

本書の目指すところと目指さないところ

本書は、UNIX ソケット API を使用して IPv6 対応アプリケーションを作成したり、既存の IPv4 アプリケーションを IPv6 にも対応させる方法を解説していきます。本書では、この目的にかなった、移植性が高く、かつセキュリティホールの生じにくい方法を紹介します。

移植性の高いプログラムを書こう

ソケット API は、たくさんのプラットフォームでサポートされています。ソケット API でアプリケーションを書くとき、誰もがそのプログラムをできるだけ多くのプラットフォームで動くようにしたいと考えるのではないのでしょうか。そのため、アプリケーションプログラミングにおいては移植性が重要な要素となります。

ご存じのとおり、UNIX 系の OS にはさまざまな種類があり、どれでもソケット API を利用できます。また、UNIX 以外にもソケット API を使用できる OS は多数あります。たとえば Windows XP もそのひとつで、ソケット API が実装されています。MacOS X は、OS 自体に BSD UNIX を使っていて、ユーザーにソケット API を提供しています(もっとも Apple は、普通は Apple 独自の API を使うように推奨していませんけどね)。

そのようなわけで、本書では IPv6 に対応したプログラムを書く場合には、移植性のある方法をお勧めしたいのです。

セキュリティに気を遣ったプログラムを書こう

今日、セキュリティはインターネットにおける重大な関心事となっています。ネットワーク管理者なら、きっと毎日のようにたくさんのスパムメールやウィルスメール、ベンダーからのセキュリティ勧告を受け取っていることでしょう。インターネットという基盤の安全を守るために、すべての開発者はセキュリティに対する構えが必要です。すなわち、コードのすべての行をできるだけ厳密に検査し、正しい API を使って

本書の目指すところと目指さないところ

正確で安全なコードを書くという姿勢をとらなくてはなりません。

このため、本書のなかでとりあげたプログラム例は、正確を期すよう注意しました。いずれの例も、セキュリティに配慮して実装されています。また本書では、IPv6 標準 API を使うように推奨していますが、標準 API の一部にもともと安全ではないものがあるので、そのような API の使用は避けるようにしています。

基礎知識

本書は、IPv6 技術のすべてを論じようとするものではありません。IPv6 技術自体に関しては、たいへん多くの文献があります。IPv6 についてまったく予備知識がない場合には、本書に取り組みまえに、読んでおくとよいでしょう。このあとの「参考文献」に、いくつかの書籍を挙げます。

また本書では、読者がある程度ソケットプログラミングに経験があることを想定しています。ソケット API のすべてを説明することは意図していません。ソケット API の詳細な解説書としてお薦めする文献を「参考文献」に挙げます。

第 2 章

IPv6 ソケットプログラミング

2.1 IPv6のためのアドレスファミリ、AF_INET6

1.3 「UNIX ソケットプログラミング」でみたように、ソケット API では IPv4 のソケットを認識するために、AF_INET という定数を使ってきました。また、ソケット上で IPv4 の通信相手を認識するためには、sockaddr_in という構造体を使っていました。

ソケット API で IPv6 を取り扱うには、AF_INET6 という定数を使います。これまで、

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

と記述していたところを、

```
s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
```

とすると、s には IPv6 を取り扱えるソケットが生成されます。

以下は、sockaddr_in と sockaddr_in6 の定義です：

sockaddr_in の定義

```
struct sockaddr_in {
    u_int8_t      sin_len;          /* sockaddr長 */
    u_int8_t      sin_family;      /* アドレスファミリ */
    u_int16_t     sin_port;        /* TCP/UDPポート番号 */
    struct in_addr sin_addr;       /* IPv4アドレス */
    int8_t       sin_zero[8];     /* パディング */
};
```

sockaddr_in6 の定義

```

struct sockaddr_in6 {
    u_int8_t      sin6_len;          /* この構造体の長さ(socklen_t)*/
    u_int8_t      sin6_family;      /* AF_INET6 (sa_family_t) */
    u_int16_t     sin6_port;        /* トランスポート層ポート番号 */
    u_int32_t     sin6_flowinfo;    /* IP6フロー情報 */
    struct in6_addr sin6_addr;      /* IP6アドレス */
    u_int32_t     sin6_scope_id;    /* スコープゾーンインデックス */
};

```

ソケット上で IPv6 の通信相手を認識するためには、sockaddr_in6 という構造体を使います。たとえば、AF_INET6 を指定して生成したソケットに対して connect(2) などの操作をするときに、sockaddr_in6 構造体を使います。

sockaddr_in6 には sin6_flowinfo と sin6_scope_id というフィールドが増えています。sin6_flowinfo は標準化がまだ済んでいないフィールドですので、本書では取り扱いませんが、使用しないときには 0 にしておきます。sin6_scope_id はホストバイトオーダーで表現しますが、2.3.1「アドレスは sockaddr 型の構造体に格納しよう」で詳しく説明します。

2.2 なぜプログラムはアドレスファミリから独立していなければならないのか？

本書では、プログラムを IPv6 に対応させる方法として、アドレスファミリに依存しないソケットプログラミングを勧めています。AF_INET6 や AF_INET などのアドレスファミリやそれらに関する知識をプログラム中に直書きすることには、以下に挙げるようなたくさん問題点があります。そこで、本書では AF_INET や AF_INET6 といったアドレスファミリに依存しない「アドレスファミリ独立」な方法を探ります。これから説明していくとおりしていけば、あなたのコードはアドレスファミリ独立になるはずです。

コードをアドレスファミリ独立にする理由として、次のようなことを挙げられます：

2.2 なぜプログラムはアドレスファミリから独立していなければならないのか？

デュアルスタック環境をサポートするためには、プログラムは IPv4 と IPv6 の両方を扱えなければなりません。AF_INET や AF_INET6 を決め打ちにすると、デュアルスタック環境ではうまく動かないプログラムができあがります。

新しいプロトコルに対応させるときには、ネットワークアプリケーションをまた書き直さないですむようにしたいものです。これは、IP 層についても言えるし、トランスポート / セッション層についても言えます。たとえば IPv7——今のところ導入の予定はないけれど、先のことはわからないから——に対応させたり、TCP の代わりに SCTP を使うことが考えられます。またシステムによっては、アドレスファミリ独立 API を使うことによって、Appletalk もサポートできるようになるかもしれません。

アドレスファミリ独立プログラミングのために、十分な手立てが用意されています。具体的な方法としてたとえば、sockaddr_storage や getaddrinfo(3)、getnameinfo(3)などの API を挙げられます。

アドレスファミリをプログラム中に直書きすると、そのアドレスファミリがサポートされていない OS では動作しなくなってしまいます。アドレスファミリ独立なプログラムなら、ソースコードにしてもバイナリコードにしても、1つのプログラムをどんな設定の OS 上でも動かすことができます。

経験上、この方法で書くほうが、IPv6 専用を書くよりもきれいで移植性のあるプログラムになります。

gethostbyname2(3)のようなアドレスファミリ依存の API は、スコープつき IPv6 アドレスをサポートしていません。getaddrinfo(3)や getnameinfo(3)ならサポートしています。

23ページに IPv4 を直書きしたプログラムを示します。アミかけの部分が IPv4 に依存しています。

他の参考文献では、次のプログラム例のように、アドレスファミリ AF_INET を AF_INET6 に置き換えることを勧めているかもしれません。またそのような文書で

は、`gethostbyname(3)`を `gethostbyname2(3)`に入れ替えるように勤めているでしょう。`gethostbyname2(3)`は、`AF_INET` 以外のアドレスファミリーを取り扱えるので、第2引数に `AF_INET6` を指定することで IPv6 アドレスの DNS 検索ができます。しかし、こうした方法にはいろいろと欠点があります。

第一に、`gethostbyname2(3)`では、IPv6 の宛先にしか接続できません。IPv4 の宛先には接続できないのです。IPv4/v6 デュアルスタック環境では、FQDN から IP アドレスへの名前解決の結果、IPv4 アドレスや IPv6 アドレスが複数得られる可能性があります。クライアントプログラムは、IPv6 アドレスだけでなく、すべてのアドレスに接続を試みるべきです。

第二に、前にも述べたとおり、IPv6 にはスコープつきアドレスの概念があります。しかし、`gethostbyname2(3)`はスコープ ID を返さないの、スコープつき IPv6 アドレスを扱えないのです。

第三に、`AF_INET6` にハードコーディングしているので、このプログラムは IPv6 に対応したカーネル上でしか動作しません。IPv6 に対応していないカーネルは普通、`AF_INET6` ソケットをサポートしないからです。プログラムをコンパイルし直すことなく、IPv4 だけのカーネルでも、IPv6 だけのカーネル上でも、また IPv4/v6 デュアルスタックカーネル上でも動くようにしたければ、やはりアドレスファミリー独立なプログラムを書く必要があります。

第四に、この方法ではコードを将来にわたって使用していくことは望めません。今後、新しいプロトコルが登場するときにも、既存のアプリケーションは書き直さないようにしたいものです。IPv6 への移行にはコストがかかりますから、IPv6 への移行と同時に他の問題も解決してしまいたいと考えているのです。そうすれば、今後再びネットワークにアクセスするコードを修正する必要がなくなるでしょう。

最後に、経験上、アドレスファミリー独立の方法で書いたアプリケーションは、移植性と安定性が高くなります。

そのようなわけで、本書では `AF_INET6` をハードコーディングするのはお勧めせず、アドレスファミリー独立なプログラミングをお勧めしているのです。次節から、アドレスファミリー独立なプログラミング方法について述べていきます。