケットをオープンする。アドレスのタイプはgetaddrinfo() が返す
* 値によって異なる。
*/
sock = socket(aip->ai_family, aip->ai_socktype,
aip->ai_protocol);
if (sock == -1) {
perror("socket");
freeaddrinfo(res);
return (-1);
}
/*
* ホストに接続する。
*/
if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
perror("connect");
(void) close(sock);
sock = -1;
continue;
}
break;
}
freeaddrinfo(res);
return (sock);
}
main(int argc, char *argv[])
{
int sock;
char buf[BUFSIZ];
int cc;

switch (argc) {
case 1:
sock = myconnect3(NULL, NULL);
break;
case 2:
sock = myconnect3(argv[1], "echo");
break;
case 3:
sock = myconnect3(argv[1], argv[2]);
break;
default:
(void) fprintf(stderr,
"Usage: %s <hostname> <port>_n", argv[0]);

```

```

exit(1);
}
if (sock == -1)
exit(1);

if (write(sock, "hello world", strlen("hello world") + 1) == -1) {
perror("write");
exit(1);
}
cc = read(sock, buf, sizeof (buf));
if (cc == -1) {
perror("read");
exit(1);
}
(void) printf("Read <%s>_n", buf);
return (0);
}

```

以上のように、これら3つのコード例の違いはごくわずかであり、IPv4ネームサーバ環境からIPv6ネームサーバ環境へのポーティングはどちらかといえばかなり単純です。

より一般的にいえば、これら3つのコード例は、アプリケーションをIPv4のみの環境からデュアルスタックのIPv4/IPv6対応環境にポーティングするために必要な変更のタイプを具体的に示しています。

サーバコードの IPv6 へのポーティング

前の章の最後にあげたコード例は、クライアント側のコードが IPv4 と IPv6 の間でどう変わるかを具体的に示していました。サーバアプリケーションを移行して、IPv4 および IPv6 クライアントアプリケーションとデータをやり取りできるようにするために必要な変更もその単純さのレベルではほぼ同じです。

まず、以下のような IPv4 サーバアプリケーションを検討してみましょう。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#define BACKLOG 1024 /* ペンディング接続の最大数 */

void do_work(int sock);

int
myserver(int port)
{
    struct sockaddr_inladdr, faddr;
    int sock, new_sock, sock_opt;
    socklen_t faddrrlen;
```

次に、これを IPv4/IPv6 デュアルスタックと比較してみましょう。

```
#include <sys/types.h>
```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
void do_work(int sock);

int
myserver2(int port)
{
    struct sockaddr_in6 laddr, faddr;
    int sock, new_sock, sock_opt;
    socklen_t faddrlen;
    char addrbuf[INET6_ADDRSTRLEN];

    /*
     * ソケットをセットアップして、接続に対するリスニングを開始する。
     */
    /* すべての sockaddr_in6 フィールドがゼロであることを確認する */
    bzero(&laddr, sizeof (laddr));
    laddr.sin6_family = AF_INET6;
    laddr.sin6_flowinfo = 0;
    laddr.sin6_port = htons(port);
    laddr.sin6_addr = in6addr_any; /* 構造体の割り当て */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return (-1);
    }

    /*
     * システムにローカルアドレスが再使用可能であることを通知する。
     */
    sock_opt = 1;

    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,
        sizeof (sock_opt)) == -1) {
        perror("setsockopt(SO_REUSEADDR)");
        (void) close(sock);
        return (-1);
    }

    if (bind(sock, (struct sockaddr *)&laddr, sizeof (laddr)) == -1) {
        perror("bind");
        (void) close(sock);
        return (-1);
    }

```

```

}

if (listen(sock, BACKLOG) == -1) {
perror("listen");
(void) close(sock);
return (-1);
}
/*
* 接続リクエストを待つ。
*/
for (;;) {
faddrlen = sizeof (faddr);
new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
if (new_sock == -1) {
if (errno != EINTR && errno != ECONNABORTED) {
perror("accept");
}
continue;
}

if (IN6_IS_ADDR_V4MAPPED(&faddr.sin6_addr)) {
struct in_addr ina;

IN6_V4MAPPED_TO_INADDR(&faddr.sin6_addr, &ina);
(void) printf("Connection from %s/%d_n",
inet_ntop(AF_INET, (void *)&ina,
addrbuf, sizeof (addrbuf)),
ntohs(faddr.sin6_port));
} else {
(void) printf("Connection from %s/%d_n",
inet_ntop(AF_INET6, (void *)&faddr.sin6_addr,
addrbuf, sizeof (addrbuf)),
ntohs(faddr.sin6_port));
}
do_work(new_sock); /* 何らかの動作を行う */
}

/* NOTREACHED */
}

void
do_work(int sock)
{
char buf[BUFSIZ];
int cc;

while (1) {

```

```

cc = read(sock, buf, sizeof (buf));
if (cc == -1) {
perror("read");
exit(1);
}
if (cc == 0) {
/* EOF */
(void) close(sock);
(void) printf("Connection closed_n");
return;
}

buf[cc + 1] = '_0';
(void) printf("Read <%s>_n", buf);

if (write(sock, buf, cc) == -1) {
perror("write");
exit(1);
}
}

main(int argc, char *argv[])
{
if (argc != 2) {
(void) fprintf(stderr,
"Usage: %s <port>_n", argv[0]);
exit(1);
}
(void) myserver2(htons(atoi(argv[1])));
return (0);
}

```

以上のコード例では、アプリケーションは単一のソケットを使ってIPv4およびIPv6クライアントとデータをやり取りします。これは、AF_INET6ソケットをin6addr_anyにバインドすることによって可能になります。

なお、このサーバはIPv4ノードのアドレスをマップされたアドレス（先頭に「::ffff:」という文字列を含む）として出力します。この動作は、IN6_IS_ADDR_V4MAPPEDマクロを使用し、マップされたアドレスとは異なる方法でinet_ntopを呼び出すことで変更することができます。

最後のコード例は、新しいgetaddrinfo()およびgetnameinfo()呼び出しを使って、前のコード例と同じサーバを作成する方法を示しています。これらの呼び出しはアドレス固有の処理をコードから削除します。また、以下のコード例は、ピアのアドレスを数値形式およびホスト名、サービス名として出力します。

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#define BACKLOG 1024 /* ペンディング接続の最大数 */

void do_work(int sock);

int
myserver3(char *servicename)
{
    struct addrinfo*aip;
    struct addrinfohints;
    struct sockaddr_storagefaddr;
    int sock, new_sock, sock_opt;
    socklen_tfaddrlen;
    int error;
    char hname[NI_MAXHOST];
    char sname[NI_MAXSERV];
    /*
     * ソケットをセットアップして、接続に対するリスニングを開始する。
     */
    bzero(&hints, sizeof (hints));
    hints.ai_flags = AI_ALL|AI_ADDRCONFIG|AI_PASSIVE;
    hints.ai_socktype = SOCK_STREAM;

    error = getaddrinfo(NULL, servicename, &hints, &aip);
    if (error != 0) {
        (void) fprintf(stderr, "getaddrinfo: %s for service %s_n",
            gai_strerror(error), servicename);
        return (-1);
    }
    sock = socket(aip->ai_family, aip->ai_socktype, aip->ai_protocol);
    if (sock == -1) {
        perror("socket");
        return (-1);
    }
    /*
     * システムにローカルアドレスが再使用可能であることを通知する。
     */
    sock_opt = 1;

    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,

```

```

sizeof (sock_opt) == -1) {
perror("setsockopt(SO_REUSEADDR)");
(void) close(sock);
return (-1);
}

void
do_work(int sock)
{
char buf[BUFSIZ];
int cc;

while (1) {
cc = read(sock, buf, sizeof (buf));
if (cc == -1) {
perror("read");
exit(1);
}
if (cc == 0) {
/* EOF */
(void) close(sock);
(void) printf("Connection closed_n");
return;
}
buf[cc + 1] = '_0';
(void) printf("Read <%s>_n", buf);

if (write(sock, buf, cc) == -1) {
perror("write");
exit(1);
}
}

main(int argc, char *argv[])
{
if (argc != 2) {
(void) fprintf(stderr,
"Usage: %s <servicename>_n", argv[0]);
exit(1);
}
(void) myserver3(argv[1]);
return (0);
}

```

以上の使用例は2つのプロトコル固有のソケットの具体例を示していますが、プログラムは通常このようなアプローチを採用しないと考えられます。通常、プログラムは単一のソケッ

ト呼び出しを使って、デュアル・スタック・ソフトウェアにIPv4およびIPv6の使用を管理させます。

アプリケーションのアップグレードを支援するツール

できるかぎり容易に IPv4 から IPv6 へのアップグレードを行えるように、Sun ではいくつかの変換ツールを開発しました。その中でも最も重要なものは Sun のソケット・コード・スクラバです。このツールは、一覧の既存のソースファイルを対象に実行でき、IPv4 呼び出しから IPv6 呼び出しへのアップグレードを行うためにコードの変更が必要になると考えられる箇所を指摘します。このツールを使用すると、プログラマは既存のアプリケーションをスキャンして、IPv6 プロトコルを使ったデータのやり取りに必要な変更を特定することができます。

Socket Code Scrubber for C/C++

事実上、今日のアプリケーションはすべて、プログラムまたは一連のプログラムとして結合された一連のモジュラサブルーチンおよびライブラリファイルから構成されています。これらのモジュールの大部分は、TCP/IP を介したデータのやり取りとはまったく関係ありません。IPv6 を介してデータのやり取りを行うようにアプリケーションをアップグレードするためには、実際に IP ルーチン呼び出ししているモジュールを見つけ出す必要があります。Sun のソケット・コード・スクラバはこのようなモジュールの特定に役立ちます。

コマンドプロシージャを作成して、大規模なアプリケーションに含まれるすべてのモジュールをスキャンすることができます。このプロシージャはバックグラウンドまたはバッチモードで実行され、すべてのソースファイルを 1 度にスキャンします。バッチの実行が完了すると、修正を必要とするすべてのモジュール名と、それぞれのモジュール内の行を指摘するログファイルが生成されます。

アプリケーションのコードはそれぞれ異なるため、それにしたがってアップグレードに必要な作業も異なりますが、ソケット・コード・スクラバは通常、コード内の 10 行程度を

変更すべきであると指摘します。これらは、API に対するパラメータを指定していたり、ルーチンが自分自身を呼び出している行です。

Socket Code Scrubberの使用例

ソケット・コード・スクラバは、コードモジュール内であらかじめ定義されているキーワード変数のリストを検索します。これらのキーワードには、主要な IPv4 パラメータとルーチン呼び出しのすべてが含まれています。キーワードが検出されると、対応するコードセクションが表示されます。これらのキーワードすべてが検出されると、開発者は編集を必要とする箇所のリストを取得できます。

以下に、ソケット・コード・スクラバの出力例を示します。この例では、プログラムは、「第 4 章 クライアントコードの IPv6 へのポーティング」の「IPv4 と IPv6 の API のちがひ」で使用した `myconnect.c` という名前の IPv4 ベースのサンプルプログラムを対象に実行されています。

```
*****
myconnect.c
*****
11: struct sockaddr_in sin;
18: sock = socket(AF_INET, SOCK_STREAM, 0);
27: hp = gethostbyname(hostname);
28: if (hp == NULL || hp->h_addrtype != AF_INET || hp->h_length != 4) {
34: sin.sin_family = AF_INET;
35: sin.sin_port = htons(port);
36: (void) memcpy((void *)&sin.sin_addr, (void *)hp->h_addr,
```

ソケット・コード・スクラバではキーワード検索を使用するため、適切なコーディング標準を使ってプログラムされているアプリケーションで最善の結果が得られることは明らかです。アプリケーション開発者が定義済みの定数を使用する代わりに、パラメータに対するハードコーディング変数コンテンツのような非標準的な方法を使用していると（「AF_INET」の代わりに「2」を使用する場合など）、ソケット・コード・スクラバは変更が必要な領域を正しく指摘できなくなってしまいます。モジュールコーディングのガイドラインとして、ネットワーク固有のコードは、一般的なアプリケーションコードに混在させるのではなく、独立したネットワーク固有のモジュールの中にまとめるようにしてください。このようなルーチンでは必要なコードの修正が予想しやすくなります。

まとめ

今後、IPv4がIPv6によって置き換えられることは確実です。ただ1つの問題はそれがいつ行われるかということです。インターネットの急速な成長により、使用可能なアドレスのプールは急速に小さくなりつつあります。また、NATなどのツールが一時的な改善策をもたらすとしても、現実的なソリューションは、あくまでも使用可能なアドレスの数を増加させ、より容易な設定を可能にする新しいIPアドレス標準です。

ソフトウェア開発者は、プロトコルに依存しないIPv6の開発に焦点を絞り、アプリケーションが将来にわたって生き続けることを保証する必要があります。たとえ現在のネットワークがまだIPv4ベースであるとしても、新しいアプリケーションはすべて新しいインタフェースを使って開発し、IPv6をサポートするようにしてください。また、これらの新しいインタフェースを使用することで、当面にわたってIPv4コミュニケーションのサポートを提供し続けることもできます。

顧客が完全なIPv6ネットワークの実装を開始した場合にも、新しいインタフェースを使用したアプリケーションはそのまま使用し続けることができるという大きな利点があります。

新しいバージョンを開発するまでの過渡的な措置として、新しいIPv6インタフェースを使用するように既存のアプリケーションをアップグレードすることもできます。Sunのソケット・コード・スクラバを使用すると、アプリケーション開発者は潜在的にアップグレードを必要とする領域を特定することができます。IPv4呼び出しとIPv6呼び出しの間には直接的な対応関係が存在するため、IPv4からIPv6への変換は比較的容易に実行できます。

Sunでは、IPv6の準備と移行を支援するためのツールと専門知識を提供します。SunのSolarisオペレーティング環境はこの移行にとって理想的なプラットフォームであるといえます。テクニカルおよび商用UNIX®の中で最大のインストール数を誇るSolaris 8オペレーティング環境には、コードを即座に移行できるように、デュアル・スタック・ネーム・サービスとさまざまなライブラリが含まれています。15年を超えるネットワーク開発の経験を持ち、さ

さまざまな革新を実現してきた Sun では、IPv6 への変換を容易かつわかりやすいものにする
数々のツールとテクノロジーを開発者のコミュニティに提供します。

付録

IPv4からIPv6へのソケットの変更

IPv6でのソケットAPIの変更と新しい機能はIETFが作成した2つの文書に記載されています。

- RFC 2553 - IPv6 の基本的なソケットインタフェース拡張機能
- 公刊準備中 - IPv6 の高度なソケットAPI (RFC 2292)¹

さまざまなアドレスファミリとアドレスの長さを処理するためのより適切な方法を提供するだけであるため、ソケットAPIに加えられた拡張の一部はIPv6に依存していません。その他の拡張機能は、ソースルーティングなど、IPv6固有の機能へのアクセスを可能にします。

これら2つの文書に加えてRFC 1933も重要な文書です。この文書は、IPv6環境の展開方法を決定するときに役立ちます。

IPv4にマップされたアドレスの使用と組み合わせるとき、ソケットAPIに加えられた変更の大部分は、アプリケーションをIPv6対応にするために必要な変更作業を容易なものにします。

1. 現在、IETF内ではRFC 2292の見直し作業が行われています。ただし、ソケットベースのアプリケーションの90%は、RFC 2553 (IPv6の基本的なソケットインタフェース拡張機能)で規定されているIPv6の拡張機能だけを使って十分に開発できるはずで

機能	現在のIPv4	デュアルIPv4/IPv6	説明
プロトコルファミリ socket(3N) の先頭の引数	AF_INET/PF_INET	AF_INET6/PF_INET6	アドレスファミリとプロトコルファミリは相互に交換して使用される
アドレスファミリ	AF_INET	AF_INET6	sockaddr ファミリフィールド
Inet アドレス構造体	sockaddr_in	struct sockaddr_in6	sockaddr_in6 は sockaddr よりも長くなる
ジェネリックアドレス構造体	struct sockaddr	struct sockaddr_storage	記憶領域を割り当てるために使用する時のみ
IP アドレス構造体	struct in_addr	struct in6_addr	
ループバックアドレス	INADDR_LOOPBACK	in6addr_loopback IN6ADDR_LOOPBACK_INIT	定数は構造体を初期化するためだけに使用できる
リスナと受信者をバインドするためのワイルドカードアドレス	INADDR_ANY	in6addr_any IN6ADDR_ANY_INIT	定数は構造体を初期化するためだけに使用できる
名前からアドレスへ	gethostbyname(3N)	getipnodebyname(3N)	IPv6 アドレスまたは IPv4 アドレスのいずれかを取得する
アドレスから名前へ	gethostbyaddr(3N)	getipnodebyaddr(3X)	関数はすでにファミリパラメータを持っている
getipnodeby* によって返されたデータ構造体の解放	-	freehostent(3N)	getipnodeby* はマルチスレッドに対応している
アドレスから名前へ	-	getaddrinfo(3N)	アプリケーションをアドレスフォーマットから分離する
getaddrinfo によって返されたデータ構造体の解放	-	freeaddrinfo(3N)	getaddrinfo はマルチスレッドに対応している
名前からアドレスへ	-	getnameinfo(3N)	アプリケーションをアドレスフォーマットから分離する
レポートエラー	-	gai_strerror(3N)	getaddrinfo および getnameinfo のエラー
文字列からアドレスへ	inet_addr(3N)	inet_pton(3N)	追加されたファミリパラメータ
アドレスから文字列へ	inet_ntoa(3N) - -	inet_ntop(3N) INET_ADDSTRLEN INET6_ADDSTRLEN	追加されたファミリパラメータ。最大の文字列バッファサイズに対する定数
TTL に対するソケットオプション	IP_TTL	IPV6_UNICAST_HOPS	ttl/hop の制限値を設定する
予約されているポートの取得	rresvport(3N)	rresvport_af(3N)	rcmd(3N) によって使用される
リモートホストでのコマンドの実行	rcmd(3N)	rcmd_af(3N)	アドレス・ファミリ・パラメータを必要とする

表 8-1 対応する IPv6/ デュアル構造体と定数

IPv4とIPv6のちがい

IPv4の一部の機能はIPv6には存在しません。たとえば、IPv4パケットフォーマットへの直接的なマップを行う機能やIPv4プロトコルオプションはIPv6には存在しません。

- IPv4のType of Serviceフィールドとその上位のフィールドは、IPv6ではTraffic Classフィールドに置き換えられた
- IPv6にはレコード・ルート・オプションはない

その他の機能はIPv6にも存在しますが、インタフェースが非常に異なっていたり、IPv4アーキテクチャからはすでに削除されていたりします。たとえば、IPv4でネットワーク番号を処理していたマクロや関数などがそれです（表8-2を参照）。サブネットの導入（RFC 950）からクラスに依存しないドメイン間ルーティング（CIDR）へのゆるやかな移行に伴い、IPv4インターネットはもはや、ネットワーク番号やIPアドレスの異なるクラスという概念に依存しなくなりつつあります。それに代わって、すべてのルーティングは、IPアドレスに追加される任意のプレフィックスを使って実行されています。IPv6は、IPv4で開発されたクラスに依存しないルーティングに基づいているため、IPv6にはクラスまたはネットワーク番号という概念はありません。

機能	IPv4インタフェース	説明
クラスに対応したアドレス	IN_CLASS_A,B,C inet_makeaddr(3N) inet_network(3N) inet_lnaof(3N) inet_netof(3N)	IPv6にはアドレスクラスという概念は存在しない（ただし、ユニキャストとマルチキャストの区別は除く）
IPプロトコルフィールドへのアクセス	IP_OPTIONSソケットオプション IP_RECVDSTADDR IP_RECVOPTS	ソースルーティングとオプション関数によって置き換えられた
IPプロトコルフィールドへのアクセス	IP_HDRINCLソケットオプション	もはや必要ない。IPv6ソケットオプションにより、アプリケーションは、すべてのIPv6オプションおよび拡張ヘッダの設定と取得を行うことができる
IPプロトコルフィールドへのアクセス	IP_TOSソケットオプション	Type Of Serviceフィールドは、IPv6のTraffic Classフィールドに置き換えられた。このフィールドは、sin6flowinfoフィールドを使って設定できる

表 8-2 IPv6に対応する機能がないIPv4ソケットの機能

参考文献

RFC 1886 — DNS Extensions to Support IP Version 6, S. Thomson, Bellcore, C. Huitema, INRIA, December 1995

RFC 1933 — Transition Mechanisms for IPv6 Hosts and Routers, R. Gilligan, E. Nordmark, Sun Microsystems, April 1996

RFC 2373 — IP Version 6 Addressing Architecture, R. Hinden, Nokia, S. Deering, Cisco Systems, July 1998

RFC 2460 — Internet Protocol, Version 6 (IPv6) Specification, S. Deering, Cisco, R. Hinden, Nokia, December 1998

RFC 2461 — Neighbor Discovery for IP Version 6 (IPv6), T. Narten, IBM, E. Nordmark, Sun Microsystems, W. Simpson, Daydreamer, December 1998

RFC 2462 — IPv6 Stateless Address Autoconfiguration, S. Thomson, Bellcore and T. Narten, IBM, December 1998

RFC 2463 — Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, A. Conta, Lucent, S. Deering, Cisco Systems, December 1998

RFC 2553 — Basic Socket Interface Extensions for IPv6, R. Gilligan, FreeGate, S. Thomson, IBM, J. Bound, Compaq, W. Stevens, Consultant, March 1999



Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

1 (800) 786.7638
1.512.434.1511

<http://www.sun.com/solaris/>