ネットワークインタフェース

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part Number 806-2730-10
2000 年 3 月

原典: *Network Interface Guide*

Part No: 806-1017-10

Revision A

Adobe PostScript

目次

表

図

はじめに

このマニュアル『ネットワークインタフェース』は、分散型アプリケーションを実装するための基本的な機能について説明し、それらの実際の使用方法を示しています。

このマニュアルで紹介するユーティリティとそのオプション、およびライブラリ機能はすべて、SunOS リリース 5.8 のものです。SunOS 5.8 は、米国 Sun Microsystems™, Inc. が開発した新しいオペレーティングリリースです。SunOS の別のバージョンでは、ユーティリティとライブラリ機能の動作がこのマニュアルの説明と異なる場合があります。

対象読者

このマニュアルは、単一コンピュータ用の既存アプリケーションをネットワーク化された分散型アプリケーションとして変換する、分散型アプリケーションを設計する、分散型アプリケーションを実装する、または Sun OS 5.8 オペレーティングシステムプラットフォーム上の分散型アプリケーションを管理する、といった作業に携わるプログラマを対象としています。ネットワーク化されたアプリケーションの詳しい手法は、『*ONC+ 開発ガイド*』で説明しています。このマニュアルは、読者にプログラミングの基礎知識があり、C プログラミング言語と UNIX オペレーティングシステムの作業に慣れていることを前提としています。ネットワークのプログラミング経験があると内容を理解しやすくなりますが、この経験はなくても差し支えありません。

内容の紹介

SunOS 5.8 プラットフォームのネットワークインタフェース部分の各サービスと機能は、次の各章で説明しています。

第 1 章では、このマニュアルの内容と目的について説明しています。

第 2 章では、トランスポート層のソケットインタフェースについて説明しています。

第 3 章では、UNIX™ System V の System Transport Level Interface について説明しています。

第 4 章では、ネットワークトランスポートとその構成を選択するためアプリケーションが使用するネットワーク選択メカニズムについて説明しています。

付録

付録 A では、UNIX ファミリソケットについて説明しています。

付録 B では、このマニュアルで例として示されているすべてのコードの機能一覧を記載しています。これらのモジュールは、付録の初めで示されている規則によって例として提供されます。

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、`http://www1.fatbrain.com/documentation/sun` の Sun Documentation Center をご覧ください。

Sun のオンラインマニュアル

http://docs.sun.com では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索をおこなうこともできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。

表 P–1 表記上の規則 続く

字体または記号	意味	例
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define \ XV_VERSION_STRING'**

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が、適宜併記されています。
- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

ネットワークインタフェースとは

このマニュアルは、SunOS 5.8 オペレーティングシステムのネットワークサービスを使用するためのプログラマ用インタフェースについて説明しています。

SunOS 5.8 は System V の Release 4 (SVR4) と完全な互換性があり、System V Interface Description (SVID) の第 3 版に準拠しています。SunOS 5.8 は、System V のすべてのネットワークサービスをサポートします。

SunOS 5.8 におけるネットワーキング

SunOS 5.8 ネットワーキングのテーマは、トランスポートの独立性です。ネットワーク化されたアプリケーションは、特定のトランスポートプロトコル用にカスタマイズしなくても実行できます。

このシステムの以前のバージョンにも、ソケット、Transport Layer Interface (TLI)、および名前をアドレスに変換する機能が付いています。SunOS 5.8 では、これらの機能が強化され、新しいネットワーク選択機能と連携して動作します。そのため、ユーザーアプリケーションで特定のプロトコルやアドレス書式の詳細を指定する必要がありません。

トランスポートに依存しない RPC (遠隔手続き呼び出し) は、 アプリケーションを配下のトランスポートから解放するインタフェースや、トランスポートとの結合を強化するインタフェースを提供します。もっとも適したレベルの選択はプログラマに任されます。

オプションを調整したり特定のアドレスを使用しなければならないアプリケーションについては、従来と同様です。ただし現在は、プロトコルスタックが異なっても移植を簡単に行えるようにアプリケーションを作成できます。

SunOS 5.8 のもう 1 つの重要な機能は、トランスポートレベルとリンクレベルにおける標準化された内部カーネルネットワークインタフェースです。トランスポートレベルでは、AT&T Transport Provider Interface (TPI) が必要です。リンクレベルでは、UNIX International Data Link Provider Interface (DLPI) が必要です。

これらのインタフェースの標準化により、トランスポートレベルとリンクレベルの STREAMS ドライバを、それらと通信するモジュールまたはドライバを変更することなく交換できます。特に、TLI とソケットは TPI をサポートする任意のトランスポートプロバイダとインタフェースをとることができ、DLPI をサポートするデバイスドライバはすべてインターネットプロトコル (IP) 下でリンクできます。

OSI (開放型システム間相互接続) 参照モデル

OSI (Open Systems Interconnect) 参照モデルは、商用ネットワークサービスアーキテクチャの基本です。単独に開発されたほかのネットワークプロトコルは、このモデルにおおよそ準拠しています。例として、TCP/IP インターネットプロトコル群が挙げられます。

OSI 参照モデルは、ネットワーキングの概念に便利なフレームワークです。基本的に、データは送信側によってネットワークに投入されます。データは通信接続を介して伝送され、受信側に配信されます。このためには、多様なネットワークハードウェアとネットワークソフトウェアが連携して動作しなければなりません。

OSI 参照モデルでは、図 1–1 に示すようにネットワーキング機能を 7 つの層に分割します。

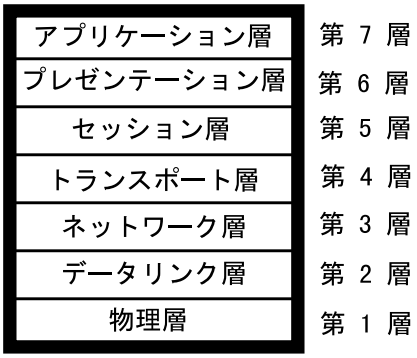

図 1–1　OSI 参照モデル

各プロトコル層は、その層の上位の層に対してサービスを行います。プロトコル層の ISO 定義は、設計者に実装における多少の自由を許します。たとえば、アプリケーションの中には、プレゼンテーション層とセッション層をスキップしてトランスポート層と直接インタフェースをとるものがあります。

OSI 層の定義

第 *1* 層*:* 物理層

モデルのハードウェア層。SPARC™ システムでは、この層はネットワーク伝送媒体へのコネクタ、マルチプレクサ、およびケーブルから構成されます。

第 *2* 層*:* データリンク層

送受信を行います。送信側では、Ethernet[1] (または類似) ソフトウェアが適切なサイズのパケットとしてデータをまとめ、それらをパッケージ化します。このパッケージ化には、意図する受信側の物理アドレスも含まれます。この層は、メッセージパケットの伝送も行い、必要に応じて再伝送します。

受信側では、Ethernet ハードウェアがアドレスによってパケットを認識して受信します。Ethernet ソフトウェアが、伝送パッケージをストライプ化し、データを再アセンブルします。Ethernet ソフトウェアは、伝送エラーを検出できます。

1. Ethernet は Xerox 社の商標です。

第 *3* 層*:* ネットワーク層

論理アドレスから物理アドレスへの変換など、メッセージルーティングを行います。SPARC システムで一般に使用されるネットワーク層は、インターネットプロトコル (IP) です。

第 *4* 層*:* トランスポート層

ネットワーク上のデータフローを制御します。SunOS 5.8 では、トランスポート層インタフェース (Transport Layer Interface、TLI)、伝送制御プロトコル (Transmission Control Protocol、TCP)、またはユーザーデータグラムプロトコル (User Datagram Protocol、UDP) のどれでも使用できます。SPARC システムでは、コネクションモードサービスは一般に TCP を介して提供され、コネクションレスサービスは一般に UDP を介して提供されます。

第 *5* 層*:* セッション層

プロセス間の高信頼セッションを管理します。遠隔手続き呼び出し (Remote Procedure Call、RPC) はこの層に属します。この層のインタフェースは、関数呼び出しの意味論を使用する遠隔通信を許可します。

第 *6* 層*:* プレゼンテーション層

コンピュータ独自のデータ表現と、ネットワークを介して送信されるプロセッサに依存しない形式間の変換を行います。SunOS 5.8 環境では、プロセッサに依存しないデータ形式は XDR です。

第 *7* 層*:* アプリケーション層

この最上位の層には、ユーザーレベルのプログラムとサービスが存在します。ユーザーレベルのプログラムには、`telnet`、`rlogin`、`ftp`、`yppasswd` などがあります。ユーザーレベルのサービスには、NFS™、NIS™、DNS などがあります。

参照モデルの各層は、業界標準がすでに定義されているか、あるいは現在その準備が進められています。各層には、その層が提供するサービスに対するインタフェースを指定する標準と、その層内のサービスが監視するプロトコルを指定する標準が定義されています。サービスインタフェース標準のユーザーは、プロトコル、およびその層のその他の実装詳細の影響は受けません。

トランスポート層

トランスポート層はアプリケーションと上位の層の間でエンドツーエンドのサービスを提供するモデルの最下位の層です。この層は、配下のネットワークのトポロジと特性をユーザーには見えないようにします。トランスポート層はまた、同時に存在する多くのプロトコル群 (ISO プロトコル、TCP および TCP/IP インターネットプロトコル群、Xerox Network Systems (XNS)、システムネットワークアーキテクチャ (System Network Architecture、SNA) など) に共通の一連のサービスを定義します。

RPC プログラミングでは、「ネットワーク」という用語はしばしばトランスポートまたはトランスポートタイプの類義語として使用されます。

トランスポート層インタフェース (TLI)

トランスポート層インタフェース (Transport Layer Interface、TLI) は、業界標準の Transport Service Definition (ISO 8072) でモデル化されています。TLI は、TCP と UDP の両方にアクセスするために使用できます。TLI は、STREAMS I/O メカニズムを使用するユーザーライブラリとして実装されます。

第 2 章

ソケットインタフェース

この章では、ソケットインタフェースについて、プログラム例を示して具体的に説明します。

ソケットはマルチスレッドに対して安全

この章で説明するインタフェースは、マルチスレッドに対して安全です。ソケット関数の呼び出しを含むアプリケーションは、マルチスレッド対応のアプリケーションで自由に使用できます。しかし、アプリケーションに有効な多重度は指定されていません。

SunOS 4 のバイナリ互換性

SunOS 4 以降の主な変更は、SunOS 5 リリースにも継承しています。パッケージにバイナリ互換性があるため、動的にリンクされた SunOS 4 ベースのソケットアプリケーションは SunOS 5 でも実行できます。

1. コンパイル行で、ソケットライブラリ (`-lsocket` または `libsocket`) を明示的に指定する必要があります。
2. 場合によっては `libnsl` もリンクする必要があります (`-lnsl -lsocket` ではなく `-lsocket -lnsl` と指定する)。
3. SunOS 5 で実行するには、ソケットライブラリを使用して SunOS 4 のソケットベースアプリケーションをすべてコンパイルし直す必要があります。

ソケットの概要

ソケットは、ネットワークプロトコルに対してもっとも一般的に使用される低レベルインタフェースです。ソケットは、1981 年以来 SunOS リリースに不可欠な部分となっています。ソケットは通信の終端であり、名前をバインドできます。ソケットにはタイプがあり、関連プロセスが 1 つ存在します。ソケットは、次のようなプロセス間通信のためのクライアントサーバーモデルを実装するために設計されました。

- ネットワークプロトコルのインタフェースが、TCP/IP、Xerox インターネットプロトコル (XNS)、UNIX ファミリのような複数の通信プロトコルを提供する必要がある
- ネットワークプロトコルのインタフェースが、接続を待機するサーバーコードと接続を開始するクライアントコードを提供する必要がある
- ネットワークプロトコルのインタフェースが、通信がコネクション型であるかコネクションレス型であるかに基づいて動作する必要がある
- アプリケーションプログラムが、`open(2)` 呼び出しを使用してアドレスをバインドするのではなく、配信するデータグラムの宛先アドレスを指定する必要がある

ソケットは、UNIX ファイルのように動作し、ネットワークプロトコルが使用できるように処置します。アプリケーションは、必要に応じてソケットを作成します。ソケットは、`close(2)`、`read(2)`、`write(2)`、`ioctl(2)`、および `fcntl(2)` インタフェースと連携して動作します。オペレーティングシステムは、ファイルのファイル記述子とソケットのファイル記述子を区別します。

ソケットライブラリ

ソケットインタフェースルーチンは、アプリケーションとリンクが必要なライブラリ内に存在します。ライブラリ `libsocket.so` は、ほかのシステムサービスライブ

ラリと共に /usr/lib に入っています。libsocket.so は、動的リンクに使用されます。

ソケットタイプ

ソケットタイプは、ユーザーにもわかる通信プロパティを定義します。インターネットファミリソケットは、TCP/IP トランスポートプロトコルへのアクセスを提供します。インターネットファミリは、IPv6 と IPv4 の両方で通信できるソケットの場合、AF_INET6 という値で識別されます。また、古いアプリケーションとのソース互換、および IPv4 に対する raw (生の) アクセスを目的とした値 AF_INET もサポートされています。

サポートされる 3 つのタイプのソケットを次に示します。

1. ストリームソケットは、プロセスが TCP を使用して通信を行えるようにします。ストリームソケットは、信頼性の高い、順序付けされた、重複のない双方向データフローをレコード境界なしで提供します。接続が確立されたあと、これらのソケットからのデータの読み取り、およびこれらのソケットに対するデータの書き込みがバイトストリームとして行えます。ソケットタイプは SOCK_STREAM です。
2. データグラムソケットは、プロセスが UDP を使用して通信を行えるようにします。データグラムソケットは、メッセージの双方向フローをサポートします。データグラムソケット側のプロセスは、送信シーケンスから順序を変えてメッセージを受信でき、重複したメッセージを受信できます。データ内のレコード境界は保持されます。ソケットタイプは SOCK_DGRAM です。
3. raw ソケットは、ICMP へのアクセスを提供します。このタイプのソケットは、通常、データグラム型ですが、実際の特性はプロトコルが提供するインタフェースに依存します。raw ソケットは、ほとんどのアプリケーションには使用されません。このタイプは、新しい通信プロトコルの開発をサポートしたり、既存プロトコルの難解な機能にアクセスしたりするために提供されています。raw ソケットを使用できるのは、スーパーユーザープロセスだけです。ソケットタイプは SOCK_RAW です。

詳細については、60ページの「特定のプロトコルの選択」を参照してください。

インタフェースセット

SunOS 5.8 には、2 セットのソケットインタフェース、BSD ソケットインタフェースと XNS 5 (Unix98) ソケットインタフェースが付属しています (XNS 5 ソケットは SunOS 5.7 以降)。XNS 5 インタフェースは、BSD インタフェースとわずかに異なります。

XNS 5 ソケットインタフェースについては、次のマニュアルページで説明されています。`accept(3XNET)`、`bind(3XNET)`、`connect(3XNET)`、`endhostent(3XNET)`、`endnetent(3XNET)`、`endprotoent(3XNET)`、`endservent(3XNET)`、`gethostbyaddr(3XNET)`、`gethostbyname(3XNET)`、`gethostent(3XNET)`、`gethostname(3XNET)`、`getnetbyaddr(3XNET)`、`getnetbyname(3XNET)`、`getnetent(3XNET)`、`getpeername(3XNET)`、`getprotobyname(3XNET)`、`getprotobynumber(3XNET)`、`getprotoent(3XNET)`、`getservbyname(3XNET)`、`getservbyport(3XNET)`、`getservent(3XNET)`、`getsockname(3XNET)`、`getsockopt(3XNET)`、`htonl(3XNET)`、`htons(3XNET)`、`inet_addr(3XNET)`、`inet_lnaof(3XNET)`、`inet_makeaddr(3XNET)`、`inet_netof(3XNET)`、`inet_network(3XNET)`、`inet_ntoa(3XNET)`、`listen(3XNET)`、`ntohl(3XNET)`、`ntohs(3XNET)`、`recv(3XNET)`、`recvfrom(3XNET)`、`recvmsg(3XNET)`、`send(3XNET)`、`sendmsg(3XNET)`、`sendto(3XNET)`、`sethostent(3XNET)`、`setnetent(3XNET)`、`setprotoent(3XNET)`、`setservent(3XNET)`、`setsockopt(3XNET)`、`shutdown(3XNET)`、`socket(3XNET)`、および `socketpair(3XNET)`。

従来の SunOS 5 BSD ソケットの動作については、対応する 3N のマニュアルページに説明されています。次のセクション 3N マニュアルページには、多くの新しいインタフェースが追加されました。`freeaddrinfo(3SOCKET)`、`freehostent(3SOCKET)`、`getaddrinfo(3SOCKET)`、`getipnodebyaddr(3SOCKET)`、`getipnodebyname(3SOCKET)`、`getnameinfo(3SOCKET)`、`inet_ntop(3SOCKET)`、および `inet_pton(3SOCKET)`。XNS 5 (Unix98) ソケットインタフェースを使用するアプリケーションの構築については、`standards(5)` のマニュアルページを参照してください。

ソケットの基本的な使用

この節では、基本的なソケットインタフェースの使用について説明します。

ソケットの作成

socket(3SOCKET) 呼び出しは、指定されたファミリに、指定されたタイプのソケットを作成します。

```
s = socket(family, type, protocol);
```

プロトコルが指定されないと (値が 0)、システムは要求されたソケットタイプをサポートするプロトコルを選択します。ソケットハンドル (ファイル記述子) が返されます。

ファミリは、sys/socket.h で定義されている定数の 1 つで指定されます。AF_*suite* という名前の定数は、表 2–1 に示すように、名前を解釈する場合に使用するアドレス書式を指定します。

表 2–1 プロトコルファミリ

AF_APPLETALK	Apple Computer, Inc. の Appletalk ネットワーク
AF_INET6	IPv6 と IPv4 用のインターネットファミリ
AF_INET	IPv4 専用のインターネットファミリ
AF_PUP	Xerox Corporation の PUP インターネット
AF_UNIX	Unix ファイルシステム

ソケットタイプは、sys/socket.h で定義されています。これらのタイプ (SOCK_STREAM、SOCK_DGRAM、または SOCK_RAW) は、AF_INET6、AF_INET、および AF_UNIX でサポートされます。次の文は、インターネットファミリでストリームソケットを作成します。

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

この呼び出しの結果、基本的な通信を提供する TCP プロトコルを使用したストリームソケットが作成されます。通常は、デフォルトプロトコル (引数 *protocol* が 0) を使用してください。55ページの「拡張機能」で説明しているように、デフォルト以外のプロトコルも指定できます。

ローカル名のバインド

ソケットは、名前のない状態で作成されます。ソケットにアドレスがバインドされるまでは、遠隔プロセスはソケットを参照できません。通信プロセスは、アドレスを介して接続されます。インターネットファミリでは、接続はローカルアドレス、リモートアドレス、ローカルポート、およびリモートポートから構成されます。protocol、local address、local port、foreign address、foreign port のような重複した順序セットは指定できません。

bind(3SOCKET) 呼び出しを使用すると、プロセスはソケットのローカルアドレスを指定できます。これは、local address、local port というセットになります。connect(3SOCKET) と accept(3SOCKET) は、アドレスの対のリモート側を決定することにより、ソケットの関連付けを完了します。bind(3SOCKET) 呼び出しは次のように使用します。

```
bind (s, name, namelen);
```

ソケットハンドルは *s* です。バインドされる名前は、サポートするプロトコルによって解釈されるバイト列です。インターネットファミリ名には、インターネットアドレスとポート番号が含まれます。

次の例は、インターネットアドレスのバインドを示しています。

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in6 sin6;
...
  s = socket(AF_INET6, SOCK_STREAM, 0);
  bzero (&sin6, sizeof (sin6));
  sin6.sin6_family = AF_INET6;
  sin6.sin6_addr.s6_addr = in6addr_arg;
  sin6.sin6_port = htons(MYPORT);
  bind(s, (struct sockaddr *) &sin6, sizeof sin6);
```

アドレス sin6 の内容は、インターネットアドレスのバインドについて説明した 61ページの「アドレスのバインド」で説明しています。

接続の確立

接続の確立は、通常、クライアントの役割を果たすプロセスと、サーバーの役割を果たすプロセスによって非同期で行われます。サーバーは、関連付けられた既知のアドレスと、接続要求のためのソケット上のブロックにソケットをバインドします。すると、関連のないプロセスがサーバーに接続できるようになります。クライアントは、サーバーがソケットに対する接続を開始すると、サーバーからサービスを要求します。クライアント側では、connect(3SOCKET) 呼び出しで接続を開始します。インターネットファミリの場合の例を次に示します。

```
struct sockaddr_in6 server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

接続呼び出しの時点でクライアントのソケットがバインドされていない場合、システムは自動的に名前を選択し、ソケットにバインドします。61ページの「アドレスのバインド」を参照してください。これは、ローカルアドレスがクライアント側のソケットにバインドされる一般的な方法です。

クライアントの接続を受信するには、サーバーはそのソケットをバインドしたあとに 2 つの処理を行う必要があります。まず、待ち行列に入れることができる接続要求の数を示します。続いて接続を受け入れます。

```
struct sockaddr_in6 from;
 ...
listen(s, 5);                   /* 5 つの接続待ち行列を許可する */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

ソケットハンドル *s* は、接続要求の送信先であるアドレスにバインドされるソケットです。listen(3SOCKET) の 2 つ目のパラメータは、待ち行列に入れることができる未処理の接続の最大数を指定します。from は、クライアントのアドレスを指定する構造体です。場合によって NULL ポインタが渡されます。*fromlen* は構造体の長さです (UNIX ファミリでは、from は struct sockaddr_un として宣言される)。

accept(3SOCKET) は一般にブロックします。accept(3SOCKET) は、要求側クライアントに接続されている新しいソケット記述子を返します。*fromlen* の値は、アドレスの実際のサイズに変更されます。

サーバーは、特定のアドレスからのみ接続を受け入れます。しかしこのことを示すことができません。サーバーは、accept(3SOCKET) が返す from アドレスをチェックし、受け入れ不可能なクライアントとの接続を閉じることができます。

サーバーは、複数のソケット上の接続を受け入れることも、あるいは accept 呼び出しのブロックを避けることもできます。これらの手法については、55ページの「拡張機能」で説明しています。

接続エラー

接続が失敗する場合はエラーが返されます (しかし、システムによってバインドされたアドレスは残る)。成功すると、ソケットがサーバーと関連付けられ、データ転送を開始できます。

表 2–2 は、接続が失敗する場合に返される一般的なエラーをいくつか示しています。

表 2–2 ソケット接続エラー

ソケットエラー	エラーの説明
ENOBUFS	呼び出しをサポートするためのメモリーが足りない
EPROTONOSUPPORT	不明なプロトコルの要求
EPROTOTYPE	サポートされないソケットタイプの要求
ETIMEDOUT	指定された時刻に接続が確立されていない。これは、宛先ホストがダウンしているか、あるいはネットワーク内の障害で伝送が中断される場合に発生する
ECONNREFUSED	ホストがサービスを拒否した。これは、要求されたアドレスにサーバープロセスが存在しない場合に発生する
ENETDOWN または EHOSTDOWN	これは、基本通信インタフェースが配信するステータス情報によって起こる
ENETUNREACH または EHOSTUNREACH	このオペレーションエラーは、ネットワークまたはホストに対する経路がないため、あるいは中間ゲートウェイまたは切り替えノードが返すステータス情報が原因で起こる。返されるステータスが十分でないために、ダウンしているネットワークとダウンしているホストが区別できない場合もある

データ転送

この節では、データの送信と受信のための関数について説明します。メッセージの送受信は、次のように通常の read(2) インタフェースと write(2) インタフェースを使用して行います。

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

また、次のように send(3SOCKET) 呼び出しと recv(3SOCKET) 呼び出しも使用できます。

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

send(3SOCKET) と recv(3SOCKET) は read(2) と write(2) に非常によく似ていますが、flags 引数が重要です。必要に応じて、sys/socket.h で定義されている以下のフラグをゼロ以外の値として指定できます。

MSG_OOB	帯域外データを送受信する
MSG_PEEK	読み取らずにデータの確認だけを行う
MSG_DONTROUTE	パケットの経路を指定せずにデータを送信する

帯域外データは、ストリームソケットに固有のものです。recv(3SOCKET) 呼び出しで MSG_PEEK が指定される場合、存在するすべてのデータがユーザーに返されますが、データは読み取られていないものとして扱われます。ソケット上の次の read(2) または recv(3SOCKET) 呼び出しは、同じデータを返します。発信パケットに適用されるパケット経路を指定しないデータ送信オプションは、現在、ルーティングテーブルの管理プロセスにだけ使用されており、一般にはほとんど使用されません。

ソケットを閉じる

SOCK_STREAM ソケットは、close(2) 関数呼び出しで破棄できます。close(2) のあとで信頼できる配信が見込まれるソケットにデータが待ち行列化されている場合、プロトコルは継続してそのデータの転送を試みます。期限が来てもデータが配信されない場合は、データが破棄されます。

shutdown(3SOCKET) は、SOCK_STREAM ソケットを正常に閉じます。両方のプロセスで送信が行われなくなっていることを認識できます。この呼び出しの書式は次のとおりです。

```
shutdown(s, how);
```

how は次のように定義されています。

0　それ以上の受信を許可しない

1　それ以上の送信を許可しない

2　それ以上の受信と送信を許可しない

ストリームソケットの接続

図 2–1 と次の 2 つの例は、インターネットファミリのストリーム接続の開始と受け入れを示しています。

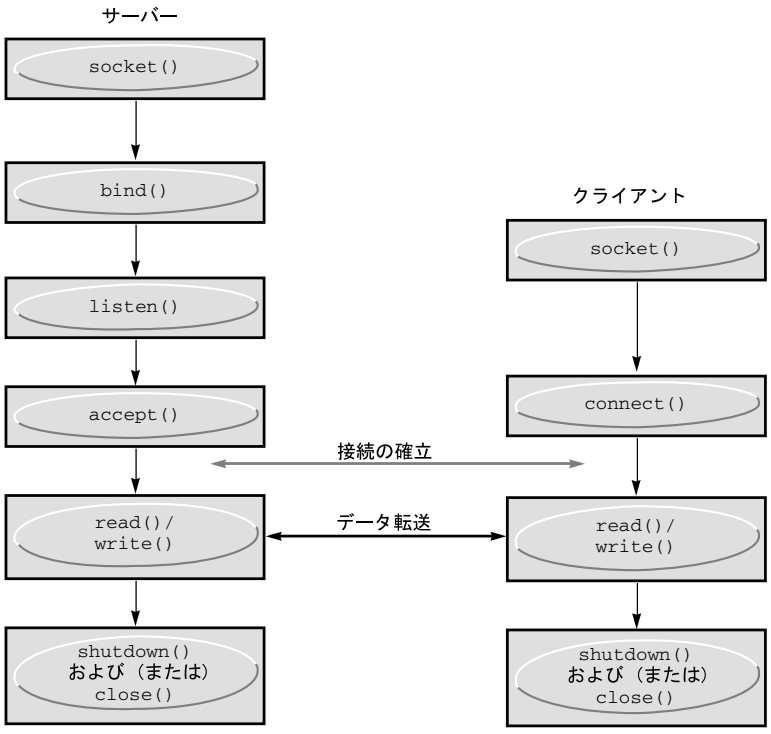

図 2–1　ストリームソケットを使用したコネクション型の通信

例 2–1 はサーバー側のプログラムです。このプログラムは、ソケットを作成してそのソケットに名前をバインドし、続いてポート番号を表示します。このプログラムは `listen(3SOCKET)` を呼び出して、ソケットが接続要求を受け入れる用意ができていることをマークし、要求の待ち行列を初期化します。プログラムの残り部分は無限ループです。ループの各パスは、新しいソケットを作成することによって新しい接続を受け入れ、待ち行列からその接続を削除します。サーバーは、ソケットからのメッセージを読み取って表示し、メッセージを閉じます。`in6addr_any` の使用については、61ページの「アドレスのバインド」で説明しています。

例 2–1 インターネットストリーム接続の受け入れ (サーバー)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
* このプログラムは、ソケットを作成したあと無限ループを開始します。
* ループごとに接続を受け入れ、その接続からのデータを出力します。
* 接続が遮断するか、あるいはクライアントが接続を閉じる時点で
* プログラムは新しい接続を受け入れます。
*/

main()
{
   int sock, length;
   struct sockaddr_in6 server;
   int msgsock;
   char buf[1024];
   int rval;

   /* ソケットを作成する。*/
   sock = socket(AF_INET6, SOCK_STREAM, 0);
   if (sock == -1) {
      perror("opening stream socket");
      exit(1);
   }
   /* ワイルドカードを使用してソケットをバインドする。*/
   bzero (&server, sizeof(server));
   bzero (&sin6, sizeof (sin6));
   server.sin6_family = AF_INET6;
   server.sin6_addr.s6_addr = in6addr_any;
   server.sin6_port = 0;
   if (bind(sock, (struct sockaddr *) &server, sizeof server)
         == -1) {
      perror("binding stream socket");
      exit(1);
   }
   /* 割り当てられたポート番号を調べ、それを出力する。*/
   length = sizeof server;
   if (getsockname(sock,(struct sockaddr *) &server,&length)
         == -1) {
```

```
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin6_port));
    /* 接続の受け入れを開始する。*/
    listen(sock, 5);
    do {
        msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
        if (msgsock == -1
            perror("accept");
        else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock,buf, 1024)) == -1)
                perror("reading stream message");
            if (rval == 0)
                printf("Ending connection\n");
            else
                /* データが出力可能であると想定する */
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    } while(TRUE);
    /*
     * このプログラムには無限ループが含まれるため、ソケット "sock" は
     * 明示的に閉じられることはありません。しかし、プロセスが中断されるか
     * 正常に終了する場合は自動的に閉じます。
     */
    exit(0);
}
```

接続を開始するため、例 2–2 のクライアント側プログラムは、ストリームソケットを作成し、接続用のソケットのアドレスを指定して connect(3SOCKET) を呼び出します。宛先ソケットが存在し、要求が受け入れられる場合は、接続が完了し、プログラムはデータを送信できます。データは、メッセージ境界なしで順番に配信されます。接続は、一方のソケットが閉じられた時点で遮断されます。このプログラム内のデータ表現ルーチン (ntohl(3SOCKET)、ntohs(3SOCKET)、htons(3SOCKET)、htonl(3XNET) など) の詳細については、byteorder(3SOCKET) のマニュアルページを参照してください。

例 2–2 インターネットファミリのストリーム接続 (クライアント)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * このプログラムはソケットを作成し、コマンド行で指定
 * されるソケットを使用して接続を開始します。この接続で
 * データがいくらか送信されたあとソケットが閉じられ、
 * 接続が終了します。
```

```
 * コマンド行の書式: streamwrite hostname portnumber
 * 使用法: pgm host port
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock, errnum h_addr_index;
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];

    /* ソケットを作成する。*/
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* コマンド行で指定される名前を使用してソケットを接続する。*/
    bzero (&sin6, sizeof (sin6));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(AF_INET6, argv[1], AI_DEFAULT, &errnum);
/*
 * getinodebyname が、指定されたホストのネットワーク
 * アドレスを含む構造体を返す。
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }

    h_addr_index = 0;
    while (hp->h_addr_list[h_addr_index] != NULL) {
        bcopy(hp->h_addr_list[h_addr_index], &server.sin6_addr,
                    hp->h_length);
        server.sin6_port = htons(atoi(argv[2]));
        if (connect(sock, (struct sockaddr *) &server,
                    sizeof (server)) == -1) {
            if (hp->h_addr_list[++h_addr_index] != NULL) {
                /* 次のアドレスを試みる */
                continue;
            }
            perror("connecting stream socket");
            freehostent(hp);
            exit(1);
        }
        break;
    }
    freehostent(hp);
    if (write( sock, DATA, sizeof DATA) == -1)
        perror("writing on stream socket");
    close(sock);
    freehostent (hp);
    exit(0);
}
```

データグラムソケット

データグラムソケットは、同期データ交換インタフェースを提供します。接続を確立するための必要条件はありません。各メッセージには、宛先アドレスが含まれます。図 2–2 は、サーバーとクライアント間の通信の流れを示しています。

注 - 次の図のサーバー側で示されている `bind(3SOCKET)` の手順は省略できます。

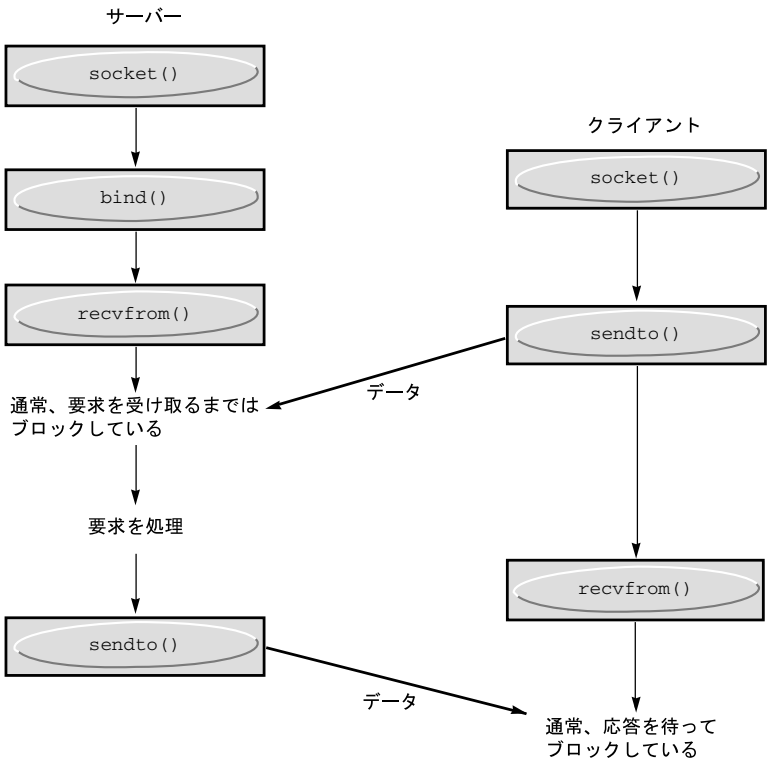

図 2–2　データグラムソケットを使用したコネクションレス型の通信

データグラムソケットは、29ページの「ソケットの作成」で説明しているように作成されます。特定のローカルアドレスが必要な場合、`bind(3SOCKET)` オペレーションが最初のデータ伝送よりも先行しなければなりません。それ以外の場合、データが最初に送信される際にシステムがローカルアドレスまたはポート (あるいはこの両方) を設定します。データの送信には、`sendto(3SOCKET)` を使用します。

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

s、*buf*、*buflen*、および *flags* パラメータは、コネクション型のソケットの場合と同じです。*to* と *tolen* の値は、意図するメッセージ受信者のアドレスを示します。ローカルにエラー条件 (到達できないネットワークなど) が検出されると、–1 が戻り、*errno* がエラー番号に設定されます。

データグラムソケット上のメッセージを受信するには、recvfrom(3SOCKET) を使用します。

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

呼び出しの前に、*fromlen* が *from* バッファーのサイズに設定されます。*fromlen* は、戻り時には、データグラムの配信元であるアドレスのサイズに設定されます。

データグラムソケットは、ソケットを特定の宛先アドレスと関連付けるために connect(3SOCKET) 呼び出しを使用することもできます。続いて、send(3SOCKET) 呼び出しを使用できます。宛先アドレスの明示的な指定がない状態でソケット上で送信されるデータはすべて、接続されたピアにアドレス指定され、そのピアから受信されるデータだけが配信されます。1 つのソケットで一度に許可されるのは、接続された 1 つのアドレスだけです。2 つ目の connect(3SOCKET) 呼び出しは、宛先アドレスを変更します。データグラムソケット上の接続要求は、ただちに返されます。システムは、ピアのアドレスを記録します。accept(3SOCKET) と listen(3SOCKET) は、データグラムソケットでは使用されません。

データグラムソケットが接続されている間、前の send(3SOCKET) 呼び出しからのエラーは非同期に返すことができます。これらのエラーは、そのソケットの後続のオペレーションで報告できます。また、getsockopt(3SOCKET) のオプションである SO_ERROR を使用してそのエラーステータスを問い合わせることもできます。

例 2–3 は、ソケットの作成、ソケットへの名前のバインド、およびソケットへのメッセージ送信によってインターネット呼び出しを送信する方法を示しています。

例 2–3 インターネットファミリデータグラムの送信

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm, the tide is full . . ."

/*
```

```
 * ここで、コマンド行引数から得られる名前を持つ受信者に
 * データグラムを送信する。
 * コマンド行の書式: dgramsend hostname portnumber
 */
main(argc, argv)
   int argc, errnum;
   char *argv[];
{
   int sock;
   struct sockaddr_in6 name;
   struct hostent *hp;

   /* 送信を行うソケットを作成する。*/
   sock = socket(AF_INET6,SOCK_DGRAM, 0);
   if (sock == -1) {
      perror("opening datagram socket");
      exit(1);
   }
   /*
    * 「送信」先ソケットの、ワイルドカードを使用しない構造名。
    * getinodebyname は、指定されたホストのネットワークアドレスを
    * 含む構造体を返します。ポート番号は、コマンド行から取得され
    * ます。
    */
   hp = getipnodebyname(AF_INET6, argv[1], AI_DEFAULT, &errnum);
   if (hp == (struct hostent *) 0) {
      fprintf(stderr, "%s: unknown host\n", argv[1]);
      exit(2);
   }
   bzero (&sin6, sizeof (sin6));
   bzero (&name, sizeof (name));
   memcpy((char *) &name.sin6_addr, (char *) hp->h_addr,
      hp->h_length);
   name.sin6_family = AF_INET6;
   name.sin6_port = htons(atoi(argv[2]));
   /* メッセージを送信する。*/
   if (sendto(sock,DATA, sizeof DATA ,0,
      (struct sockaddr *) &name,sizeof name) == -1)
      perror("sending datagram message");
   close(sock);
   exit(0);
}
```

例 2–4 は、ソケットの作成、ソケットへの名前のバインド、およびソケットからの読み取りによってインターネット呼び出しを読み取る方法を示しています。

例 2–4 インターネットファミリデータグラムの読み取り

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * このプログラムは、データグラムソケットを作成し、そのソケッ
 * トに名前をバインドし、続いてそのソケットから読み取ります。
 */
```

```
main()
{
   int sock, length;
   struct sockaddr_in6 name;
   char buf[1024];

   /* 読み取られるソケットを作成する。*/
   sock = socket(AF_INET6, SOCK_DGRAM, 0);
   if (sock == -1) {
      perror("opening datagram socket");
      exit(1);
   }
   /* ワイルドカードを使用して名前を作成する。*/
   bzero (&sin6, sizeof (sin6));
   name.sin6_family = AF_INET6;
   name.sin6_addr.s6_addr = in6addr_any;
   name.sin6_port = 0;
   if (bind(sock,(struct sockaddr *)&name, sizeof name) == -1) {
      perror("binding datagram socket");
      exit(1);
   }
   /* 割り当てられたポート値を確認し、それを出力する。*/
   length = sizeof(name);
   if (getsockname(sock,(struct sockaddr *) &name, &length)
         == -1)  {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(name.sin6_port));
   /* ソケットから読み取りを行う。*/
   if (read(sock, buf, 1024) == -1 )
      perror("receiving datagram packet");
   /* データが出力可能であると想定する。*/
   printf("-->%s\n", buf);
   close(sock);
   exit(0);
}
```

入出力の多重化

要求は、複数のソケットまたはファイルの間で多重化できます。このためには select(3C) を使用します。

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

select(3C) の最初の引数は、続く 3 つの引数によって示されるリスト内のファイル記述子の数です。

select(3C) の 2 つ目、3 つ目、4 つ目の引数は、3 つのファイル記述子セット (読み取りを行う記述子セット、書き込みを行うセット、および例外条件が認められるセット) を指します。帯域外データは、唯一の例外条件です。これらのポインタはどれも、適切にキャストされた NULL です。各セットは、ロング整数ビットマスクの配列を含む構造体です。配列のサイズは、FD_SETSIZE (select.h で定義されている) で設定します。配列には、各 FD_SETSIZE ファイル記述子のための 1 ビットを保持するだけの長さがあります。

マクロ FD_SET(*fd*, &*mask*) はセット mask 内のファイル記述子 *fd* を追加し、マクロ FD_CLR(*fd*, &*mask*) はこの記述子を削除します。セット mask は、使用前にゼロにする必要があります。マクロ FD_ZERO(&*mask*) は、セット mask を消去します。

select(3C) の 5 つ目の引数によって、タイムアウト値を指定できます。timeout ポインタが NULL の場合、select(3C) は、記述子が選択できるようになるか、あるいはシグナルが受信されるまでブロックします。timeout 内のフィールドが 0 に設定されると、select(3C) はただちにポーリングして戻ります。

select(3C) は、通常、選択されたファイル記述子の数を返します。select(3C) は、タイムアウトの期限が過ぎると 0 を返します。select(3C) は、ファイル記述子マスクが変更されず、エラーまたは割り込みが発生した場合、そこで指定されたエラー番号 *errno* に対し −1 を返します。成功した場合に返される 3 つのセットは読み取り可能なファイル記述子、書き込み可能なファイル記述子、または例外条件が保留されたファイル記述子を示します。

FD_ISSET(*fd*, &*mask*) マクロを使用して、選択マスク内のファイル記述子のステータスをテストしてください。セット mask 内に *fd* が存在する場合はゼロ以外の値が返され、存在しない場合は 0 が返されます。読み取りセットで FD_ISSET(*fd*, &*mask*) マクロの前に select(3C) を使用して、待ち行列に入っている、ソケット上の接続要求を確認します。

例 2–5 は、accept(3SOCKET) 呼び出しによって新しい接続をピックアップするタイミングを決定するために、読み取り用の「リスニング (待機) 」ソケットで select を使用する方法を示しています。このプログラムは、接続要求を受け入れ、データを読み取り、単一のソケットで切断します。

例 2–5 select(3C) を使用して保留状態の接続を確認する

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1
```

```
/*
 * このプログラムは、accept を呼び出す前に、select を使用
 * して、他のだれかが接続を試みていないかチェックします。
 */

main()
{
   int sock, length;
   struct sockaddr_in6 server;
   int msgsock;
   char buf[1024];
   int rval;
   fd_set ready;
   struct timeval to;

   /* ソケットを開き、そのソケットを先の例と同様にバインドする。*/

   /* 接続の受け入れを開始する。*/
   listen(sock, 5);
   do {
      FD_ZERO(&ready);
      FD_SET(sock, &ready);
      to.tv_sec = 5;
      to.tv_usec = 0;
      if (select(sock + 1, &ready, (fd_set *)0, (fd_set *)0, &to) == -1) {
         perror("select");
         continue;
      }
      if (FD_ISSET(sock, &ready)) {
         msgsock = accept(sock, (struct sockaddr *)0,
            (int *)0);
         if (msgsock == -1)
            perror("accept");
         else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock, buf, 1024)) == -1)
               perror("reading stream message");
            else if (rval == 0)
               printf("Ending connection\n");
            else
               printf("-->%s\n", buf);
         } while (rval > 0);
         close(msgsock);
      } else
         printf("Do something else\n");
   } while (TRUE);
    exit(0);
}
```

以前のバージョンの select(3C) ルーチンでは、引数は fd_sets のポインタではなく、整数のポインタでした。ファイル記述子の数が整数内のビット数よりも小さい場合は、現在でもこのような呼び出しを使用できます。

select(3C) は、同期多重スキーマを提供します。SIGIO と SIGURG シグナル (55 ページの「拡張機能」を参照) は、出力の完了、入力の有効性、および例外条件の非同期通知を提供します。

標準のルーチン

ネットワークアドレスを見つけ、構成しなければならない場合があります。この節では、ネットワークアドレスを操作するルーチンについて説明します。特に明記しない限り、この節に示す関数はインターネットファミリにだけ適用されます。

リモートホスト上のサービスを見つけるには、クライアントとサーバーが通信を行う前にさまざまなレベルの割り当てを行う必要があります。サービスには、人間が使用するための名前が付いています。サービス名とホスト名は、ネットワークアドレスに変換しなければなりません。最後に、そのアドレスを使用してホストを見つけ、ホストへの経路を定めます。割り当ての細部は、ネットワークアーキテクチャによって異なります。望ましいのは、ホストに名前が付いていることをネットワークが要求せず、ホストの物理位置の同一性を保護できることです。ホストのアドレスが指定されていると、ホストの位置をより柔軟に見つけることができます。

標準ルーチンは、ホスト名をネットワークアドレスに、ネットワーク名をネットワーク番号に、プロトコル名をプロトコル番号に、サービス名をポート番号に、適切なプロトコルをサーバープロセスとの通信における使用にそれぞれ割り当てます。標準ルーチンのどれかを使用する場合は、ファイル netdb.h を含めなければなりません。

ホスト名とサービス名

インタフェース getaddrinfo(3SOCKET)、getnameinfo(3SOCKET)、および freeaddrinfo(3SOCKET) を使用すると、ホスト上のサービスの名前とアドレスを簡単に変換できます。IPv6 の場合、getipnodebyname(3SOCKET) と getservbyname(3SOCKET) を呼び出してアドレスの結合方法を決定する代わりに、これらのインタフェースを使用できます。同様に IPv4 では、これらのインタフェースを gethostbyname(3NSL) と getservbyname(3SOCKET) の代わりに使用できます。IPv6 アドレスと IPv4 アドレスは、どちらも透過的に処理されます。

getaddrinfo(3SOCKET) は、指定されたホストの結合アドレスとポート番号、およびサービス名を返します。getaddrinfo(3SOCKET) が返す情報はすべて

動的に割り当てられるため、メモリーリークが防止されるように freeaddrinfo(3SOCKET) を使用して解放しなければなりません。getnameinfo(3SOCKET) は、指定されたアドレスとポート番号に関連付けられたホスト名とサービス名を返します。getaddrinfo(3SOCKET) と getnameinfo(3SOCKET) が返す EAI_xxx コードに基づくエラーメッセージを出力するには、gai_strerror(3SOCKET) を呼び出します。

getaddrinfo(3SOCKET) の使用例を次に示します。

```
struct addrinfo         *res, *aip;
struct addrinfo         hints;
int                     sock = -1;
int                     error;

/* ホストアドレスを取得する。アドレスのタイプは問わない。*/
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);
}
```

res が指す構造体の getaddrinfo(3SOCKET) が返す情報を処理したあと、次の文によって記憶領域を解放する必要があります。

```
freeaddrinfo(res);
```

次の例に示すように、getnameinfo(3SOCKET) はエラー原因を調べるのに特に便利です。

```
struct sockaddr_storage faddr;
int                     sock, new_sock;
socklen_t               faddrlen;
int                     error;
char                    hname[NI_MAXHOST];
char                    sname[NI_MAXSERV];

 ...
     faddrlen = sizeof (faddr);
     new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
     if (new_sock == -1) {
         if (errno != EINTR && errno != ECONNABORTED) {
             perror("accept");
         }
         continue;
     }
```

```
error = getnameinfo((struct sockaddr *)&faddr, faddrlen, hname,
            sizeof (hname), sname, sizeof (sname), 0);
if (error) {
    (void) fprintf(stderr, "getnameinfo: %s\n",
                gai_strerror(error));
} else {
    (void) printf("Connection from %s/%s\n", hname, sname);
}
```

hostent – ホスト名

インターネットホスト名からアドレスへの割り当ては、次のように hostent 構造体で示されます。

```
struct hostent {
    char  *h_name;            /* ホストの正式名称 */
    char  **h_aliases;        /* 別名リスト */
    int   h_addrtype;         /* ホストアドレスのタイプ (AF_INET6 など) */
    int   h_length;           /* アドレスの長さ */
    char  **h_addr_list;      /* NULL で終わるアドレスのリスト */
};
/* 最初のアドレス、ネットワークバイトオーダー */
#define h_addr h_addr_list[0]
```

getipnodebyname(3SOCKET) は、インターネットホスト名を hostent 構造体に割り当てます。getipnodebyaddr(3SOCKET) は、インターネットホストアドレスを hostent 構造体に割り当てます。freehostent(3SOCKET) は、hostent 構造体のメモリーを解放します。inet_ntop(3SOCKET) は、インターネットホストアドレスを表示可能な文字列に割り当てます。

このルーチンは、ホストの名前、その別名、アドレスタイプ (アドレスファミリ)、および NULL で終わる可変長アドレスのリストを含む hostent 構造体を返します。このアドレスリストが必要なのは、ホストが多くのアドレスを持つことができるためです。h_addr 定義は下位互換性のためであり、この定義は hostent 構造体のアドレスリストの最初のアドレスです。

netent – ネットワーク名

次に、ネットワーク名を番号に割り当て、netent 構造体を戻すルーチンを示します。

```
/*
 * ネットワーク番号が 32 ビットに収まると想定します。
 */
struct netent {
   char     *n_name;      /* ネットの正式名称 */
   char     **n_aliases;  /* 別名リスト */
   int      n_addrtype;   /* ネットアドレスのタイプ */
   int      n_net;        /* ネット番号、ホストバイトオーダー */
```

```
};
```

getnetbyname(3SOCKET)、getnetbyaddr_r(3SOCKET)、および getnetent(3SOCKET) は、上記のホストルーチンに対するネットワーク側のルーチンです。

protoent – プロトコル名

protoent 構造体は、getprotobyname(3SOCKET)、getprotobynumber(3SOCKET)、および getprotoent(3SOCKET) で使用されるプロトコル名マッピングを定義します。

```
struct protoent {
   char     *p_name;          /* プロトコルの正式名称 */
   char     **p_aliases       /* 別名リスト */
   int      p_proto;          /* プロトコル番号 */
};
```

servent – サービス名

インターネットファミリサービスは、特定の既知のポートに常駐し、特定のプロトコルを使用します。サービス名からポート番号へのマッピングは、servent 構造体で表現されます。

```
struct servent {
   char     *s_name;         /* サービスの正式名称 */
   char     **s_aliases;     /* 別名リスト */
   int      s_port;          /* ポート番号、ネットワークバイトオーダー */
   char     *s_proto;        /* 使用するプロトコル */
};
```

getservbyname(3SOCKET) は、サービス名と修飾プロトコルを servent 構造体に割り当てます (修飾プロトコルは省略可能)。次の呼び出しは、任意のプロトコルを使用する Telnet サーバーのサービス仕様を返します。

```
sp = getservbyname("telnet", (char *) 0);
```

次の呼び出しは、TCP プロトコルを使用する Telnet サーバーを返します。

```
sp = getservbyname("telnet", "tcp");
```

getservbyport(3SOCKET) と getservent(3SOCKET) も提供されます。getservbyport(3SOCKET) には、getservbyname(3SOCKET) のインタフェースに似たインタフェースがあります (ルックアップを修飾するためオプションのプロトコル名を指定可能)。

その他のルーチン

名前とアドレスの操作を簡易化するルーチンには、アドレスに関連するデータベースルーチンのほかにもいくつかあります。表 2–3 は、可変長のバイト列、およびバイトスワッピングのネットワークアドレスと値を要約したものです。

表 2–3 実行時ライブラリルーチン

呼び出し	機能説明
memcmp(3C)	バイト列を比較する。同じ場合は 0、異なる場合は 0 以外の値を返す
memcpy(3C)	*s2* の *n* バイトを *s1* にコピーする
memset(3C)	base の最初の *n* バイトの領域に値 value を割り当てる
htonl(3SOCKET)	ホストからネットワークバイトオーダーへの 32 ビット量の変換
htons(3SOCKET)	ホストからネットワークバイトオーダーへの 16 ビット量の変換
ntohl(3SOCKET)	ネットワークからホストバイトオーダーへの 32 ビット量の変換
ntohs(3SOCKET)	ネットワークからホストバイトオーダーへの 16 ビット量の変換

バイトスワッピングルーチンが提供されているのは、アドレスはネットワークオーダーで供給されるとオペレーティングシステムが考えるためです。一部のアーキテクチャでは、ホストバイトオーダーがネットワークバイトオーダーと異なるため、プログラムは必要に応じて値のバイトスワップを行わなければなりません。ネットワークアドレスを返すルーチンは、ネットワークオーダーで返します。バイト

スワッピング問題が発生するのは、ネットワークアドレスを解釈する場合だけです。たとえば、次のコードは TCP ポートまたは UDP ポートをフォーマットします。

```
printf("port number %d\n", ntohs(sp->s_port));
```

これらのルーチンを必要としないマシンでは、アドレスは NULL マクロとして定義されます。

クライアントサーバープログラム

もっとも一般的な分散型アプリケーションは、クライアントサーバーモデルです。この方式では、クライアントプロセスはサーバープロセスからのサービスを要求します。

代替方式として、休止しているサーバープロセスを削除できるサービスサーバーがあります。たとえば、インターネットサービスデーモン `inetd(1M)` は、構成ファイルを読み取ることにより、起動時に決定されるさまざまなポートで待機します。`inetd(1M)` のサービスを受けるポートで接続が要求されると、`inetd(1M)` はクライアントにサービスを行うために適切なサーバーを生成します。クライアントは、その接続で中間媒体が何らかの役割を果たしたことを認識しません。`inetd(1M)` については、72ページの「`inetd(1M)` デーモン」で詳しく説明しています。

サーバー

ほとんどのサーバーには、既知のインターネットポート番号または UNIX ファミリ名でアクセスします。例 2–6 は、リモートログインサーバーのメインループを示しています。

例 2–6 リモートログインサーバー

```
main(argc, argv)
   int argc;
   char *argv[];
{
   int f;
   struct sockaddr_in6 from;
   struct sockaddr_in6 sin;
   struct servent *sp;
```

```
   sp = getservbyname("login", "tcp");

   if (sp == (struct servent *) NULL) {
      fprintf(stderr, "rlogind: tcp/login: unknown service");
      exit(1);
   }
   ...
   #ifndef DEBUG
   /* サーバーを制御端末から分離する。*/
   ...
   #endif
    sin.sin6_port = sp->s_port; /* 限られたポート */
      sin.sin6_addr.s6_addr = in6addr_any;
      ...
      f = socket(AF_INET6, SOCK_STREAM, 0);
      ...
      if (bind( f, (struct sockaddr *) &sin, sizeof sin ) == -1) {
      ...
      }
      ...
      listen(f, 5);
      while (TRUE) {
         int g, len = sizeof from;
         g = accept(f, (struct sockaddr *) &from, &len);
         if (g == -1) {
            if (errno != EINTR)
               syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
         }
         if (fork() == 0) {
            close(f);
            doit(g, &from);
         }
         close(g);
      }
      exit(0);
}
```

例 2–7 は、サーバーがそのサービス定義を取得する方法を示しています。

例 2–7 リモートログインサーバー : 手順 1

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
  fprintf(stderr, "rlogind: tcp/login: unknown service\n");
  exit(1);
}
```

getservbyname(3SOCKET) の結果は、プログラムがサービス要求を待機するインターネットポートを定義するためにあとで使用されます。標準のポート番号の一部は、/usr/include/netinet/in.h に入っています。

例 2–8 は、オペレーションの非 DEBUG モードで、サーバーがその呼び出し側の制御端末との関連付けを解除する方法を示しています。

例 2–8　制御端末との関連付けを解除する

```
   (void) close(0);
   (void) close(1);
   (void) close(2);
   (void) open("/", O_RDONLY);
   (void) dup2(0, 1);
   (void) dup2(0, 2);
   setsid();
```

このコードは、サーバーが制御端末のプロセスグループからのシグナルを受信することを防ぎます。関連付けを解除したあと、サーバーはエラーレポートを端末に送信できないため、syslog(3C) を使用してエラーをログに記録する必要があります。

サーバーは次にソケットを作成し、サービス要求を待機します。bind(3SOCKET) が使用されると、サーバーは指定された位置で待機します (限られたポート番号で待機するリモートログインサーバーはスーパーユーザーとして動作する)。

例 2–9 は、ループの本体を示しています。

例 2–9　リモートログインサーバー : 本体

```
while(TRUE) {
  int g, len = sizeof(from);
  if (g = accept(f, (struct sockaddr *) &from, &len) == -1) {
   if (errno != EINTR)
    syslog(LOG_ERR, "rlogind: accept: %m");
   continue;
  }
  if (fork() == 0) {  /* 子 */
   close(f);
   doit(g, &from);
  }
  close(g);     /* 親 */
}
```

accept(3SOCKET) は、クライアントがサービスを要求するまでメッセージをブロックします。accept(3SOCKET) は、SIGCHLD のようなシグナルで割り込みを受ける場合、失敗を示すメッセージを返します。accept(3SOCKET) からの戻り値がチェックされ、エラーが発生している場合は syslog(3C) によってエラーが記録されます。

続いてサーバーは子プロセスの fork(2) を行い、リモートログインプロトコル処理の本体を呼び出します。接続要求を待ち行列に入れるために親プロセスが使用するソケットは、子プロセスで閉じられます。accept(3SOCKET) が作成するソケットは、親プロセスで閉じられます。クライアントのアドレスは、クライアントの認証を行うためにサーバーアプリケーションの doit() ルーチンに渡されます。このルーチンは、クライアントと共に実際のアプリケーションプロトコルを実行します。

クライアント

この節では、クライアントリモートログインプロセスで行われる処理について説明します。サーバー側と同様に、まずリモートログインのサービス定義の位置を確認します。

```
sp = getservbyname("login", "tcp");
if (sp == (struct servent *) NULL) {
  fprintf(stderr,"rlogin: tcp/login: unknown service");
  exit(1);
}
```

次に、getipnodebyname(3SOCKET) 呼び出しで宛先ホストを調べます。

```
hp = getipnodebyname (AF_INET6, argv[1], AI_DEFAULT, &errnum);
if (hp == (struct hostent *) NULL) {
  fprintf(stderr, "rlogin: %s: unknown host", argv[1]);
  exit(2);
}
```

次に、要求されたホストでサーバーに接続し、リモートログインプロトコルを開始します。アドレスバッファーはクリアされ、外部ホストのインターネットアドレスと、ログインサーバーが待機するポート番号が書き込まれます。

```
memset((char *) &server, 0, sizeof server);
bzero (&sin6, sizeof (sin6));
memcpy((char*) &server.sin6_addr,hp->h_addr,hp->h_length);
server.sin6_family = hp->h_addrtype;
server.sin6_port = sp->s_port;
```

ソケットが作成され、接続が開始されます。s のバインドが解除されるため、connect(3SOCKET) は暗黙に bind(3SOCKET) を実行します。

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
  perror("rlogin: socket");
  exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof server) < 0) {
  perror("rlogin: connect");
  exit(4);
}
```

コネクションレス型のサーバー

サービスの中にはデータグラムソケットを使用するものがあります。rwho(1) サービスは、LAN に接続されたホストのステータス情報を提供します (in.rwhod(1M) は大量のネットワークトラフィックを発生させるため、実行を避けてください)。rwho(1) サービスを利用するには、特定のネットワークに接続されたすべ

てのホストに情報をブロードキャスト送信できる環境が必要です。これは、データグラムソケットを使用する例の 1 つです。

rwho(1) サーバーを実行するホスト上のユーザーは、ruptime(1) を使用して別のホストの現在のステータスを取得できます。典型的な出力を例 2–10 に示します。

例 2–10 ruptime(1) プログラムの出力

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

各ホストには、rwho(1) サーバープロセスによってステータス情報が周期的にブロードキャスト送信されます。このサーバープロセスもステータス情報を受信し、データベースを更新します。このデータベースは、各ホストのステータスのために解釈されます。サーバーはそれぞれ個別に動作し、ローカルネットワークとそのブロードキャスト機能によってのみ結合されます。

大量のネットトラフィックが生成されるため、ブロードキャストの使用はかなり困難です。サービスが広範に、かつ頻繁に使用されない限り、周期的なブロードキャストのために手間がかかり簡潔さが失われることになります。

例 2–11 は、rwho(1) サーバーの簡潔な例を示しています。このコードは、ネットワーク上のほかのホストによるステータス情報ブロードキャストを受信し、そのホストのステータスを供給するという 2 つのタスクを行います。最初のタスクは、プログラムのメインループで行われます。別の rwho(1) サーバープロセスによって送信されたことを確認するために rwho(1) ポートで受信されたパケットがチェックされ、到着時刻が記録されます。パケットは、続いてホストのステータスを使用してファイルを更新します。一定の時間内にホストからの通信がない場合、データベースルーチンはホストがダウンしていると想定し、そのことを記録します。ホストが稼動している間サーバーがダウンしていることがあるため、このアプリケーションはエラーになりがちです。

例 2–11 rwho(1) サーバー

```
main()
{
   ...
   sp = getservbyname("who", "udp");
   net = getnetbyname("localnet");
```

```
    sin.sin6_addr = inet_makeaddr(net->n_net, in6addr_any);
    sin.sin6_port = sp->s_port;
    ...
    s = socket(AF_INET6, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
          == -1) {
       syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
       exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    while(1) {
       struct whod wd;
        int cc, whod, len = sizeof from;
       cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
          (struct sockaddr *) &from, &len);
       if (cc <= 0) {
       if (cc == -1 && errno != EINTR)
          syslog(LOG_ERR, "rwhod: recv: %m");
       continue;
       }
       if (from.sin6_port != sp->s_port) {
          syslog(LOG_ERR, "rwhod: %d: bad from port",
             ntohs(from.sin6_port));
          continue;
       }
       ...
       if (!verify( wd.wd_hostname)) {
          syslog(LOG_ERR, "rwhod: bad host name from %x",
             ntohl(from.sin6_addr.s6_addr));
          continue;
       }
       (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
       whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
       ...
       (void) time(&wd.wd_recvtime);
       (void) write(whod, (char *) &wd, cc);
       (void) close(whod);
    }
    exit(0);
}
```

2 つ目のサーバータスクは、そのホストのステータスの供給です。このタスクでは、周期的にシステムステータス情報を取得し、その情報をメッセージとしてパッケージ化し、ほかの `rwho(1)` サーバーに知らせるためローカルネットワーク上でそれをブロードキャスト送信する必要があります。このタスクはタイマーで実行され、シグナルによって起動されます。システムステータス情報の位置確認が伴いますが、重要なものではありません。

ステータス情報が、ローカルネットワーク上でブロードキャスト送信されます。ブロードキャストをサポートしないネットワークでは、別の方式を使用してください。

拡張機能

分散型アプリケーションを構築する場合、通常は、これまでに説明したメカニズムで十分対応できます。この節に挙げた機能は必要に応じて参照してください。

帯域外データ

ストリームソケットの抽象化には、帯域外データが含まれます。帯域外データは、接続されたストリームソケットペア間の論理的に独立した伝送チャネルです。帯域外データは、通常のデータとは無関係に配信されます。帯域外データ機能が使用される場合、一度に 1 つ以上の帯域外メッセージが確実に配信されなければなりません。このメッセージには 1 バイト以上のデータを含むことができ、いつでも 1 つ以上のメッセージの配信を保留できます。

帯域内シグナリングだけをサポートする (つまり、緊急のデータが通常のデータと共に順序どおり配信される) 通信プロトコルでは、メッセージは通常のデータストリームから抽出され、個別に格納されます。これにより、ユーザーは間に入るデータをバッファリングする必要にせまられることなく、緊急データを順番に受信するか順不同で受信するかを選択できます。

帯域外データは、MSG_PEEK を使用して先読みできます。ソケットにプロセスグループがある場合は、その存在がプロトコルに通知される時に SIGURG シグナルが生成されます。SIGIO に関する 58ページの「割り込み方式のソケット入出力」で説明しているように、プロセスは適切な fcntl(2) 呼び出しを使用して、プロセスグループまたはプロセス ID が SIGURG によって通知されるように設定できます。複数のソケットに配信待ちの帯域外データがある場合は、select(3C) 呼び出しを使用してそのようなデータ保留のあるソケットを確認できます。

帯域外データが送信された位置のデータストリームには、論理マークが置かれます。リモートログインアプリケーションとリモートシェルアプリケーションは、この機能を使用してクライアントプロセスとサーバープロセス間にシグナルを伝えます。シグナルが受信された時点で、データストリームのそのマークまでのデータがすべて破棄されます。

帯域外メッセージを送信するには、MSG_OOB フラグを send(3SOCKET) または sendto(3SOCKET) に適用します。帯域外メッセージを受信するには、順に取得する場合を除き MSG_OOB を recvfrom(3SOCKET) または recv(3SOCKET) に指定します (順に取得する場合は、MSG_OOB フラグは不要)。SIOCATMARK ioctl(2) は、

読み取りポインタが現在、データストリーム内のマークを指しているかどうかを示します。

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

戻り時に *yes* が 1 の場合、次の読み取りはマークのあとのデータを返します。*yes* が 1 でない場合は、帯域外データが到着したと仮定して、次の読み取りは帯域外シグナルを送信する前にクライアントによって送信されたデータを提供します。割り込みまたは停止のシグナルを受ける場合に出力をフラッシュするリモートログインプロセス内のルーチンを、例 2–12 に示します。このコードは、マークまでの通常のデータを (これを破棄するために) 読み取り、続いて帯域外バイトを読み取ります。

プロセスは、初めにマークまでを読み取らずに、帯域外データの読み取りまたは先読みを行うことができます。これは、プロトコルが緊急の帯域内データを通常のデータと共に配信し、前もってその存在の通知だけを行う場合には困難になります (例: インターネットファミリ内のソケットストリームの提供に使用されるプロトコル、TCP)。このようなプロトコルでは、MSG_OOB フラグによって recv(3SOCKET) が行われた時点で帯域外バイトが到着していないことがあります。このような場合、呼び出しはエラー EWOULDBLOCK を返します。また、入力バッファー内に十分な帯域内データが存在し、バッファーが空になるまでは通常のフロー制御がピアによる緊急データの送信を防止する場合もあります。この場合プロセスは、待ち行列に入ったデータを十分に読み取ってからでないと緊急データを配信できません。

例 2–12　帯域外データの受信時における端末入出力のフラッシュ

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
  int out = FWRITE;
  char waste[BUFSIZ];
  int mark = 0;

  /* ローカル端末出力をフラッシュする */
  ioctl(1, TIOCFLUSH, (char *) &out);
  while(1) {
   if (ioctl(rem, SIOCATMARK, &mark) == -1) {
    perror("ioctl");
    break;
   }
   if (mark)
    break;
   (void) read(rem, waste, sizeof waste);
  }
  if (recv(rem, &mark, 1, MSG_OOB) == -1) {
   perror("recv");
   ...
```

```
  }
  ...
}
```

ソケットストリーム内の緊急のインラインデータの位置を保持する機能もあります。この機能は、ソケットレベルのオプション SO_OOBINLINE として提供されています。使用方法については、getsockopt(3SOCKET) のマニュアルページを参照してください。このオプションを使用すると緊急データの位置 (マーク) は保持されますが、緊急データは MSG_OOB フラグなしで返される通常のデータストリーム内のマークの直後に続きます。複数の緊急指示を受信するとマークは移動しますが、帯域外データが消失することはありません。

非ブロックソケット

一部のアプリケーションは、ブロックしないソケットを必要とします。たとえば、すぐに完了できず (完了を待っている間に) プロセスを一時的に中断させてしまう要求は実行されません。エラーコードが返されます。ソケットが作成され、別のソケットに対する接続が確立されたあと、例 2–13 に示すように fcntl(2) 呼び出しを発行してこのソケットを非ブロックに設定できます。

例 2–13 非ブロックソケットの設定

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
  perror("fcntl F_GETFL");
  exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
  perror("fcntl F_SETFL, FNDELAY");
  exit(1);
}
...
```

非ブロックソケットで入出力を行う場合は、オペレーションが正常にブロックする場合に発生する、errno.h 内のエラー EWOULDBLOCK をチェックしてください。accept(3SOCKET)、connect(3SOCKET)、send(3SOCKET)、recv(3SOCKET)、read(2)、および write(2) はすべて、EWOULDBLOCK を返すことができます。send(3SOCKET) のようなオペレーションを完全には実行でき

ないが、ストリームソケットを使用する場合などに部分的な書き込みは可能という場合には、すぐに送信できるデータは処理され、実際に送信された量が関数の戻り値となります。

非同期ソケット入出力

複数の要求を同時に処理するアプリケーションでは、プロセス間の非同期通信を必要とします。非同期ソケットは、SOCK_STREAM タイプでなければなりません。ソケットを非同期にするには、例 2–14 に示すように fcntl(2) 呼び出しを発行します。

例 2–14 ソケットを非同期にする

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == -1)
  perror("fcntl F_GETFL");
  exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == -1)
  perror("fcntl F_SETFL, FNDELAY | FASYNC");
  exit(1);
}
...
```

ソケットの初期化と接続が終わり、非同期に設定されると、ファイルを非同期で読み書きする場合のように通信が行われます。send(3SOCKET)、write(2)、recv(3SOCKET)、read(2) は、データ転送を開始します。次の節で説明しているように、データ転送はシグナル方式の入出力ルーチンによって行われます。

割り込み方式のソケット入出力

SIGIO シグナルは、ソケット (実際には任意のファイル記述子) がデータ転送を終了した時点をプロセスに通知します。SIGIO を使用する手順は次のとおりです。

- signal(3C) 呼び出しまたは sigvec(3UCB) 呼び出しを使用して、SIGIO シグナルハンドラを設定する

- シグナルの経路をそれ自体のプロセス ID またはプロセスグループ ID に指定するため、fcntl(2) を使用してプロセス ID またはプロセスグループ ID を設定する (ソケットのデフォルトのプロセスグループはグループ 0)
- 58ページの「非同期ソケット入出力」に示すように、ソケットを非同期に変換する

例 2–15 は、ソケットで要求が保留される場合にその保留要求に関する情報を特定のプロセスが受信する例を示しています。SIGURG のハンドラを追加すると、このコードは SIGURG シグナルを受信する目的でも使用できます。

例 **2–15** 入出力要求の非同期通知

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* SIGIO または SIGURG シグナルを受信するプロセスを s に設定する。*/
if (fcntl(s, F_SETOWN, getpid()) < 0) {
  perror("fcntl F_SETOWN");
  exit(1);
}
```

シグナルとプロセスグループ ID

SIGURG と SIGIO の場合、各ソケットにはプロセス番号とプロセスグループ ID があります。これらの値はゼロに初期化されますが、先の例で示しているように、F_SETOWN fcntl(2) を使用してあとから定義し直すことができます。fcntl(2) の 3 番目の引数が正の場合、ソケットのプロセス ID を設定します。fcntl(2) の 3 番目の引数が負の場合、ソケットのプロセスグループ ID を設定します。SIGURG シグナルと SIGIO シグナルの受信側として許可されるのは、呼び出し側のプロセスだけです。同様に、fcntl(2)、F_GETOWN は、ソケットのプロセス番号を返します。

SIGURG と SIGIO の受信は、ioctl(2) を使用してソケットをユーザーのプロセスグループに割り当てることによっても有効にできます。

```
/* oobdata はルーチンを処理する帯域外データ */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
  perror("ioctl: SIOCSPGRP");
}
```

サーバープロセスで便利なシグナルとして、ほかに SIGCHLD が挙げられます。このシグナルは、任意の子プロセスがその状態を変更した場合にプロセスに配信されます。通常、サーバーはこのシグナルを使用して、明示的に終了を待機せずに、あ

るいは終了ステータスを周期的にポーリングせずに、終了した子プロセスの「リープ (刈り取り)」を行います。たとえば、先に示したリモートログインサーバーループは、例 2–16 に示すように拡大できます。

例 2–16 SIGCHLD シグナル

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
  int g, len = sizeof from;
  g = accept(f, (struct sockaddr *) &from, &len);
  if (g < 0) {
   if (errno != EINTR)
    syslog(LOG_ERR, "rlogind: accept: %m");
   continue;
  }
  ...
}

#include <wait.h>

reaper()
{
  int options;
  int error;
  siginfo_t info;

  options = WNOHANG | WEXITED;
  bzero((char *) &info, sizeof(info));
  error = waitid(P_ALL, 0, &info, options);
}
```

親サーバープロセスがその子のリープに失敗する場合、ゾンビプロセスが生じます。

特定のプロトコルの選択

socket(3SOCKET) 呼び出しの 3 番目の引数が 0 の場合、socket(3SOCKET) は要求されたタイプである戻されたソケットにデフォルトプロトコルを使用することを選択します。通常はデフォルトプロトコルで十分であり、ほかの選択肢はありません。raw ソケットを使用して低レベルのプロトコルやハードウェアインタフェースと直接通信を行う場合は、必要に応じてプロトコル引数で非多重化を設定してください。たとえば、インターネットファミリ内の raw ソケットを使用して IP 上に新しいプロトコルを実装できます。この場合、ソケットは指定されたプロトコルのパケットだけを受信します。特定のプロトコルを取得するには、プロトコルファミリで定義されているようにプロトコル番号を決定します。インターネットファミリ

の場合、44ページの「標準のルーチン」で説明しているライブラリルーチンの 1 つ (getprotobyname(3SOCKET) など) を使用してください。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

このコードでは、ソケット s はストリームベースの接続を使用しますが、プロトコルタイプはデフォルトの tcp ではなく newtcp を使用します。

アドレスのバインド

TCP と UDP は、ローカル *IP* アドレス、ローカルポート番号、外部 *IP* アドレス、および外部ポート番号の 4 つを使用してアドレス指定を行います。TCP では、これらの 4 つの要素は一意でなければなりません。UDP にはこのような要求はありません。ホストは複数のネットワークに常駐でき、ユーザーは割り当てられているポート番号に直接アクセスできないため、ユーザープログラムがローカルアドレスとローカルポートに使用する適切な値を常に認識できるとは限りません。この問題を避けるため、アドレスの一部を指定せずにおき、必要時にシステムにこれらの部分を適切に割り当てることができます。これらの要素の各部は、ソケット API のさまざまな部分によって指定できます。

bind(3SOCKET)	ローカルアドレスまたはローカルポート (あるいはこの両方)
connect(3SOCKET)	外部アドレスと外部ポート

accept(3SOCKET) 呼び出しは、外部クライアントからの接続情報を検出します。そのため、この呼び出しによってローカルアドレスとローカルポートがシステムに指定され (accept(3SOCKET) の呼び出し側が何も指定しなくても)、外部アドレスと外部ポートが返されます。

listen(3SOCKET) を呼び出すと、ローカルポートが選択されます。ローカル情報を割り当てるために明示的な bind(3SOCKET) がなされていない場合に listen(3SOCKET) を呼び出すと、一時的なポート番号が割り当てられます。

特定のポートに常駐するが、どのローカルアドレスが選択されてもかまわないというサービスは、それ自体をそのポートに bind(3SOCKET) し、ローカルアドレスを指定しないままにしておくことができます。このためには、<netinet/in.h>

内の定数値を持つ変数 in6addr_any に設定します。ローカルポートを固定する必要がない場合、listen(3SOCKET) を呼び出すと、ポートが選択されます。アドレス in6addr_any またはポート番号 0 を指定することを、ワイルドカードの使用と呼びます (AF_INET の場合、in6addr_any ではなく INADDR_ANY を使用する)。

ワイルドカードアドレスは、インターネットファミリにおけるローカルアドレスのバインドを簡易化します。次のコード例は、特定のポート番号 MYPORT をソケットにバインドし、ローカルアドレスは指定しないままにします。

例 2–17 ソケットにポート番号をバインドする

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in6 sin;
...
  s = socket(AF_INET6, SOCK_STREAM, 0);
  bzero (&sin6, sizeof (sin6));
  sin.sin6_family = AF_INET6;
  sin.sin6_addr.s6_addr = in6addr_any;
  sin.sin6_port = htons(MYPORT);
  bind(s, (struct sockaddr *) &sin, sizeof sin);
```

ホスト上の各ネットワークインタフェースは、通常、一意の IP アドレスを持ちます。ワイルドカードローカルアドレスを持つソケットは、指定されたポート番号にあてたメッセージと、ホストに割り当てられた可能性のあるアドレスに送信されたメッセージを受信できます。たとえば、ホストにアドレス 128.32.0.4 と 10.0.0.78 を持つ 2 つのインタフェースがあり、ソケットが例 2–17 のようにバインドされている場合、プロセスは 128.32.0.4 または 10.0.0.78 にアドレス指定された接続要求を受け入れることができます。特定のネットワーク上のホストだけに接続を許可するには、サーバーは適切なネットワーク上のインタフェースのアドレスをバインドします。

同様に、ローカルポート番号は指定しないままにしておくことができます (0 を指定)。この場合、システムがポート番号を選択します。次に、特定のローカルアドレスをソケットにバインドし、ローカルポート番号を指定しないままにする例を示します。

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, ":ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

システムは、次の 2 つの基準でローカルポート番号を選択します。

- 1024 未満のインターネットポート番号 (IPPORT_RESERVED) は、特権ユーザー (スーパーユーザー) 用として予約されています。非特権ユーザーは、1024 を超え

る任意のインターネットポート番号を使用できます。インターネットポート番号の最大値は 65535 です。

- 現在ほかのソケットにバインドされていないポート番号であること。

クライアントのポート番号と IP アドレスは、accept(3SOCKET) (*from* の結果) または getpeername(3SOCKET) で見つけます。

ポート番号を選択するためにシステムが使用するアルゴリズムがアプリケーションに適さない場合もあります。これは、関連付けが 2 段階のプロセスで作成されるためです。たとえば、インターネットファイル転送プロトコルでは、データ接続は常に同じローカルポートから発生しなければならないと定めています。しかし、異なる外部ポートに接続することによって、関連付けの重複が避けられます。この場合、前のデータ接続のソケットが存在していると、システムは同じローカルアドレスとポート番号をソケットにバインドすることを許可しません。デフォルトのポート選択アルゴリズムを無効にするには、次に示すようにアドレスをバインドする前にオプション呼び出しを行う必要があります。

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

この呼び出しを行うと、すでに使用されているローカルアドレスをバインドできます。同じローカルアドレスとローカルポートを持つほかのソケットが同じ外部アドレスと外部ポートを持たないことをシステムが接続時に検証するため、この呼び出しは一意性という条件に違反することはありません。関連付けがすでに存在する場合、エラー EADDRINUSE が返されます。

マルチキャストの使用

IP マルチキャストは、タイプが SOCK_DGRAM または SOCK_RAW であるソケット AF_INET6 または AF_INET およびインタフェースドライバがマルチキャストをサポートするサブネットワーク上でだけサポートされます。

IPv4 マルチキャストデータグラムの送信

マルチキャストデータグラムを送信するには、sendto(3SOCKET) 呼び出しで宛先アドレスとして 224.0.0.0 ～ 239.255.255.255 の範囲の IP マルチキャストアドレスを指定します。

デフォルトでは、IP マルチキャストデータグラムは存続期間 (TTL) 1 で送信されます。この場合、単一のサブネットワーク外にデータグラムが転送されることはありません。ソケットオプション IP_MULTICAST_TTL を指定すると、後続のマルチキャストデータグラムの TTL を 0 ～ 255 の範囲の任意の値に設定してマルチキャストの配信範囲を制御できます。

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

TTL 0 のマルチキャストデータグラムはいかなるサブネット上でも伝送されませんが、送信ホストが宛先グループに属しており、かつ送信側ソケットでマルチキャストループバックが無効になっていない場合は、ローカルに配信できます (下記を参照)。最初の配信先となるサブネットに 1 つ以上のマルチキャストルーターが接続されている場合は、1 を超える TTL を持つマルチキャストデータグラムを複数のサブネットに配信できます。意味のある配信範囲制御を提供するため、マルチキャストルーターは TTL に「しきい値」の概念をサポートします。この概念は、一定の TTL 未満のデータグラムが特定のサブネットを越えることを防止します。マルチキャストデータグラムの初期 TTL と、それらに対してしきい値が強制する規則を次に示します。

0	同じホストに制限される
1	同じサブネットに制限される
32	同じサイトに制限される
64	同じ地域に制限される
128	同じ大陸に制限される
255	配信範囲内で制限されない

「サイト」と「地域」は厳密には定義されません。サイトは、ローカルな事柄として、小さな管理単位にさらに分割できます。

アプリケーションは、上記の TTL 以外に初期 TTL を選択できます。たとえば、アプリケーションは、TTL シーケンス 0、1、2、4、8、16、32 を使用し、TTL 0 から開始して応答が得られるまでより大きな TTL のマルチキャスト照会を送ることによってネットワークリソースの「拡張リング検索」が行えます。

マルチキャストルーターは、224.0.0.0 ～ 224.0.0.255 の範囲の宛先アドレスを持つマルチキャストデータグラムの転送を、TTL の値にかかわらず拒否します。この範囲のアドレスは、経路指定プロトコルとその他の低レベルトポロジの発見または保守プロトコル (ゲートウェイ発見、グループメンバーシップ報告など) の使用に予約されています。

ホストにマルチキャスト機能を持つ複数のインタフェースがある場合でも、各マルチキャスト伝送は単一のネットワークインタフェースから送信されます (ホストがマルチキャストルーターでもあり、TTL が 1 を超える場合には、発信インタフェース以外のインタフェースにマルチキャストを転送できる)。一定のソケットからの後続の伝送については、ソケットオプションを使用してデフォルトを無効にできます。

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr))
```

addr は、使用する発信インタフェースのローカル IP アドレスです。デフォルトインタフェースに戻すには、アドレス INADDR_ANY を指定します。インタフェースのローカル IP アドレスは、SIOCGIFCONF ioctl を使用して取得します。インタフェースがマルチキャストをサポートするかどうかを確認するには、SIOCGIFFLAGS ioctl を使用してインタフェースフラグを取り出し、IFF_MULTICAST フラグが設定されているかどうかをテストしてください。このオプションは、インターネットトポロジと明確な関係があるマルチキャストルーターなどのシステムサービスを主な対象としています。

(発信インタフェース上で) 送信ホスト自体が属しているグループにマルチキャストデータグラムが送信されると、ローカル配信の IP 層によってデータグラムのコピーがデフォルトでループバックされます。後続のデータグラムがループバックされるかどうかにかかわらず、別のソケットオプションが送信側に明示的な制御を与えます。

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop))
```

このコードの loop の値は、ループバックを無効にする場合は 0、ループバックを有効にする場合は 1 です。単一のホストにインスタンスを 1 つしか持たないアプリケーション (ルーターやメールデーモンなど) では、このオプションを使用するとアプリケーション自体の伝送を受信するオーバーヘッドが排除されるため、パフォーマンスが向上します。このオプションは、単一のホストに複数のインスタンスを持つアプリケーション (会議システムプログラムなど) や送信側が宛先グループに属さないアプリケーション (時間照会プログラムなど) には通常使用しないでください。

送信ホストが別のインタフェースの宛先グループに属している場合、1 を超える TTL で送信されたマルチキャストデータグラムは、他方のインタフェース上の送信ホストに配信できます。このような配信には、ループバック制御オプションは何の効果もありません。

IPv4 マルチキャストデータグラムの受信

ホストは、IP マルチキャストデータグラムを受信する前に、1 つ以上の IP マルチキャストグループのメンバーになる必要があります。プロセスは、次のソケットオプションを使用して、マルチキャストグループに加わるようにホストに求めることができます。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

mreq は次の構造体です。

```
struct ip_mreq {
    struct in_addr imr_multiaddr;   /* 加わるマルチキャストグループ */
    struct in_addr imr_interface;   /* 加わるインタフェース */
}
```

各メンバーシップは単一のインタフェースに関連付けられ、複数のインタフェース上の同じグループに加わることができます。imr_interface が in6addr_any になるように 指定してデフォルトのマルチキャストインタフェースを選択するか、あるいはホストのローカルアドレスの 1 つを指定して特定の (マルチキャスト機能を持った) インタフェースを選択してください。

メンバーシップを取り消すには、次のコードを使用します。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq))
```

mreq には、メンバーシップの追加に使用したのと同じ値が入ります。ソケットが閉じられるか、あるいはソケットを保持しているプロセスが停止される時には、ソケットに関連付けられたメンバーシップも取り消されます。特定のグループ内で複数のソケットがメンバーシップを要求でき、ホストは最後の要求が取り消されるまでそのグループのメンバーにとどまります。

ソケットのどれかがデータグラムの宛先グループ内でメンバーシップを要求した場合、カーネル IP 層が受信マルチキャストパケットを受け入れます。特定のソケットに対するマルチキャストデータグラムの配信は、単一キャストデータグラムの場合

と同様に、宛先ポートと、ソケットに関連付けられたメンバーシップ (raw ソケットの場合はプロトコルタイプ) に基づいています。特定のポートに送信されたマルチキャストデータグラムを受信するには、ローカルアドレスを未指定のまま (INADDR_ANY などを指定) ローカルポートにバインドします。

次の指定が bind(3SOCKET) の前にあると、複数のプロセスを同じ SOCK_DGRAM UDP ポートにバインドできます。

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

この場合、共有ポートに向けられた各受信マルチキャストまたは受信ブロードキャスト UDP データグラムは、そのポートにバインドされているすべてのソケットに配信されます。下位互換性の理由から、これは単一キャストの受信データグラムには適用されません。データグラムの宛先ポートにバインドされているソケットの数にかかわらず、単一キャストデータグラムが複数のソケットに配信されることはありません。SOCK_RAW ソケットは、SO_REUSEADDR オプションがなくても単一の IP プロトコルタイプを共有できます。

マルチキャストに関連する新しいソケットオプションの説明は、<netinet/in.h> を参照してください。IP アドレスはすべて、ネットワークバイトオーダーで渡されます。

IPv6 マルチキャストデータグラムの送信

マルチキャストデータグラムを送信するには、sendto(3SOCKET) 呼び出しで宛先アドレスとして ff00::0/8 の範囲の IP マルチキャストアドレスを指定します。

デフォルトでは、IP マルチキャストデータグラムはホップ制限 1 で送信されます。この場合、単一のサブネットワーク外にデータグラムが転送されることはありません。ソケットオプション IPV6_MULTICAST_HOPS を指定すると、後続のマルチキャストデータグラムのホップ制限を 0 から 255 の範囲の任意の値に設定してマルチキャストの配信範囲を制御できます。

```
uint_l;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,sizeof(hops))
```

ホップ制限 0 のマルチキャストデータグラムはどのサブネットにも伝送されませんが、送信ホストが宛先グループに属しており、送信側ソケットでマルチキャストループバックが無効になっていない場合は、ローカルに配信できます (次を参照)。

最初の配信先となるサブネットに 1 つ以上のマルチキャストルーターが接続されている場合は、1 を超えるホップ制限を持つデータグラムを複数のサブネットに配信できます。IPv4 マルチキャストアドレスと異なり、IPv6 マルチキャストアドレスには、アドレスの最初の部分にコード化された明示的な配信範囲情報が含まれます。定義されている配信範囲を次に示します (x は未指定)。

`ffX1::0/16`	ノード – ローカルな配信範囲 — 同じノードに制限される
`ffX2::0/16`	リンク – ローカルな配信範囲
`ffX5::0/16`	サイト – ローカルな配信範囲
`ffX8::0/16`	組織 – ローカルな配信範囲
`ffXe::0/16`	全世界的な配信範囲

アプリケーションは、マルチキャストアドレスの配信範囲とは個別に、異なるホップ制限値を使用できます。たとえば、アプリケーションは、ホップ制限シーケンス 0、1、2、4、8、16、32 を使用し、ホップ制限 0 から開始して応答が受信されるまでより大きなホップ制限のマルチキャスト照会を送信することにより、ネットワークリソースの「拡張リング検索」を実行する場合があります。

ホストにマルチキャスト機能を持つ複数のインタフェースがある場合でも、各マルチキャスト伝送は単一のネットワークインタフェースから送信されます (ホストがマルチキャストルーターでもあり、ホップ制限 が 1 を超える場合には、発信インタフェース以外のインタフェースにマルチキャストを転送できる)。一定のソケットからの後続の伝送については、ソケットオプションを使用してデフォルトを無効にできます。

```
uint_t ifindex;

ifindex = if_nametoindex )"hme3");
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex, sizeof(ifindex))
```

`ifindex` は、希望する発信インタフェースのインタフェースインデックスです。デフォルトインタフェースに戻すには、値 `0` を指定します。

(発信インタフェース上で) 送信ホスト自体が属しているグループにマルチキャストデータグラムが送信されると、ローカル配信の IP 層によってデータグラムのコピーがデフォルトでループバックされます。後続のデータグラムがループバックされるかどうかにかかわらず、別のソケットオプションが送信側に明示的な制御を与えます。

```
uint_t loop;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop))
```

loop の値は、ループバックを無効にする場合は 0、ループバックを有効にする場合は 1 です。単一のホストにインスタンスを 1 つしか持たないアプリケーション (ルーターやメールデーモンなど) では、このオプションを使用するとアプリケーション自体の伝送を受信するオーバーヘッドが排除されるため、パフォーマンスが向上します。このオプションは、単一のホストに複数のインスタンスを持つアプリケーション (会議システムプログラムなど) や送信側が宛先グループに属さないアプリケーション (時間照会プログラムなど) に通常は使用しないでください。

送信ホストが別のインタフェースの宛先グループに属している場合、1 以上のホップ制限で送信されたマルチキャストデータグラムは、他方のインタフェース上の送信ホストに配信できます。このような配信には、ループバック制御オプションは何の効果もありません。

IPv6 マルチキャストデータグラムの受信

ホストは、IP マルチキャストデータグラムを受信する前に、1 つ以上の IP マルチキャストグループのメンバーになる必要があります。プロセスは、次のソケットオプションを使用して、マルチキャストグループに加わるようにホストに求めることができます。

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq, sizeof(mreq))
```

mreq は次の構造体です。

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;   /* IPv6 マルチキャストアドレス */
    unsigned int    ipv6mr_interface;   /* インタフェースインデックス */
}
```

各メンバーシップは単一のインタフェースに関連付けられ、複数のインタフェース上の同じグループに加わることができます。ipv6_interface が 0 になるように指定してデフォルトのマルチキャストインタフェースを選択するか、あるいはホストのインタフェースの 1 つのインタフェースインデックスを指定してその (マルチキャスト機能を持った) インタフェースを選択してください。

グループから抜けるには、次のコードを使用します。

```
struct ipv6_mreq mreq;
```

```
setsockopt(sock, IPPROTO_IPV6, IP_LEAVE_GROUP, &mreq, sizeof(mreq))
```

mreq には、メンバーシップの追加に使用した同じ値が入ります。ソケットが閉じられるか、あるいはソケットを保持しているプロセスが停止する時には、ソケットに関連付けられたメンバーシップも取り消されます。特定のグループ内で複数のソケットがメンバーシップを要求でき、ホストは最後の要求が取り消されるまでそのグループのメンバーにとどまります。

ソケットのどれかがデータグラムの宛先グループ内でメンバーシップを要求した場合、カーネル IP 層が受信マルチキャストパケットを受け入れます。特定のソケットに対するマルチキャストデータグラムの配信は、単一キャストデータグラムの場合と同様に、宛先ポートと、ソケットに関連付けられたメンバーシップ (raw ソケットの場合はプロトコルタイプ) に基づいています。特定のポートに送信されたマルチキャストデータグラムを受信するには、ローカルアドレスを未指定のまま (INADDR_ANY などに指定) ローカルポートにバインドします。

次の指定が bind(3SOCKET) の前にあると、複数のプロセスを同じ SOCK_DGRAM UDP ポートにバインドできます。

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

この場合、共有ポートに向けられた各受信マルチキャスト UDP データグラムは、そのポートにバインドされているすべてのソケットに配信されます。下位互換性の理由から、これは単一キャストの受信データグラムには適用されません。データグラムの宛先ポートにバインドされているソケットの数にかかわらず、単一キャストデータグラムが複数のソケットに配信されることはありません。SOCK_RAW ソケットは、SO_REUSEADDR オプションがなくても単一の IP プロトコルタイプを共有できます。

マルチキャストに関連する新しいソケットオプションの説明は、<netinet/in.h> を参照してください。IP アドレスはすべて、ネットワークバイトオーダーで渡されます。

ゼロコピーとチェックサム負荷解除

SunOS 5.6 以降では、TCP/IP プロトコルスタックは、ゼロコピーと TCP チェックサム負荷解除という 2 つの新しい機能をサポートするように機能が拡張されています。

- ゼロコピーは、仮想メモリ MMU の再マッピングと、書き込み時にコピーを行う手法を使用して、アプリケーションとカーネル空間の間でデータを移動します。

- チェックサム負荷解除は、特殊なハードウェアロジックによりTCP チェックサム計算の負荷を解除します。

注 - ゼロコピーとチェックサム負荷解除は互いに機能的には依存していませんが、最大のパフォーマンスを得るには連携して動作する必要があります。チェックサム負荷解除はネットワークインタフェースからのハードウェアサポートを必要とし、このハードウェアサポートがないとゼロコピーは有効になりません。

ゼロコピーを実行するためには VM ページの再マッピングを適用する前に、アプリケーションがページ型のバッファーを供給する必要があります。負荷が高い書き込み時コピーの失敗を避けるには、アプリケーションは伝送側に大きな循環バッファーを使用する必要があります。一般的なバッファー割り当ては 16 の 8K バッファーです。

ソケットオプション

送信バッファー領域または受信バッファー領域の変更などにより、`setsockopt(3SOCKET)` と `getsockopt(3SOCKET)` を使用してソケットオプションのいくつかを設定、取得できます。呼び出しの一般的な書式を次に示します。

```
setsockopt(s, level, optname, optval, optlen);
```

および

```
getsockopt(s, level, optname, optval, optlen);
```

場合によっては (バッファーサイズを設定する場合など)、これらの指定がオペレーティングシステムに対する参考情報にしかならない場合があります。値を適切に調整する権限はオペレーティングシステムに与えられています。

表 2–4 に、これらの呼び出しの引数を示します。

表 2–4 `setsockopt(3SOCKET)` と `getsockopt(3SOCKET)` の引数

引数	説明
s	オプションの適用先であるソケット
level	`sys/socket.h` 内の 記号定数 `SOL_SOCKET` によって示されるプロトコルレベル (ソケットレベルなど) を指定する

表 2–4　setsockopt(3SOCKET) と getsockopt(3SOCKET) の引数　続く

引数	説明
optname	オプションを指定する、sys/socket.h で定義されている記号定数
optval	オプションの値を示す
optlen	オプションの値の長さを示す

getsockopt(3SOCKET) の場合、*optlen* は、初期状態では *optval* によって示される記憶領域のサイズに設定され、最終的に使用された記憶領域の長さに設定されます。

既存ソケットのタイプ (ストリームやデータグラムなど) を決定すると便利な場合があります。inetd(1M) によって呼び出されるプログラムは、SO_TYPE ソケットオプションと getsockopt(3SOCKET) 呼び出しを使用してこの決定を行うことができます。

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

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) <0) {
  ...
}
```

getsockopt(3SOCKET) のあと、sys/socket.h で定義しているように、type はソケットタイプの値に設定されます。データグラムソケットの場合、type は SOCK_DGRAM です。

inetd(1M) デーモン

システムに装備されているデーモンの 1 つが inetd(1M) です。システムの起動時に呼び出され、 /etc/inet/inetd.conf ファイルを読み、その内容に従いサービスを取得します。デーモンは /etc/inet/inetd.conf ファイルに記述されている各サービスごとに 1 つのソケットを作成し、各ソケットに適切なポート番号の割り当てを行います。inetd(1M) に関するより詳しい情報は、マニュアルページを参照してください。

inetd(1M) は各ソケットのポーリングを行い、そのソケットに対するサービスの接続要求待ちを行います。SOCK_STREAM タイプのソケットの場合、inetd(1M) は待機ソケットに対し accept(3SOCKET) を実行、新しいファイル記述子 0 および 1 (stdin および stdout) において、新しいソケットに対して fork(2)、dup(2) を実行し、その他の開かれているファイル記述子を終了させ適切なサーバーに対し exec(2) を実行します。

inetd(1M) の主な利点は、使用されていないサービスがシステム資源を使用しない点にあります。また、接続の確立に関する処理の大部分を inetd(1M) が請け負う点も大きな利点の 1 つです。inetd(1M) によって開始されたサーバーのソケットは、ファイル記述子 0 および 1 においてクライアントに接続され、ただちに read(2)、write(2)、send(3SOCKET)、または recv(3SOCKET) の実行が可能です。サーバーは stdio の取り決めに従い、必要な場合に fflush(3C) を利用する限りバッファー入出力の使用が可能です。

getpeername(3SOCKET) はソケットに接続されたピア (処理) のアドレスを戻り値として返します。これは inetd(1M) で開始されたサーバーにおいて、有効な手段です。たとえば、インターネットアドレス (たとえば、クライアントの IPv6 アドレスとしては定型的なアドレス fec0::56:a00:20ff:fe7d:3dd2 など) のログを記録する場合、inetd(1M) を使用しているサーバーならば以下が使用可能です。

```
struct sockaddr_storage name;
int namelen = sizeof (name);
char abuf[INET6_ADDRSTRLEN];
struct in6_addr addr6;
struct in_addr addr;

if (getpeername(fd, (struct sockaddr *)&name, &namelen) == -1) {
    perror("getpeername");
    exit(1);
} else {
    addr = ((struct sockaddr_in *)&name)->sin_addr;
    addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
    if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6 && IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* これは IPv4 によりマップされた IPv6 アドレスです。*/
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));

    }
    syslog("Connection from %s\n", abuf);
}
```

ブロードキャストとネットワーク構成の判定

ブロードキャストは IPv6 ではサポートされていません。サポートしているのは IPv4 のみです。

データグラムソケットにより送信されたメッセージは、接続されているネットワークのすべてのホストに届くようブロードキャストを行うことができます。システムのソフトウェア上でのブロードキャストのシミュレーションは行われないため、ネットワークがブロードキャストをサポートしていなければなりません。ブロードキャストはネットワーク上のすべてのホストにサービスを強制するため、ネットワークメッセージはネットワークに大きい負荷をかける場合があります。ブロードキャストは主に 2 つの目的に使用されます。アドレスが不明なローカルネットワーク上の資源の検索、またアクセス可能なすべての隣接ホストへ送られる情報のルーティングを行う場合に使用されます。

ブロードキャストメッセージの送信を行うには、インターネットデータグラムソケットを作成します。

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

ソケットのポート番号を割り当てます。

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

ネットワークのブロードキャストアドレス宛てに送信を行うことにより、データグラムは 1 つのネットワーク上のみでブロードキャストを行うことが可能です。また netinet/in.h 内で定義されている特別なアドレス INADDR_BROADCAST に送信することにより、接続されているすべてのネットワークに対し、データグラムのブロードキャストを行うことが可能です。

システムにはシステム上のネットワークインタフェースに関する情報の数 (IP アドレスおよびブロードキャストアドレスを含む) を判定するメカニズムが装備されています。SIOCGIFCONF ioctl(2) 呼び出しはホストのインタフェース構成を単一 ifconf 構造体において (戻り値として) 返します。この構造体にはホストが接続されている各ネットワークインタフェースによりサポートされる各アドレスファミリに対し個別の ifreq 構造体の配列が含まれます。例 2–18 ではこれらの構造体の net/if.h 内における定義を行っています。

例 **2–18** net/if.h

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* たとえば名前が "en0" */
union {
  struct sockaddr ifru_addr;
  struct sockaddr ifru_dstaddr;
  char ifru_oname[IFNAMSIZ]; /* 名前の場合、その他 */
  struct sockaddr ifru_broadaddr;
  short ifru_flags;
  int ifru_metric;
  char ifru_data[1]; /* インタフェース依存データ */
  char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

インタフェース構成を取り出す呼び出しは以下の通りです。

```
/*
 * インタフェースの数を検索するため SIOCGIFNUM ioctl を実行。
 *
 * 発見されたインタフェースの数に相当する空間を割り当て。
 *
 * 割り当てられたバッファーに対し SIOCGIFCONF を実行。
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
        numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
        perror("ioctl(SIOCGIFCONF)");
        exit(1);
}
...
}
```

この呼び出しの後、*buf* にはホストが接続されている各ネットワークに対して個別の ifreq 構造体の配列が格納されます。これらの構造体の順序付けは、まずインタ

フェース名で、次にサポートするアドレスファミリによって行われます。
ifc.ifc_len は ifreq 構造体に使用されるバイト数に合わせて設定されます。

各構造体は対応するネットワークの利用可能状況、ポイントツーポイント (point-to-point) またはブロードキャストなどを表示するインタフェースフラグのセットを持ちます。例 2–19 は SIOCGIFFLAGS ioctl(2) が ifreq 構造体により指定されたインタフェースのフラグを返すコードの例です。

例 2–19 インタフェースフラグの取得

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); —n >= 0; ifr++) {
   /*
    * 別の目的でアドレスファミリ用に使用されているインタフェースを
    * 使用しないよう注意する。
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
      ...
   }
     if ((ifr->ifr_flags & IFF_UP) == 0 ||
      (ifr->ifr_flags & IFF_LOOPBACK) ||
      (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
      continue;
}
```

例 2–20 では SIOCGIFBRDADDR ioctl(2) によってインタフェースのブロードキャストが取得可能であることを表現しています。

例 2–20 インタフェースのブロードキャストアドレス

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
  ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
  sizeof ifr->ifr_broadaddr);
```

SIOCGIFBRDADDR ioctl(2) はポイントツーポイントインタフェースのアドレスを取得する目的にも使用できます。

インタフェースブロードキャストアドレスの取得が完了したら、sendto(3SOCKET) を使用してブロードキャストデータグラムの送信を行います。

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

ブロードキャストまたはポイントツーポイントアドレスをサポートするホストが接続されている各インタフェースに対し 1 つの sendto(3SOCKET) を使用します。

第 3 章

XTI と TLI を使用したプログラミング

X/Open トランスポートインタフェース (XTI) およびトランスポート層インタフェース(TLI) はネットワークプログラミングインタフェースを構成する関数のセットです。 XTI は SunOS 4 用の旧 TLI インタフェースを発展させたものです。両インタフェースはサポートされていますが、 XTI がインタフェースセットの将来の方向を表しています。

XTI/TLI はマルチスレッドに対して安全

この章で取り上げるインタフェースはマルチスレッドに対して安全です。これは XTI/TLI 関数呼び出しを含むアプリケーションはマルチスレッドアプリケーション

内で自由に使用可能なことを意味します。アプリケーションの多重度は特定されていません。

XTI/TLI は非同期安全ではない

XTI/TLI インタフェースの非同期環境におけるふるまいは特定されていません。これらのインタフェースのシグナルハンドラルーチンからの使用は行わないことを推奨します。

XTI と TLI について

TLI は AT&T の System V Release 3 とともに 1986 年に導入されました。TLI は トランスポート層インタフェース API を規定しました。TLI インタフェースは ISO Transport Service Definition をモデルに設計され、 OSI トランスポート層とセッション層の間の API を提供します。その後 TLI インタフェースは AT&T の System V Release 4 の UNIX バージョンでさらに発展し、 SunOS 5.6 オペレーティングシステムインタフェースにも取り入れられました。

XTI インタフェースは TLI インタフェースを発展させたもので、このインタフェースの将来の方向を表しています。TLI を使用するアプリケーションとの互換性が保証されています。ただちに TLI のアプリケーションを XTI のアプリケーションに移行する必要性はありません。新しいアプリケーションは XTI インタフェースを使用し、古いアプリケーションは必要性が発生した時点で XTI に移行可能です。

TLI はアプリケーションがリンクするライブラリ (`libns1`) 内の関数呼び出しのセットとして実装されています。XTI アプリケーションは c89 フロントエンドを使用してコンパイルされ、 `xnet` ライブラリ (`libxnet`) とリンクされる必要があります。XTI におけるコンパイルに関する詳細は、`standards(5)` のマニュアルページを参照してください。

注 - XTI インタフェースを使用するアプリケーションは `xti.h` ヘッダーファイルを使用するのに対し、TLI インタフェースを使用するアプリケーションは `tiuser.h` ヘッダーファイルを使用しています。

XLI/TLI において必要不可欠なのがトランスポートエンドポイント (*transport endpoints*) およびトランスポートプロバイダ (*transport provider*) の概念です。トランスポートエンドポイントは通信を行う 2 つのエンティティであり、トランスポートプロバイダはホスト上の基本的な通信サポートを提供するルーチンのセットです。XTI/TLI はトランスポートプロバイダへのインタフェースであり、プロバイダそのものではありません。図 3–1 を参照してください。

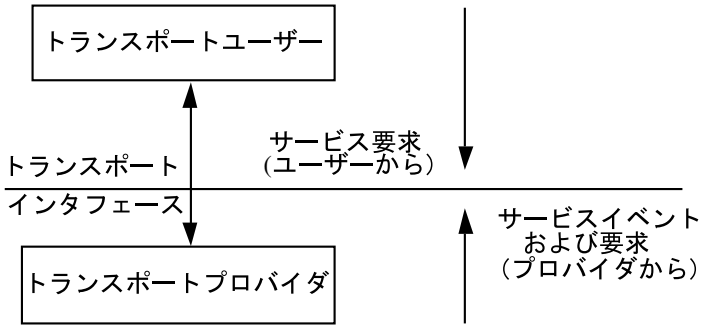

図 3–1　XLI/TLI の仕組み

XLI/TLI コードは第 4 章で説明されている追加インタフェースやメカニズムとともに現在のトランスポートプロバイダとは独立した形で作成できます。SunOS 5 にはいくつかのトランスポートプロバイダ (たとえば、TCP) をオペレーティングシステムの一部として用意しています。トランスポートプロバイダはサービスを実行し、トランスポートユーザーはサービスを要求します。トランスポートユーザーがトランスポートプロバイダへサービス要求を発行します。たとえば、TCP や UDP 上のデータ転送要求があります。

また XTI/TLI はトランスポートに依存しないプログラミングにも使用できます。そのために XTI/TLI には 2 つのコンポーネントが用意されています。

- トランスポートの選択、名前からアドレスへの変換 (name-to-address) を始めとするトランスポートサービスを実行するライブラリルーチン。ネットワークサービスライブラリには ユーザー処理で XTI/TLI を実装する関数セットが用意されています。第 4 章を参照してください。

  TLI を使用したプログラムは以下の方法で、ネットワークサービスライブラリ `libnsl` とリンクを行う必要があります。

```
% cc prog.c -lnsl
```

- 状態遷移規則は、トランスポートルーチンを呼び出すシーケンス （一連の流れ）を定義します。状態遷移規則の詳細については123ページの「状態遷移」 を参照

してください。状態テーブルは状態およびイベントの処理に基づいて、ライブラリ呼び出しの正当なシーケンスの定義を行います。これらのイベントには、ユーザー生成ライブラリ呼び出し、プロバイダ生成イベントのインジケータが含まれます。XLI/TLI のプログラマはインタフェースを使用する前にすべての状態遷移をよく理解しておく必要があります。

XLI/TLI には 2 つのサービスモードが存在します。コネクションモードとコネクションレスモードです。以下のセクションではこれらの 2 つのモードの概要について説明します。

コネクションレスモード

コネクションレスモードはメッセージ指向のモードです。データはユニット間の関係を持たない自己内包型ユニットで転送されます。このサービスはデータの特性を判定する対等ユーザー間の確立された関連付けのみを必要とします。メッセージを送る上で必要な情報のすべては (たとえば、宛先アドレス) 送信されるデータとともに 1 回のサービス要求でトランスポートプロバイダに渡されます。各メッセージは完全に自己内包型です。以下のアプリケーションにはコネクションレスモードサービスが適しています。

- 短期要求対話または短期応答対話
- 動的に再構成が可能な場合
- データの順次転送を必要としない場合

コネクションレストランスポートは信頼性が低いことを理解してください。メッセージのシーケンスを維持しないため、メッセージが行方不明になる場合があります。

コネクションレスモードルーチン

コネクションレスモードトランスポートサービスには 2 つのフェーズ (処理段階) があります。ローカル管理とデータ転送です。ローカル管理フェーズはコネクションモードサービスにおけるのと同じローカル処理の定義を行います。

データ転送フェーズは指定された対等ユーザーへデータユニット (通常データグラムと呼ばれる) の転送を可能にします。各データユニットは宛先ユーザーの転送アドレスとともに転送されます。`t_sndudata(3NSL)` は送信を行い `t_rcvudata(3NSL)`

はメッセージの受信を行います。表 3–1 はコネクションレスモードデータ転送のすべてのルーチンの一覧です。

表 3–1 コネクションレスモードデータ転送のルーチン

コマンド	説明
t_sndudata	トランスポートの他ユーザーへメッセージを送信する。
t_rcvudata	トランスポートの他ユーザーからのメッセージを受信する。
t_rcvuderr	最後に送ったメッセージのエラー情報を検出する。

コネクションレスモードサービス

コネクションレスモードサービスはトランザクション処理アプリケーションなどの短期の要求および応答対話に適しています。データは自己内包型ユニットで転送され、複数のユニット間の論理的な関連は必要ありません。

エンドポイントの初期化

トランスポートのユーザーはデータ転送を行う前に XTI/TLI エンドポイントの初期化を行う必要があります。まず t_open(3NSL) を使用して適切なコネクションレスサービスプロバイダを選択し、t_bind(3NSL) を使用してその識別アドレスを確定します。

プロトコルオプションのネゴシエーションには t_optmgmt(3NSL) を使用します。コネクションモードサービス同様、各トランスポートプロバイダがサポートされているオプションを任意に指定します。オプションのネゴシエーションはプロトコル固有の処理です。例 3–1 はサーバーが着信クエリーの待ち状態から、その後、処理、応答を行う例です。例ではサーバーの定義および初期化シーケンスのコードも見ることができます。

例 3–1 CLTS サーバー

```
#include <stdio.h>
#include <fcntl.h>
```

```
#include <xti.h> /* TLI アプリケーションは <tiuser.h> を使用*/
#define SRV_ADDR 2 /* サーバーの既知アドレス*/

main()
{
   int fd;
   int flags;
   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;
   extern int t_errno;

 if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *) NULL))
         == -1) {
      t_error("unable to open /dev/exmp");
      exit(1);
   }
  if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR))
         == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->addr.len = sizeof(int);
   *(int *)bind->addr.buf = SRV_ADDR;
   bind->qlen = 0;
   if (t_bind(fd, bind, bind) == -1) {
      t_error("t_bind failed");
      exit(3);
   }
   /*
    * TLI インタフェースアプリケーションでは以下のコードが必要です。
    * XTI アプリケーションでは必要ありません。
    *-------------------------------------------------------------
    * バインドされたアドレスの検査
    *
    * if (bind -> addr.len != sizeof(int) ||
    *       *(int *)bind->addr.buf != SRV_ADDR) {
    * fprintf(stderr, "t_bind bound wrong address\n");
    * exit(4);
    * }
    * ---------------------------------------
    */
```

サーバーは t_open(3NSL) を使用して、任意のトランスポートプロバイダとのトランスポートエンドポイントを確立します。各プロバイダには関連付けられたサービスタイプがあるため、ユーザーは適切なトランスポートプロバイダファイルを開くことにより特定のサービスを選択できます。このコネクションレスモードサーバーは 3 つ目の引数を NULL に設定すると t_open(3NSL) により返されるプロバイダの特性を無視します。トランザクションサーバーはトランスポートプロバイダが以下の特性を持っていると判断します。

- トランスポートアドレスは整数値であり各ユーザー固有のものである。
- トランスポートプロバイダは T_CLTS サービスタイプをサポートしている (コネクションレストランスポートサービスまたはデータグラム)。

■ トランスポートプロバイダはプロトコル固有オプションを必要としない。

コネクションレスサーバーはアクセスの可能性のあるすべてのクライアントがサーバーにアクセスできるようエンドポイントにトランスポートアドレスをバインドします。t_alloc(3NSL) により t_bind 構造体が割り当てられ、アドレスの buf および len フィールドが設定されます。

コネクションモードサーバーとコネクションレスモードサーバーの違いの 1 つは、コネクションレスモードサービスの場合は t_bind 構造体の qlen フィールドが 0 であるという点です。待機状態の接続要求が存在しないということです。

XLI/TLI インタフェースはコネクションモードサービス内でトランスポートコネクションの確立を行いながら 2 ユーザー間において固有のクライアントサーバー関係を定義します。このような関係はコネクションレスモードサービスでは存在しません。

TLI では t_bind(3NSL) より返されるバインドされたアドレスが指定されているアドレスと同一であることを保証するため、サーバーによるアドレスのチェックを必要としています。また要求されたアドレスがビジーの場合、t_bind(3NSL) はエンドポイントに、個別の使用されていないアドレスをバインドすることが可能です。

データ転送

ユーザがトランスポートエンドポイントにアドレスをバインドさせた後、データグラムはエンドポイントを介し送受信が可能になります。各送信メッセージは宛先ユーザーのアドレスとともに送信されます。また XTI/TLI はデータユニット転送へのプロトコルオプションの指定を可能にします (たとえば、transit delay)。各トランスポートプロバイダはオプションセットをデータグラム上で定義します。データグラムが宛先ユーザーに渡された時点で、関連付けられたプロトコルオプションも渡されます。

例 3–2 は、コネクションレスモードサーバーのデータ転送フェーズの例です。

例 3–2 データ転送ルーチン

```
if ((ud = (struct t_unitdata *) t_alloc(fd, T_UNITDATA,T_ALL))
        == (struct t_unitdata *) NULL) {
     t_error("t_alloc of t_unitdata struct failed");
     exit(5);
  }
  if ((uderr = (struct t_uderr *) t_alloc(fd, T_UDERROR, T_ALL))
        == (struct t_uderr *) NULL) {
     t_error("t_alloc of t_uderr struct failed");
     exit(6);
  }
```

```
   while(1) {
      if (t_rcvudata(fd, ud, &flags) == -1) {
         if (t_errno == TLOOK) {
               /* 前に送られたデータグラムのエラー*/
               if(t_rcvuderr(fd, uderr) == -1) {
                  exit(7);
               }
            fprintf(stderr, "bad datagram, error=%d\n",
               uderr->error);
            continue;
         }
         t_error("t_rcvudata failed");
         exit(8);
      }
      /*
       * Query() が要求の処理を行い、応答を ud->udata.buf に格納、
       * ud->udata.len が設定される。
       */
      query(ud);
      if (t_sndudata(fd, ud) == -1) {
         t_error("t_sndudata failed");
         exit(9);
      }
   }
}

/* 引数の使用 */
void
query(ud)
struct t_unitdate *ud;
{
   /* 簡略化のため関数の切り口のみ */
}
```

データグラムのバッファー化を行うにはサーバーはまず以下の形式を持つ t_unitdata 構造体の割り当てを行う必要があります。

```
struct t_unitdata {
  struct netbuf addr;
  struct netbuf opt;
  struct netbuf udata;
}
```

addr は入力されるデータグラムの送信元アドレスと、出力されるデータグラムの宛先アドレスを保持します。opt はデータグラムのプロトコルオプションを保持します (プロトコルオプションがある場合)。udata はデータを保持します。addr、opt、および udata フィールドはいかなる入力値にも対応できるよう、十分なバッファーを割り当てておく必要があります。 これを確実に行うため、t_alloc(3NSL) の T_ALL 引数は各 netbuf 構造体の maxlen フィールドを必要に応じ設定します。この例ではプロバイダはプロトコルオプションをサポートしていないため、opt の netbuf 構造体の maxlen は 0 に設定されています。またサーバーはデータグラムエラー用に t_uderr 構造体の割り当てを行います。

トランザクションサーバーは無限ループ状態でクエリーの受信、処理およびクライアントへの応答を行います。最初に次のクエリーを受信するため t_rcvudata(3NSL) を呼び出します。t_rcvudata(3NSL) はデータグラムが入力されるまでブロックし、クエリーを戻します。

t_rcvudata(3NSL) の 2 番目の引数はデータグラムのバッファーを行うための t_unitdata 構造体を特定しています。

3 番目の引数 flags は整数型変数を指し、t_rcvudata(3NSL) からの戻り値にユーザーの udata バッファーがデータグラムを格納するだけの大きさが確保されていないことを示す T_MORE が設定されることが可能です。

上の事態が発生した場合、t_rcvudata(3NSL) の次の呼び出しは残りのデータグラムを取り戻します。t_alloc(3NSL) が最大サイズのデータグラムを格納するのに十分な大きさの udata バッファーを割り当てるため、このトランザクションサーバーは flags のチェックを行う必要がありません。これは t_rcvudata(3NSL) のみに当てはまり、他の受信プリミティブには適用されません。

データグラムが受信された場合、トランザクションサーバーは要求の処理を行うため、query ルーチンを呼び出します。このルーチンは ud が指す構造体に応答を格納し、ud->udata.len を応答のバイト数に合わせ設定します。t_rcvudata(3NSL) により戻される ud->addr の送信元アドレスは t_sndudata(3NSL) の宛先アドレスです。応答の準備が完了した時点で、t_sndudata(3NSL) が呼び出され、応答はクライアントに送信されます。

データグラムエラー

トランスポートプロバイダが t_sndudata(3NSL) によって送られたデータグラムの処理を行えない場合、ユーザーにユニットデータエラーイベント T_UDERR を戻します。このイベントにはエラーを識別するため、データグラムの宛先アドレスとオプション、プロトコル固有エラー値が含まれます。データグラムのエラーはプロトコル固有 (専用) のものです。

注 - ユニットデータエラーイベントは指定された宛先へのデータグラム転送の成功または失敗を必ずしも示しません。コネクションレスサービスはデータの確実な転送を保証していないことを覚えておいてください。

別のデータグラムの受信を行おうとした場合トランザクションサーバーにはエラーが通知されます。この場合、t_rcvudata(3NSL) は失敗し、t_errno は TLOOK に設定されます。TLOOK が設定されている場合、発生の可能性のあるイベントは

T_UDERR のみであることから、サーバーはイベントの取り出しを行うため t_rcvudata(3NSL) を呼び出します。t_rcvuderr(3NSL) の 2 番目の引数は事前に割り当てられている t_uderr 構造体です。この構造体は t_rcvuderr(3NSL) によって格納され、次の形式です。

```
struct t_uderr {
  struct netbuf addr;
  struct netbuf opt;
  t_scalar_t error;
}
```

addr および opt は不正なデータグラムで指定された宛先アドレスおよびプロトコルオプションを特定し、error はプロトコル固有のエラーコードです。トランザクションサーバーはエラーコードを出力し、処理を継続します。

コネクションモード

コネクションモードは回路型のモードです。データは確立された接続を使用して順次送信されます。またこのモードはデータ転送フェーズ中のアドレス解決および転送を行わない識別処理を提供します。データストリーム型の通信を必要とするアプリケーションにこのサービスを使用してください。コネクションモードトランスポートサービスには 4 つのフェーズが存在します。

- ローカル管理
- 接続確立
- データ転送
- 接続解放

ローカル管理フェーズはトランスポートユーザーとトランスポートプロバイダ間のローカル操作の定義を行います (図 3–2 参照)。たとえば、ユーザーはトランスポートプロバイダと通信チャネルの確立を行う必要があります。トランスポートユーザーとトランスポートプロバイダ間の各チャネルは通信の固有エンドポイントであり、トランスポートエンドポイントと呼ばれます。t_open(3NSL) はコネクションモードサービスの供給およびトランスポートエンドポイントの確立を行うため、ユーザーによる特定のトランスポートプロバイダの選択が可能です。

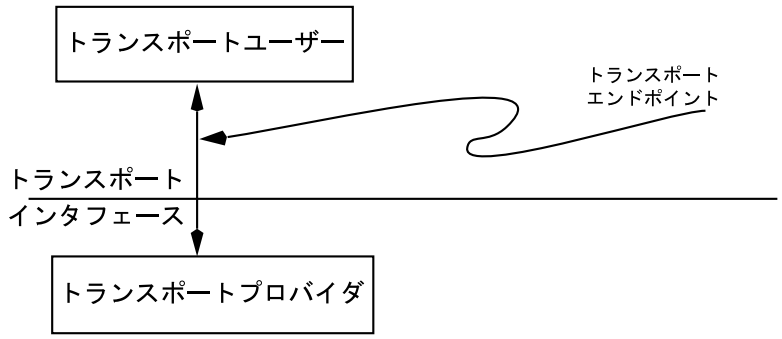

図 3–2　トランスポートエンドポイント

コネクションモードルーチン

各ユーザーは個別にトランスポートプロバイダに認識される必要があります。トランスポートアドレスが各トランスポートエンドポイントと関連付けられます。単一のユーザー処理によって複数のトランスポートエンドポイントの管理が可能です。コネクションモードサービスでは、アドレスを指定することにより、あるユーザーから別のユーザーへの接続の要求が行われます。トランスポートアドレスの構造はトランスポートプロバイダにより定義されます。アドレスは構造化されていない文字列 (たとえば、`file_server`) でも、ネットワーク上のデータの経路指定に必要な情報を含む符号化されたビットパターンのような形式でも指定可能です。各トランスポートプロバイダはユーザー識別のためのメカニズムを独自に定義します。アドレスは `t_bind(3NSL)` によりトランスポートのエンドポイントに割り当てることが可能です。

また、`t_open(3NSL)` および `t_bind(3NSL)` に加え、いくつかのルーチンがローカル操作をサポートしています。表 3–2 は XTI/TLI のローカル管理ルーチンの一覧表です。

表 3–2　XLI/TLI のエンドポイント上の操作ルーチン

コマンド	説明
`t_alloc`	XTI/TLI のデータ構造体を割り当てる
`t_bind`	トランスポートアドレスをトランスポートエンドポイントへバインドする
`t_close`	トランスポートエンドポイントを終了させる

表 3–2 XLI/TLI のエンドポイント上の操作ルーチン　続く

コマンド	説明
t_error	XTI/LTI エラーメッセージを出力する
t_free	t_alloc(3NSL) により割り当てられた構造体を解放する
t_getinfo	特定のトランスポートプロバイダに関連付けられたパラメタセットを戻す
t_getprotaddr	エンドポイントに関連付けられたローカルおよび/またはリモートのアドレスを戻す (XTI のみ)
t_getstate	トランスポートエンドポイントの状態を戻す
t_look	トランスポートエンドポイントの現在のイベントを戻す
t_open	選択されたトランスポートプロバイダへ接続されたトランスポートエンドポイントを確立する
t_optmgmt	トランスポートプロバイダを使用したプロトコル個有のオプションでネゴシエーションする
t_sync	トランスポートエンドポイントとトランスポートプロバイダとの同期をとる
t_unbind	トランスポートアドレスをトランスポートエンドポイントからアンバインドする

コネクションフェーズは 2 ユーザー間で接続の確立または仮想回路の作成を可能にします。図 3–3 を参照してください。

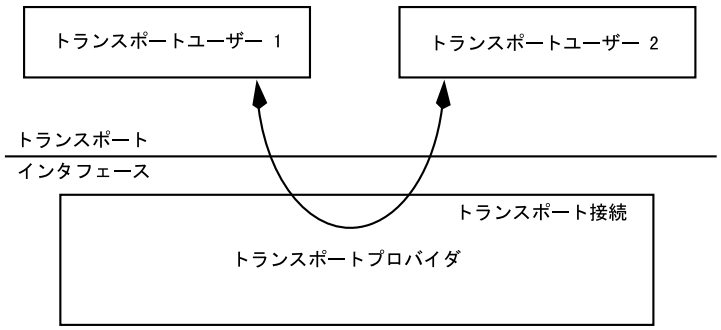

図 3–3　トランスポート接続

たとえば、サーバーがクライアントグループへサービスを通知すると接続フェーズが発生し、要求待ちを行うために t_listen(3NSL) によってブロックします。クライアントは t_connect(3NSL) の呼び出しにより通知されたアドレスでサーバーへの接続を行います。接続要求により t_listen(3NSL) はサーバーへ戻され、接続を完了するため t_accept(3NSL) の呼び出しを行います。

表 3–3 はトランスポート接続を確立するためのルーチンの一覧です。各ルーチンの仕様については、マニュアルページを参照してください。

表 3–3　トランスポート接続を確立するためのルーチン

コマンド	説明
t_accept	トランスポート接続の要求を受け入れる
t_connect	指定された宛先のトランスポートユーザーとの接続を確立する
t_listen	他のトランスポートユーザーからの接続要求を待機する
t_rcvconnect	t_connect(3NSL) が非同期モードで呼び出されている場合、接続の確立を完了する (112ページの「拡張機能」を参照)

データ転送フェーズはユーザーによる接続内の双方向のデータ転送を可能にします。接続を介し t_snd(3NSL) がデータを送信し、t_rcv(3NSL) がデータを受信します。あるユーザーから他のユーザーへ送信されたデータはすべて送信された時の順序で受信されることを前提としています。表 3–4 はコネクションモードデータ転送ルーチンの一覧です。

表 3–4　コネクションモードデータ転送ルーチン

コマンド	説明
t_rcv(3NSL)	トランスポート接続を介し到着したデータの受信
t_snd(3NSL)	トランスポート接続を介してデータを送信

XTI/TLI の接続解放には 2 通りあります。放棄型解放はトランスポートプロバイダへただちに接続を解放するよう指示を与えます。他のユーザーに送られる予定で、まだ送信されていないデータはトランスポートプロバイダによる破棄が可能です。t_snddis(3NSL) は放棄型解放の初期化を行います。t_rcvdis(3NSL) が放棄型解放を受信します。トランスポートプロバイダは通常、いくつかの形式の放棄型解放操作をサポートしています。

トランスポートプロバイダの中にはデータの破棄を行わずに接続の終了を行う正常型解放をサポートしているものがあります。t_sndrel(3NSL) および t_rcvrel(3NSL) はこの機能を持っています。表 3–5 は接続解放ルーチンの一覧です。各ルーチンの仕様についてはマニュアルページを参照してください。

表 3–5　接続解放ルーチン

コマンド	説明
t_rcvdis(3NSL)	接続の切断および残っているユーザーデータに対する原因コードを返す
t_rcvrel(3NSL)	正常型解放の接続要求の受信確認
t_snddis(3NSL)	接続の中止または接続要求の拒否
t_sndrel(3NSL)	接続の正常型解放を要求

コネクションモードサービス

コネクションモードサービスの基本概念をクライアントプログラムとそのサーバープログラムを使用して説明します。例ではセグメントを使用しています。

例の中ではクライアントがサーバー処理への接続を確立します。サーバーはクライアントへファイルの転送を行います。クライアントはそのファイルを受信し、標準出力へ書き出します。

エンドポイントの初期化

クライアントとサーバーの接続が行われる前に、それぞれが t_open(3NSL) を使用してトランスポートプロバイダ (トランスポートエンドポイント) へのローカル接続を行い、t_bind(3NSL) によりその識別 (またはアドレス) を確立する必要があります。

多くのプロトコルでは XTI/TLI で定義されているサービスのサブセットの実行が可能です。各トランスポートプロバイダは提供するサービスを決定したり、サービスを制限したりする特性を持っています。トランスポート特性を定義しているデータは t_info 構造体で t_open(3NSL) によって返されます。表 3–6 は t_info 構造体のフィールドの一覧です。

表 3–6 t_info 構造体

フィールド	内容
addr	トランスポートアドレスの最大サイズ
options	トランスポートユーザーとトランスポートプロバイダ間を送信可能なプロトコル固有オプションの最大バイト数
tsdu	コネクションモードまたはコネクションレスモードで送信可能な最大メッセージサイズ
etsdu	トランスポート接続によって送信可能な優先データ最大メッセージサイズ
connect	接続の確立時にユーザー間で送信可能なユーザーデータの最大バイト数
discon	接続の放棄型解放時にユーザー間で送信可能なユーザーデータの最大バイト数
servtype	トランスポートプロバイダによりサポートされているサービスのタイプ

XTI/TLI により定義されている 3 つのサービスタイプは以下の通りです。

1. T_COTS — トランスポートプロバイダはコネクションモードサービスをサポートしているが 、正常型解放機能を提供していない。接続の終了は放棄型解放によって行われ、送信されていないデータは破棄される。
2. T_COTS_ORD — トランスポートプロバイダは正常型解放機能を持つコネクションモードサービスを提供する。
3. T_CLTS — トランスポートプロバイダはコネクションレスモードサービスを提供する。

t_open(3NSL) により識別されるトランスポートプロバイダに関連付けが行えるのは 1 つのサービスのみです。

t_open(3NSL) はトランスポートエンドポイントのデフォルトプロバイダ特性を戻します。エンドポイントが開かれると変化する特性もあります。これはネゴシエーションを行ったオプションで発生します (オプションのネゴシエーションについてはこの章の以降のページで説明する)。t_getinfo(3NSL) はトランスポートエンドポイントの現在の特性を戻します。

ユーザーが選択したトランスポートプロバイダとエンドポイントを確立した後に、クライアントおよびサーバーは識別の確定を行う必要があります。これを行うのがトランスポートエンドポイントへトランスポートアドレスのバインドを行う t_bind(3NSL) です。サーバーの場合、このルーチンが接続要求待機にエンドポイントが使用されていることをトランスポートプロバイダへ通知します。

t_optmgmt(3NSL) はローカル管理フェーズ中に使用できます。ユーザーによるトランスポートプロバイダとのプロトコルオプション値のネゴシエーションを可能にします。各トランスポートプロトコルは quality-of-service パラメータなど独自のネゴシエーション可能なプロトコルオプションを定義します。オプションがプロトコル固有のものであるため、特定のプロトコル用に作成されたプログラムだけがこの機能を使用できます。

クライアント

これらの機能の詳細を説明するためにクライアントおよびサーバーのローカル管理の必要条件を例に示します。例 3–3 ではクライアントプログラムで必要とされる定義を行ってから必要なローカル管理を行うステップを示しています。

例 3–3 オープンおよびバインドのクライアント実装

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#define SRV_ADDR 1          /* サーバーのアドレス*/
```

```
main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   extern int t_errno;

   if ((fd = t_open("/dev/exmp", O_RDWR, (struct t_info *),NULL))
         == -1) {
      t_error("t_open failed");
      exit(1);
   }
   if (t_bind(fd, (struct t_bind *) NULL, (struct t_bind *) NULL)
         == -1) {
      t_error("t_bind failed");
      exit(2);
   }
```

t_open(3NSL) の最初の引数はトランスポートプロトコルの識別を行うファイルシステムオブジェクトのパスです。/dev/exmp は汎用通信ベースのトランスポートプロトコルの識別を行う特別なファイル名の例です。2 つ目の引数 O_RDWR は読み書き可能なオープンを指定しています。3 つ目の引数は t_info 構造体を指定し、トランスポートのサービス特性が戻されます。

このデータはプロトコルに依存しないソフトウェアに適しています (131ページの「プロトコルに依存しない処理に関する指針」を参照)。この例では NULL ポインタが渡されています。例 3–3 においては、トランスポートプロバイダは以下の特性を持っている必要があります。

- トランスポートアドレスは各ユーザーを固有に識別する整数の値である。
- 例は正常型解放を使用しているためトランスポートプロバイダは T_COTS_ORD サービスタイプをサポートする。
- トランスポートプロバイダはプロトコル固有オプションを必要としない。

ユーザーが T_COTS_ORD 以外のサービスを必要とする場合、別のトランスポートプロバイダをオープンすることが可能です。T_CLTS サービス要請についての例は 108 ページの「読み取り/書き込み用インタフェース」 で説明しています。

t_open(3NSL) は後に続くすべての XTI/TLI 関数呼び出しで使用されるトランスポートエンドポイントファイルハンドルを戻します。識別子はトランスポートプロトコルファイルをオープンして得られるファイル記述子です (open(2) を参照)。

クライアントはエンドポイントにアドレスを割り当てるため t_bind(3NSL) を呼び出します。t_bind(3NSL) の最初の引数はトランスポートエンドポイントハンドルです。2 つ目の引数はエンドポイントへバインドするアドレスを示す t_bind 構造

体を指定します。3 つ目の引数はプロバイダがバインドしたアドレスを示す t_bind 構造体を指定します。

クライアントのアドレスは多くの場合、他の処理がアクセスを行わないため重要性を持ちません。そのため t_bind(3NSL) への 2 つ目および 3 つ目の引数は NULL です。2 つ目の NULL 引数がユーザー用のアドレスの選択のためトランスポートプロバイダへ指示を行います。

t_open(3NSL) または t_bind(3NSL) が失敗した場合、プログラムは stderr による適切なエラーメッセージを表示するために t_error(3NSL) を呼び出します。整数型の外部変数 t_error(3NSL) はエラー値に割り当てられます。エラー値のセットが tiuser.h に定義されています。

t_error(3NSL) は perror(3C) と類似しています。トランスポート機能エラーがシステムエラーの場合、t_errno(3NSL) は TSYSERR に設定され、errno は適切な値に設定されます。

サーバー

サーバーの例においても接続要求待機を行うためにトランスポートエンドポイントを確立する必要があります。 例 3–4 では定義とローカル管理を行うステップを例で示しています。

例 3–4 オープンおよびバインドのサーバー実装

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT −1
#define SRV_ADDR 1          /* サーバーのアドレス*/
int conn_fd;   /* ここで接続の確立*/
extern int t_errno;

main()
{
   int listen_fd;       /* トランスポートエンドポイント待機*/
   struct t_bind *bind;
   struct t_call *call;

   if ((listen_fd = t_open("/dev/exmp", O_RDWR,
      (struct t_info *) NULL)) == −1) {
      t_error("t_open failed for listen_fd");
      exit(1);
   }
   if ((bind = (struct t_bind *)t_alloc( listen_fd, T_BIND, T_ALL))
         == (struct t_bind *) NULL) {
```

```
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;

    /*
     * プロバイダのアドレスの形式を推測するため
     * このプログラムはトランスポートに依存する
     */
     bind->addr.len = sizeof(int);
    *(int *) bind->addr.buf = SRV_ADDR;
    if (t_bind (listen_fd, bind, bind) < 0 ) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }

    #if (!defined(_XOPEN_SOURCE) ||(_XOPEN_SOURCE_EXTENDED −0 != 1))
    /*
     * 正しいアドレスがバインドされているかどうか
     *
     * XTI の場合このテストは不要
     */

    if (bind->addr.len != sizeof(int) ||
        *(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
     }
     #endif
```

クライアント同様、サーバーはまず選択したトランスポートプロバイダとトランスポートエンドポイントを確立するため t_open(3NSL) を呼び出します。エンドポイント listen_fd は接続要求待機を行うために使用されます。

次にサーバーはアドレスをエンドポイントへバインドします。アドレスは各クライアントがサーバーへアクセスする際に使用されます。2 つ目の引数はエンドポイントへバインドするアドレスを指定する t_bind 構造体を指します。t_bind 構造体は以下の形式です。

```
struct t_bind {
  struct netbuf addr;
  unsigned qlen;
}
```

addr はバインドされたアドレスを示し、qlen は未処理の接続要求の最大件数を指定します。すべての XTI 構造および定数定義は xti.h を介しアプリケーションプログラムで使用可能になります。すべての TLI 構造体および定数定義は tiuser.h に格納されます。

アドレスは以下の形式で netbuf 構造体で指定されます。

```
struct netbuf {
  unsigned int maxlen;
```

```
  unsigned int len;
  char *buf;
}
```

maxlen はバッファーの最大長をバイト単位で指定、len はバッファー内のデータのバイト長を指定、そして buf はデータを格納しているバッファーを指します。

t_bind 構造体では、データはトランスポートアドレスを識別します。qlen は待機可能な接続要求の最大数を指定します。qlen の値が正の場合、エンドポイントを接続要求の待機に使用することが可能となります。t_bind(3NSL) は、ただちにバイトされたアドレスに各接続要求の待機を行うようトランスポートプロバイダに指示します。サーバーは 1 つずつ接続要求の待機を解除し、受け付けまたは拒否を行う必要があります。次の接続要求を受信する前に 1 つの接続要求の処理および応答を行うサーバーの場合、qlen の値は 1 が適切です。応答を行う前に複数の接続要求の待機を解除するサーバーの場合は、より長い待ち行列を指定する必要があります。この例のサーバーでは、一度に 1 つの接続要求の処理しか行わないため、qlen は 1 に設定してあります。

t_alloc(3NSL) は t_bind 構造体を割り当てるために呼び出されます。t_alloc(3NSL) には 3 つの引数があります。トランスポートエンドポイントのファイル記述子、割り当てる構造体の識別子、そして割り当てる netbuf バッファーを指定するフラグです (netbuf バッファーを使用する場合)。T_ALL はすべての netbuf バッファーの割り当てを指定し、この例では addr バッファーが割り当てられる要因となります。バッファーのサイズは自動的に決定され、maxlen に格納されます。

各トランスポートプロバイダは個別にアドレス空間を管理します。トランスポートプロバイダには複数のトランスポートエンドポイントに同じトランスポートアドレスをバインドするものと、各エンドポイントに固有のアドレスをバインドするものがあります。XTI と TLI のアドレスバインド方法には大きく異なる部分があります。

TLI のルールでは、プロバイダが要求されたアドレスのバインドが可能かを判定します。バインドが行えない場合、そのアドレス空間で別の有効アドレスを捜しトランスポートエンドポイントにバインドします。アプリケーションプログラムは、バインドされたアドレスが事前にクライアントに通知されたものと同一であることをチェックする必要があります。XTI ではプロバイダが要求されたアドレスのバインドを行えないと判定した場合、t_bind(3NSL) をエラーで終了します。

t_bind(3NSL) が成功した場合、プロバイダは接続要求の待機を開始し、通信の次のフェーズに移ります。

接続の確立

このフェーズでは XTI/TLI は異なる処理をクライアントおよびサーバーに要求します。クライアント t_connect(3NSL) を使用して指定されたサーバーに接続要求を行うことにより接続の確立を開始します。 サーバーはクライアントの要求を t_listen(3NSL) を呼び出して受信します。サーバーはクライアントの要求を受け付け、または拒否しなければなりません。接続を確立するため t_accept(3NSL) を呼び出すか、または t_snddis(3NSL) を呼び出し、要求を拒否します。クライアントは t_connect(3NSL) が返されることにより結果を認識をします。

TLI は接続の確立時に、すべてのトランスポートプロバイダにはサポートされていない可能性のある 2 つの機能をサポートしています。

- 接続の確立を行っている間のクライアントとサーバーのデータ転送。クライアントは接続要求時、サーバーへデータを送信できます。このデータは t_listen(3NSL) によってサーバーに渡されます。サーバーは接続要求の受け付け、または拒否を行う時にクライアントにデータを送信することが可能です。t_open(3NSL) により返される接続特性が 2 ユーザー間で (データの転送が可能な場合) 転送可能なデータのサイズを決定します。
- プロトコルオプションのネゴシエーション。クライアントはトランスポートプロバイダおよび (または) リモートユーザーへ任意のプロトコルオプションを指定できます。XTI/TLI はローカルおよびリモート両方のオプションネゴシエーションをサポートします。オプションネゴシエーションはプロトコル固有の機能です。

これらの機能はプロトコルに依存するソフトウェアを作成します (131ページの「プロトコルに依存しない処理に関する指針」を参照)。

クライアント

例 3–5 はクライアントの接続確立を行うためのコード例です。

例 3–5 クライアントからサーバーへの接続

```
if ((sndcall = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR))
      == (struct t_call *) NULL) {
   t_error("t_alloc failed");
   exit(3);
}

/*
 * プロバイダのアドレスの形式を認知していると推測されるため、
 * このプログラムはトランスポートに依存します。
 */
sndcall->addr.len = sizeof(int);
*(int *) sndcall->addr.buf = SRV_ADDR;
```

```
if (t_connect( fd, sndcall, (struct t_call *) NULL) == -1 ) {
   t_error("t_connect failed for fd");
   exit(4);
}
```

t_connect(3NSL) 呼び出しはサーバに接続するために使用されます。t_connect(3NSL) の 1 つ目の引数は、クライアント側のエンドポイントを特定します。2 つ目の引数は、宛先サーバーを特定する t_call 構造体を指します。この構造体は以下の形式です。

```
struct t_call {
  struct netbuf addr;
  struct netbuf opt;
  struct netbuf udata;
  int sequence;
}
```

addr はサーバーのアドレスを特定します。opt は接続へプロトコル固有オプションを指定します。udata はサーバーへの接続要求とともに送信可能なユーザーデータを特定します。sequence フィールドの t_connect(3NSL) における役割はありません。コーディング例ではサーバーのアドレスのみが渡されます。

t_alloc(3NSL) は t_call 構造体を動的に割り当てます。t_alloc(3NSL) の 3 つ目の引数、T_ADDR は netbuf バッファーの割り当てがシステムに必要であることを示します。サーバーのアドレスは buf にコピーされ、len は同様に適切な値に設定されます。

t_connect(3NSL) の 3 つ目の引数は、新たに確立された接続の情報を取り出すのに使用でき、サーバーによって接続要求の応答とともに送られたユーザーデータの返送が可能です。ここでは 3 つ目の引数はクライアントにより NULL に設定されています。接続は t_connect(3NSL) の応答が成功している場合に確立されます。サーバーが接続要求を拒否した場合、t_connect(3NSL) は t_errno を TLOOK に設定します。

イベント処理

TLOOK エラーには特殊な性質があります。XTI/TLI ルーチンにエンドポイント上の予測されていない非同期トランスポートイベントによる割り込みが発生した場合 TLOOK が設定されます。この場合 TLOOK は XTI/TLI ルーチンについてのエラーのレポートを行わず、また保留イベントのため通常のルーチン処理が行われません。表 3–7 は XTI/TLI で定義されているイベントの一覧です。

表 3–7　非同期エンドポイントイベント

名称	説明
T_LISTEN	接続要求がトランスポートエンドポイントに着信
T_CONNECT	前の接続要求の確認着信 (サーバーが接続要求を受け付けた場合に生成)
T_DATA	ユーザーデータ着信
T_EXDATA	優先ユーザーデータ着信
T_DISCONNECT	接続の中止または接続要求の拒否の着信の通知
T_ORDERL	接続の正常型解放要求の着信
T_UDERR	最後に着信したデータグラム内のエラー通知 (108ページの「読み取り/書き込み用インタフェース」を参照)

123ページの「状態遷移」内の状態テーブルでは、各状態で発生する可能性のあるイベントを表にしています。t_look(3NSL) は TLOOK エラーが発生した場合にユーザーによるイベントの判定を可能にします。例の中では、接続要求が拒否された場合、クライアントは終了します。

サーバー

クライアントが t_connect(3NSL) を呼び出した場合、サーバーのトランスポートエンドポイントへ接続要求が送られます。サーバーは各クライアントの接続要求を受け付け、接続のサービスを提供するため処理生成を行います。

```
if ((call = (struct t_call *) t_alloc(listen_fd, T_CALL, T_ALL))
      == (struct t_call *) NULL) {
   t_error("t_alloc of t_call structure failed");
   exit(5);
}
while(1) {
   if (t_listen( listen_fd, call) == -1) {
      t_error("t_listen failed for listen_fd");
      exit(6);
   }
   if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
      run_server(listen_fd);
}
```

サーバーは t_call 構造を割り当て、閉ループを行います。ループは接続要求のため t_listen(3NSL) でブロックします。要求が着信した時点で、サーバーは接続要求を受け付けるため accept_call() を呼び出します。accept_call() は接続を代替トランスポートエンドポイントで (下記で説明されている方法で) 受け付け、エンドポイントのハンドルを返します (*conn_fd* はグローバル変数)。接続が代替エンドポイントによって受け付けられるため、サーバーは引き続き元のエンドポイントの待機を行うことが可能です。呼び出しがエラーなしに受け付けられた場合、run_server が接続のサービスを提供するために起動されます。

XTI/TLI ではこのような処理のブロックを防ぐルーチンのために非同期モードをサポートします (112ページの「拡張機能」を参照)。

接続要求が着信すると、サーバーは例 3–6 で示されるようにクライアントの要求を受け付けるため accept_call() を呼び出します。

注 - このサーバーは一度に 1 つの接続要求の処理しか行う必要がないことを暗黙の前提としています。これは通常のサーバーではあまりない状況です。複数の接続要求の同時処理を行うために必要なコードは XTI/TLI のイベントメカニズムのため、複雑です (このようなサーバーの場合は、113ページの「高度なプログラム例」を参照)。

例 **3–6** accept_call 関数

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
   int resfd;

   if ((resfd = t_open("/dev/exmp", O_RDWR, (struct t_info *) NULL))
         == -1) {
      t_error("t_open for responding fd failed");
      exit(7);
  }
   if (t_bind(resfd,(struct t_bind *) NULL, (struct t_bind *NULL))
         == -1) {
      t_error("t_bind for responding fd failed");
      exit(8);
   }
   if (t_accept(listen_fd, resfd, call) == -1) {
      if (t_errno == TLOOK) {         /*切断である必要あり*/
         if (t_rcvdis(listen_fd,(struct t_discon *) NULL) == -1) {
            t_error("t_rcvdis failed for listen_fd");
            exit(9);
         }
         if (t_close(resfd) == -1) {
            t_error("t_close failed for responding fd");
            exit(10);
         }
```

```
            /*上に戻り、他の呼び出しの待機*/
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}
```

accept_call() には 2 つの引数があります。

listen_fd	接続要求が着信したトランスポートエンドポイントのファイルハンドル
call	接続要求に関するすべての情報が格納された t_call 構造体を指す

サーバーはトランスポートプロバイダのクローンのデバイス特殊ファイルを開き、アドレスをバインドすることによって、最初に別のトランスポートエンドポイントをオープンします。NULL はプロバイダによってバインドされたアドレスを返さないよう指定します。新しいトランスポートエンドポイント、*resfd* がクライアントの接続要求を受け付けます。

t_accept(3NSL) の最初の 2 つの引数は待機を行うトランスポートエンドポイントおよび接続が受け付けられるエンドポイントをそれぞれ指定します。待機を行うエンドポイント上で接続を受け付けると、接続の間、他のクライアントはサーバーへアクセスすることができません。

t_accept(3NSL) の 3 つ目の引数は、接続要求を格納している t_call 構造体を指します。この構造体は呼び出しを行っているユーザーのアドレス、および t_listen(3NSL) によって戻されたシーケンス番号が格納されています。サーバーが複数の接続要求の待機を行った場合、シーケンス番号は重要です。112ページの「拡張機能」ではこの例を説明しています。t_call 構造体はプロトコルオプションおよびクライアントに渡すユーザーデータも識別します。このトランスポートプロバイダはプロトコルオプションまたは接続中のユーザーデータの転送をサポートしないため、t_listen(3NSL) によって戻された t_call 構造体は変更なしに t_accept(3NSL) へ渡されます。

この例は簡略化されています。サーバーは t_open(3NSL) または t_bind(3NSL) 呼び出しが失敗した時点で終了します。exit(2) は listen_fd のトランスポートエンドポイントを閉じ、クライアントへは切断要求が送信されます。クライアントの t_connect(3NSL) 呼び出しは失敗し、t_errno は TLOOK に設定されます。

t_accept(3NSL) は、待機しているエンドポイント上で接続が受け付けられる前に非同期イベントが発生し、t_errno が TLOOK に設定された場合失敗する可能性があります。この状態では待機を行っている接続要求が 1 つでも、切断要求のみが

送信可能であることが表 3–8 で示されています。このイベントはクライアントが事前に行った接続の要求を取り消した場合に発生する可能性があります。切断要求が着信した場合、サーバーは t_rcvdis(3NSL) を呼び出し応答を行う必要があります。このルーチンの引数は、切断要求のデータを取り出すために使用される t_discon 構造体へのポインタです。この例ではサーバーは NULL を渡します。

切断要求を受け取った後、accept_call は応答を行うトランスポートエンドポイントを閉じ、 DISCONNECT を戻してサーバーに接続がクライアントによって切断されたことを伝えます。その後サーバーは他の接続要求の待機を行います。

図 3–4 はサーバーの接続確立の方法を示します。

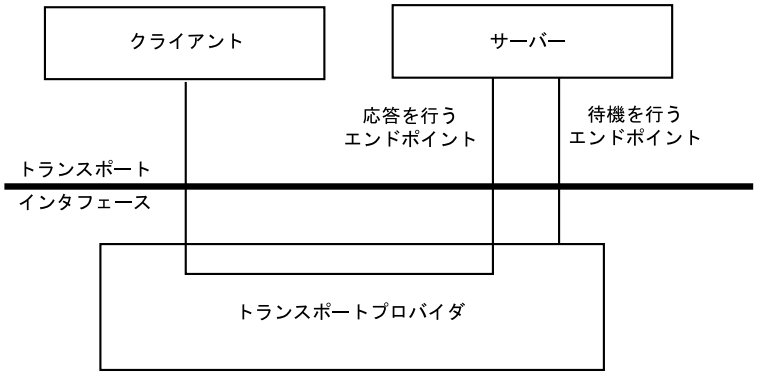

図 3–4　トランスポートエンドポイントの待機と応答

トランスポートの接続は新しく応答を行うエンドポイント上で確立され、待機を行ったエンドポイントは他の接続要求の待機を行うため解放されます。

データ転送

接続が確立された後、サーバーおよびクライアントは t_snd(3NSL) および t_rcv(3NSL) を使用してデータの転送が行えるようになります。XTI/TLI はこれ以降、クライアントおよびサーバーを区別なく扱います。どちらのユーザーもデータの送信、受信、および接続の解放が行えます。

トランスポート接続上のデータのクラスには以下の 2 種類があります。

1. 普通データ (Normal data)
2. 優先データ (Expedited data)

優先データは緊急度の高いデータに使用されます。優先データの正確な意味論はトランスポートプロバイダにより異なります。すべてのトランスポートプロトコルが優先データをサポートしているわけではありません (t_open(3NSL) を参照)。

ほとんどのコネクション型モードプロトコルはバイトストリームによってデータの転送を行います。「バイトストリーム」では接続上で送信されるデータにメッセージ境界を与えていません。トランスポートプロトコルによってはトランスポート接続上でメッセージ境界を持っているものもあります。このサービスは XTI/TLI によってサポートされていますが、プロトコルに依存しないソフトウェアで使用することは避けてください。

メッセージ境界は t_snd(3NSL) および t_rcv(3NSL) の T_MORE フラグによって発行されます。トランスポートサービスデータユニット (TSDU) と呼ばれるメッセージは、2 つのトランスポートユーザー間を個別のユニットとして転送を行うことが可能です。メッセージの最大サイズは基本のトランスポートプロトコルにより定義されます。メッセージサイズは t_open(3NSL) または t_getinfo(3NSL) で得られます。

メッセージは複数のユニットで送信できます。そのためには (複数のユニットの) 最後のメッセージ以外のすべての t_snd(3NSL) 呼び出しに T_MORE フラグを設定します。フラグは現在と次の t_snd(3NSL) 呼び出しは論理ユニットであることを指定します。最後のメッセージで T_MORE フラグを外すことにより論理ユニットの終わりを指定します。

同様に論理ユニットを複数のユニットで送信することも可能です。もし t_rcvv(3NSL) が T_MORE フラグが設定された状態で戻された場合、ユーザーはメッセージの続きを受信するため再び t_rcvv(3NSL) を呼び出す必要があります。ユニットの最後のメッセージは T_MORE が設定されていない t_rcvv(3NSL) の呼び出しによって認識することが可能です。

T_MORE フラグは XTI/TLI によるデータのパッケージ方法および遠隔ユーザーへのデータの配信方法については指定しません。各トランスポートプロトコルおよび各プロトコルの実装方法によりデータのパッケージおよび配信を異なる方法で行えます。

たとえば、ユーザーが t_snd(3NSL) への 1 回の呼び出しで完全なメッセージを送った場合、トランスポートプロバイダが受信ユーザーへ 1 つのユニットでデータの配信を行う保証はありません。同様に、2 つのユニットにより送信されたメッセージは遠隔トランスポートユーザーへ 1 つのユニットとして配信される可能性があります。

トランスポートによりサポートされている場合、メッセージ境界は T_MORE の値を t_snd(3NSL) に設定し、t_rcvv(3NSL) の後にテストを行うことでのみ保持され

ます。これにより受信ユーザーは送られたときと同一の内容とメッセージ境界のメッセージを見ることが保証されます。

クライアント

例のサーバーはクライアントへトランスポート接続を使用しログファイルを転送しています。クライアントはそのデータを受信し、標準出力ファイルへ書き込みます。クライアントとサーバーではメッセージ境界を持たないバイトストリームインタフェースが使用されています。クライアントは以下によりデータを受信します。

```
while ((nbytes = t_rcv(fd, buf, nbytes, &flags))!= -1){
   if (fwrite(buf, 1, nbytes, stdout) == -1) {
      fprintf(stderr, "fwrite failed\n");
      exit(5);
   }
}
```

クライアントは受信データを受け取るため繰り返し t_rcvv(3NSL) を呼び出します。t_rcvv(3NSL) はデータが着信するまでブロックします。t_rcvv(3NSL) はデータの量に合わせ *nbytes* のデータを *buf* に格納しバッファーに書き込んだバイト数を戻します。クライアントは標準出力ファイルへデータを書き込み、処理を継続します。データ転送ループは t_rcvv(3NSL) が失敗した段階で終了します。t_rcvv(3NSL) は正常型解放または切断要求が着信した時点で失敗します。何らかの理由で fwrite(3C) が失敗した場合、クライアントは終了し、トランスポートエンドポイントは閉じられます。トランスポートエンドポイントがデータ転送中に閉じられた場合 (exit(2) または t_close(3NSL) によって)、接続は中止され遠隔ユーザーは接続の切断要求を受信します。

サーバー

サーバーは子プロセスを生成しクライアントへデータを転送することによりデータ転送を管理します。親プロセスは接続要求の待機を行うためループを継続します。例 3–7 で示すとおり、サーバーでは子プロセスの生成を行うため run_server が呼び出されます。

例 3–7 ループバックおよび待機を行うための子プロセスの生成

```
connrelease()
{
   /*ここで必要なため conn_fd  はグローバル*/
   if (t_look(conn_fd) == T_DISCONNECT) {
      fprintf(stderr, ``connection aborted\n'');
      exit(12);
   }
```

```
   /*上記以外の場合、正常型解放を要求-通常終了*/
   exit(0);
}
run_server(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp;                        /*ログファイルへのファイルポインタ*/
   char buf[1024];

   switch(fork()) {
   case −1:
      perror("fork failed");
      exit(20);
   default:          /*親*/
      /* conn_fd を閉じ、戻って再び待機する*/
      if (t_close(conn_fd) == −1) {
         t_error("t_close failed for conn_fd");
         exit(21);
      }
      return;
   case 0:                             /*子*/
      /* listen_fd を閉じ、サービスを行う */
      if (t_close(listen_fd) == −1) {
         t_error("t_close failed for listen_fd");
         exit(22);
      }
      if ((logfp = fopen("logfile", "r")) == (FILE *) NULL) {
         perror("cannot open logfile");
         exit(23);
      }
      signal(SIGPOLL, connrelease);
      if (ioctl(conn_fd, I_SETSIG, S_INPUT) == −1) {
         perror("ioctl I_SETSIG failed");
         exit(24);
      }
      if (t_look(conn_fd) != 0){       /*切断するかどうか*/
         fprintf(stderr, "t_look: unexpected event\n");
         exit(25);
      }
      while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
         if (t_snd(conn_fd, buf, nbytes, 0) == −1) {
            t_error("t_snd failed");
            exit(26);
         }
```

フォーク後、親プロセスは待機のメインループへ戻ります。子プロセスは新たに確立されたトランスポート接続を管理します。フォークが失敗した場合、`exit(2)` が両方のトランスポートエンドポイントを閉じ、クライアントに接続の切断要求を送り、クライアントの `t_connect(3NSL)` 呼び出しは失敗します。

サーバーの処理はログファイルから一度に 1024 バイトを読み込み、クライアントに `t_snd(3NSL)` を使用して送ります。*buf* はデータバッファーの開始点を指し、*nbytes* は送信するデータのバイト数を指定します。4 つ目の引数には、0 または以下の 2 つのオプションフラグを指定することが可能です。

- T_EXPEDITED - データが優先であることを指定
- T_MORE - このブロックのメッセージが次のブロックによって継続されることを指定

この例ではどちらのフラグもサーバーによって設定されていません。

ユーザーがトランスポートプロバイダをデータであふれさせた場合、トランスポートから十分なデータが取り除かれるまで t_snd(3NSL) がブロックを行います。

t_snd(3NSL) は接続の切断要求を捜しません (接続が切断したことを表示)。接続が中止された場合、データが失われる可能性があるためサーバーへ通知する必要があります。1 つの解決法は、各 t_snd(3NSL) 呼び出しの前、または t_snd(3NSL) 失敗の後に、着信するイベントのチェックを行うよう t_look(3NSL) を呼び出すことです。例ではより整理された手法を使用しています。ioctl(2) の I_SETSIG によって指定されているイベントが発生した場合にユーザー要求をシグナルにします。 streamio(7I) のマニュアルページを参照してください。S_INPUT によりエンドポイント conn_fd に入力が着信した場合にシグナルをユーザープロセスへ送信します。接続の切断要求が着信した場合、シグナルを検知するルーチン (connrelease) がエラーメッセージを出力し、終了します。

サーバーが t_snd(3NSL) および t_rcv(3NSL) 呼び出しを交互に行う場合、着信する接続の切断要求を認識するために、t_rcv(3NSL) を使用することが可能です。

接続の解放

データ転送中のいかなる場合でもいずれかのユーザーがトランスポート接続を解放し会話を終了させることが可能です。

- 最初の方法の放棄型解放は、ただちに接続を切断し、宛先ユーザーへ配信されていないデータを破棄します。

  どちらのユーザーも放棄型解放を行うために t_snddis(3NSL) を呼び出すことが可能です。XTI/TLI で問題が発生した場合は、トランスポートプロバイダによる接続の中止も可能です。t_snddis(3NSL) は接続の中止時に、ユーザーから遠隔ユーザーへのデータの送信を可能としています。放棄型解放はすべてのトランスポートプロバイダによりサポートされています。接続の中止時にデータを送信する機能は、すべてのトランスポートプロバイダではサポートされていません。

  遠隔ユーザーが接続中止を通知された段階で、切断要求を受けるためには t_rcvdis(3NSL) を呼び出します。呼び出しにより接続が切断された原因を識別するコードおよび、切断要求とともに送られたデータを戻します (切断要求が遠

隔ユーザーによって行われた場合) 。原因コードが基本的なプロトコル固有のものであり、プロトコルに依存しないソフトウェアによる判断は行うべきではありません。

- 2 つ目の方法の正常型解放では、データが失われないよう接続を終了させます。すべてのプロバイダは放棄型解放をサポートしていなければなりませんが、正常型解放はすべてのコネクション型プロトコルにサポートされているオプションではありません。

正常型解放をサポートするトランスポートの選択については 140ページの「トランスポート選択」を参照してください。

サーバー

この例は、トランスポートプロバイダが正常型解放をサポートしていることを前提としています。すべてのデータがサーバーによって送信された時点で接続は以下のように解放されます。

```
if (t_sndrel(conn_fd) == -1) {
   t_error(``t_sndrel failed'');
   exit(27);
}
pause(); /*正常型解放の要求が着信するまで*/
```

正常型解放は各ユーザーによる 2 段階の処理を必要とします。サーバーは t_sndrel(3NSL) を呼び出すことが可能です。このルーチンは接続の切断要求を送ります。クライアントが要求を受信した場合、引き続きデータをサーバーに送ることが可能です。すべてのデータが送られた時点で、クライアントは t_sndrel(3NSL) を呼び出し接続の切断要求を送り返します。接続は両方のユーザーが切り離し要求を受け取ってから解放されます。

この例で、データはサーバーからクライアントへのみ送信されます。そのためサーバーが解放に着手したあとにクライアントからデータを受信する場合の決まりはありません。サーバーは解放に着手した後、pause(2) を呼び出します。

クライアントは正常型解放により応答し、connrelease() により検知されるシグナルを生成します。(例 3–7 で、サーバーは ioctl(2) の I_SETSIG を受信イベントに対しシグナルを生成するために使用)。この状態で発生する可能性のある XTI/TLI イベントは切断要求、または正常型解放のため、connrelease は正常型解放要求が着信した時点で通常通り終了します。connrelease の exit(2) がトランスポートエンドポイントを閉じ、バインドされたアドレスの解放を行います。終了を行わずにトランスポートエンドポイントを閉じる場合、t_close(3NSL) を呼び出します。

クライアント

クライアントは、サーバーが接続を解放するのと同様に解放を行います。クライアントは t_rcv(3NSL) が失敗するまで受信データの処理を行います。サーバーが接続を解放した場合 (t_snddis(3NSL) または t_sndrel(3NSL) を使用した場合)、t_rcv(3NSL) は失敗し、 t_errno を TLOOK に設定します。その後クライアントは以下の方法で接続の解放を処理します。

```
if ((t_errno == TLOOK) && (t_look(fd) == T_ORDREL)) {
   if (t_rcvrel(fd) == -1) {
      t_error(``t_rcvrel failed'');
      exit(6);
  }
  if (t_sndrel(fd) == -1) {
      t_error(``t_sndrel failed'');
      exit(7);
  }
  exit(0);
 }
```

クライアントのトランスポートエンドポイント上の各イベントは正常型解放要求のチェックが行われます。正常型解放の要求が見つかった場合、クライアントは要求の処理を行うために t_rcvrel(3NSL) と、解放要求の応答を送るため t_sndrel(3NSL) を呼び出します。その後クライアントは終了し、トランスポートエンドポイントを閉じます。

トランスポートプロバイダが正常型解放をサポートしていない場合、t_snddis(3NSL) と t_rcvdis(3NSL) とともに放棄型解放を使用します。各ユーザーはデータの喪失を防ぐ手段をとる必要があります。たとえば、会話の終わりが判断できるようデータストリーム内で特殊なバイトパターンを使用します。

読み取り/書き込み用インタフェース

ユーザーが接続中に受信したデータの処理を行うために、既存のプログラム上 (たとえば /usr/bin/cat など) で exec(2) を使用してトランスポート接続の確立を行いたい場合があります。既存のプログラムは read(2) および write(2) を使用します。XTI/TLI は直接トランスポートプロバイダへの読み取り/書き込みインタフェースをサポートしていないものの、これを行うことが可能です。インタフェースはトランスポート接続上のデータ転送フェーズ内で read(2) および write(2) 呼び出しを可能としています。このセクションでは XTI/TLI のコネクションモードサービスへの読み取り/書き込みインタフェースについて説明しています。このインタフェースはコネクションレスモードサービスでは使用できません。

読み取り/書き込みインタフェースは 90ページの「コネクションモードサービス」のクライアント例を (変更を加えた形で) 使い説明しています。クライアントはデータ転送フェーズの段階までは同一の処理を行います。そこからクライアントは読み取り/書き込みインタフェースと cat(1) を受信データの処理に使用します。cat(1) はトランスポート接続上で変更なしに実行されます。例 3–3 のクライアントとこのクライアントの相違点のみを例 3–8 に示します。

例 3–8 読み取り/書き込みインタフェース

```
#include <stropts.h>
   .
   ./*
     同一のローカル管理および接続確立手順
     */
   .
   if (ioctl(fd, I_PUSH, "tirdwr") == -1) {
      perror(``I_PUSH of tirdwr failed'');
      exit(5);
  }
   close(0);
   dup(fd);
   execl(``/usr/bin/cat'', ``/usr/bin/cat'', (char *) 0);
   perror(``exec of /usr/bin/cat failed'');
   exit(6);
}
```

クライアントは tirdwr をトランスポートエンドポイントに関連付けられたストリーム内にプッシュすることにより読み取り/書き込みインタフェースを呼び出します。streamio(7I) の I_PUSH を参照してください。tirdwr はトランスポートプロバイダより上位に位置する XTI/TLI を純粋な読み取り/書き込みインタフェースに変換します。モジュールが設置された段階で、クライアントはトランスポートエンドポイントを標準入力ファイルとして確立するために close(2) および dup(2) を呼び出し、入力の処理を行うために /usr/bin/cat を使用します。

トランスポートプロバイダへ tirdwr をプッシュすることにより、XTI/TLI は変更されます。read(2) および write(2) の意味論が使用されなければならず、メッセージ境界の保持は行われなくなります。tirdwr は XTI/TLI 意味論を復元するためにトランスポートプロバイダからポップすることが可能です (streamio(7I) の I_POP を参照)。

注意 - トランスポートエンドポイントがデータ転送フェーズ中にある場合のみ tirdwr モジュールをストリーム上にプッシュすることが可能です。モジュールがプッシュされたあとは、ユーザーは XLI/TLI のルーチンを呼び出すことはできません。XTI/TLI ルーチンが呼び出された場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成し、使用不可であることを通知します。さらに tirdwr モジュールをストリーム上からポップした場合、トランスポート接続は中止されます (streamio(7I) の I_POP を参照)。

書き込み

write(2) を使用しトランスポート接続へデータを送信します。tirdwr はデータをトランスポートプロバイダへ通過させます。メカニズム上は許可されているゼロ長データパケットを送った場合、tirdwr はメッセージを破棄します。トランスポート接続が、たとえば、遠隔ユーザーが接続を t_snddis(3NSL) を使用したため中止された場合、ストリーム上にハングアップ状態が生成され、それ以降の write(2) 呼び出しは失敗し、errno は ENXIO に設定されます。ハングアップ後も入手可能なデータの取り出しは可能です。

読み取り

read(2) を使用し、トランスポート接続にてデータを受信します。tirdwr はデータをトランスポートプロバイダから通過させます。プロバイダからユーザーへ渡されるその他のイベント、または要求は tirdwr によって以下の通り処理されます。

- read(2) はユーザーへ送られる優先データの識別は行えません。優先データ要求を受信した場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成します。エラーはその後に続くシステム呼び出しを失敗に終わらせます。read(2) を 優先データの受信に使用しないでください。
- tirdwr は放棄型の切断要求を破棄し、ストリーム上にハングアップ状態を生成します。後続の read(2) 呼び出しは残りのデータの回収を行い、それ以降の呼び出しに対しては 0 を戻します (ファイルの終わりを伝える)。
- tirdwr は正常型の切断要求を破棄し、ユーザーへゼロ長のメッセージを配信します。read(2) で説明されているのと同様、0 を戻すことによりユーザーにメッセージの終了を通知します。

- その他の XTI/TLI 要求が受信された場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成します。エラーはその後に続くシステム呼び出しを失敗に終わらせます。接続が確立された後にユーザーがストリーム上に tirdwr のプッシュを行った場合、要求は生成されません。

閉じる

ストリーム上に tirdwr が存在する場合、接続が行われている間、トランスポート接続上でデータの送受信が可能です。どちらのユーザーも、トランスポートエンドポイントに関連付けられたファイル記述子を閉じることにより、またはストリーム上から tirdwr モジュールをポップさせることにより接続を終了させることが可能です。いずれの場合においても tirdwr は以下の処理を行います。

- 正常型解放要求が事前に tirdwr により受信されている場合、正常型解放を行うため要求はトランスポートプロバイダに渡されます。正常型解放手続きを発した遠隔ユーザーはデータ転送が完了した時点で要求していた結果を受け取ります。
- tirdwr によって事前に接続の切断要求が受信されている場合、特別な処置は行われません。
- 正常型解放要求も接続の切断要求も事前に tirdwr によって受信されていない場合、接続を中止するためトランスポートプロバイダへ切断要求が渡されます。
- ストリーム上で事前にエラーが発生し、接続の切断要求が tirdwr によって受信されていない場合、トランスポートプロバイダへは切断要求が渡されます。

プロセスは tirdwr がストリーム上にプッシュされてから正常型解放を行うことはできません。tirdwr はトランスポート接続の反対側のユーザーによって正常型解放が発行された場合のみ解放を受け付けます。このセクションのクライアントが 90 ページの「コネクションモードサービス」のサーバープログラムと交信を行った場合、サーバーは正常型解放の要求によりデータの転送を終了します。その後サーバーはクライアントからの次の要求を待ちます。その時点でクライアントは終了し、トランスポートエンドポイントは閉じられます。ファイル記述子が閉じられると、tirdwr は接続のクライアント側からの正常解放型要求を発行します。これによりサーバーがブロックされている要求を生成します。

TCP など、プロトコルによってはデータが完全形で配信されるよう、この正常型解放を必要とするものがあります。

拡張機能

このセクションでは一歩踏み込んだ XTI/TLI の概念を説明します。

- ライブラリ呼び出し用の非ブロッキング (非同期) モード
- XTI/TLI 環境での TCP および UDP オプションの設定および取得方法
- サーバーによる複数の未処理接続要求のサポートおよびイベント方式の操作のプログラム例

非同期実行モード

多くの XTI/TLI ライブラリルーチンは受信イベントの待機を行うためブロックを行います。ただし、処理時間要求度の高いアプリケーションでは使用するべきではありません。アプリケーションは非同期 XTI/TLI イベントを待機している間にローカル処理が行えます。

XTI/TLI イベントの非同期処理は、非同期機能および XTI/TLI ライブラリルーチンの非ブロックモードの組み合わせによりアプリケーションによる使用が可能です。`poll(2)` システムコールおよび、`ioctl(2)` の `I_SETSIG` コマンドの使用による非同期イベント処理については『*ONC+ 開発ガイド*』に記述されています。

イベントのブロックを行う各 XTI/TLI ルーチンは特別な非ブロッキングモードで実行することが可能です。たとえば、`t_listen(3NSL)` は通常接続要求のブロックを行います。サーバーは非ブロッキング (または非同期) モードの `t_listen(3NSL)` に呼び出しを行うことにより定期的にトランスポートエンドポイント接続要求待機のポーリングを行うことができます。非同期モードはファイル記述子で `O_NDELAY` または `O_NONBLOCK` を設定することにより使用可能となります。これらのモードはフラグとして `t_open(3NSL)` 、または XTI/TLI ルーチンを呼び出す前に `fcntl(2)` を呼び出すことにより設定が可能です。このモードは常時 `fcntl(2)` によって有効化、無効化が行えます。この章のすべてのプログラム例ではデフォルトの同期処理モードを使用しています。

`O_NDELAY` と `O_NONBLOCK` は各 XLI/TLI ルーチンへ異なる影響を与えます。特定のルーチンへの影響は `O_NDELAY` と `O_NONBLOCK` の正確な意味論を判断する必要があります。

高度なプログラム例

以下の例は重要な 2 つの概念を示しています。1 つ目はサーバーにおける複数の未処理接続要求への管理能力を示しています。2 つ目はイベント方式の XLI/TLI の使用法およびシステムコールインタフェースです。

例 3–4 のサーバーは 1 つの未処理接続要求しかサポートしませんが、XTI/TLI ではサーバーによる複数の未処理接続要求の管理が可能です。複数の接続要求を同時に受信する理由の 1 つは、クライアントを順位付けるからです。複数の接続要求を受信した場合、サーバーはクライアントの優先順位により接続要求を受け付けることが可能です。

複数の未処理接続要求を同時に処理する 2 つ目の理由はシングルスレッド処理の限界です。トランスポートプロバイダによっては、サーバーが 1 つの接続要求の処理を行っている間、他のクライアントはビジー状態の応答を受けます。同時に複数の接続要求の処理が可能な場合、クライアント側にビジー応答が渡されるのは、サーバーで同時に処理可能なクライアントの数を超過した場合のみです。

サーバーの例はイベント方式です。プロセスはトランスポートエンドポイントに受信する XTI/TLI イベントのポーリングを行い、受信したそれぞれのイベントに対し適切な処置を行います。例は受信イベントに対し複数のトランスポートエンドポイントのポーリングを行う能力を示しています。

例 3–9 の定義およびエンドポイント確立機能は、サーバーの例 3–4 と同じものです。

例 3–9 エンドポイント確立 (複数接続へ変更可能)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 1                  /*サーバー既知アドレス*/

int conn_fd;                        /*サーバーの接続*/
extern int t_errno;
/*接続要求を格納*/
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
```

```
     * 1 つのトランスポートエンドポイントをオープンし、バインドする
     * 複数の指定も可能
     */
    if ((pollfds[0].fd = t_open(''/dev/tivc'', O_RDWR,
          (struct t_info *) NULL)) == -1) {
       t_error(''t_open failed'');
       exit(1);
    }
    if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
          T_ALL)) == (struct t_bind *) NULL) {
       t_error(''t_alloc of t_bind structure failed'');
       exit(2);
    }
    bind->qlen = MAX_CONN_IND;
    bind->addr.len = sizeof(int);
    *(int *) bind->addr.buf = SRV_ADDR;
    if (t_bind(pollfds[0].fd, bind, bind) == -1) {
       t_error(''t_bind failed'');
       exit(3);
    }
    /*正しいアドレスがバインドされたかどうか */
    if (bind->addr.len != sizeof(int) ||
       *(int *)bind->addr.buf != SRV_ADDR) {
       fprintf(stderr, ''t_bind bound wrong address\n'');
       exit(4);
    }
}
```

t_open(3NSL) によって戻されるファイル記述子はトランスポートエンドポイントの受信データのポーリングを制御する pollfd 構造体に格納されます (poll(2) を参照)。この例では 1 つのトランスポートエンドポイントのみが確立されます。しかしながら、例の残りの部分は複数のトランスポートエンドポイントを管理するために書かれています。例 3–9 へ少し変更を加えることにより複数のトランスポートエンドポイントをサポートできるようになります。

このサーバーでは t_bind(3NSL) 用に qlen を 1 より大きい値に設定します。これは、サーバーが複数の未処理接続要求の待機を行うことを指定します。サーバーは現在の接続要求の受け付けを、他の接続要求を受け付ける前に行います。この例では、一度に MAX_CONN_IND (の設定値) までの接続要求の待機が行えます。MAX_CONN_IND で指定されている未処理接続要求のサポートが行えない場合、トランスポートプロバイダは qlen の値を小さくするためにネゴシエーションを行うことが可能です。

サーバーはアドレスのバインドが完了し、接続要求処理の準備が整ったら、例 3–10 で示す動作を行います。

例 3–10 接続要求の処理

```
pollfds[0].events = POLLIN;

while (TRUE) {
```

```
 if (poll(pollfds, NUM_FDS, -1) == -1) {
   perror(``poll failed'');
   exit(5);
 }
 for (i = 0; i < NUM_FDS; i++) {
   switch (pollfds[i].revents) {
      default:
         perror(``poll returned error event'');
      exit(6);
      case 0:
         continue;
      case POLLIN:
         do_event(i, pollfds[i].fd);
         service_conn_ind(i, pollfds[i].fd);
    }
   }
}
```

pollfd 構造体の events フィールドは POLLIN に設定され、サーバーへ受信する XTI/TLI イベントを通知します。次にサーバーは無限ループに入り、トランスポートエンドポイントのポーリングを行い、発生したイベントの処理を行います。

poll(2) 呼び出しは無期限に受信イベントのブロックを行います。応答時に各エントリ (各トランスポートエンドポイントに対し 1 つ) は新しいイベントのチェックを受けます。revents が 0 の場合、エンドポイントにイベントの発生はなく、サーバーは次のエンドポイントへ処理を移します。revents が POLLIN の場合はエンドポイント上にイベントがあります。この場合、do_event が呼び出され、イベントの処理が行われます。revents がそれ以外の値の場合は、エンドポイント上のエラーを通知し、サーバーは終了します。複数のエンドポイントの場合は、サーバーはこの記述子を閉じて継続処理を行うことが処理上適しています。

ループの各繰り返しに対し、service_conn_ind が未処理接続要求の処理を行うために呼び出されます。他の接続要求が保留状態の場合、service_conn_ind は新しい接続要求を保存し、あとで応答します。

例 3–11 の do_event は受信するイベントを処理するために呼び出されます。

例 3–11 イベント処理ルーチン

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {
   default:
      fprintf(stderr, "t_look: unexpected event\n");
      exit(7);
   case T_ERROR:
```

```
            fprintf(stderr, "t_look returned T_ERROR event\n");
            exit(8);
        case -1:
            t_error("t_look failed");
            exit(9);
        case 0:
            /*POLLIN の戻りのため、本来起きるべきではない*/
            fprintf(stderr,"t_look returned no event\n");
            exit(10);
        case T_LISTEN:
            /*calls 配列内の未使用要素を捜す*/
            for (i = 0; i < MAX_CONN_IND; i++) {
                if (calls[slot][i] == (struct t_call *) NULL)
                    break;
            }
            if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
                    T_ALL)) == (struct t_call *) NULL) {
                t_error("t_alloc of t_call structure failed");
                exit(11);
            }
            if (t_listen(fd, calls[slot][i] ) == -1) {
                t_error("t_listen failed");
                exit(12);
            }
            break;
        case T_DISCONNECT:
            discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
            if (discon == (struct t_discon *) NULL) {
                t_error("t_alloc of t_discon structure failed");
                exit(13)
            }
            if(t_rcvdis( fd, discon) == -1) {
                t_error("t_rcvdis failed");
                exit(14);
            }
            /*配列内から切断要求エントリを見つけ、削除*/
            for (i = 0; i < MAX_CONN_IND; i++) {
                if (discon->sequence == calls[slot][i]->sequence) {
                    t_free(calls[slot][i], T_CALL);
                    calls[slot][i] = (struct t_call *) NULL;
                }
            }
            t_free(discon, T_DIS);
            break;
        }
    }
```

引数は番号 (*slot*) とファイル記述子 (*fd*) です。*slot* は各トランスポートエンドポイントのエントリを持つグローバル配列 `calls` のインデックスです。各エントリはエンドポイント宛に受信される接続要求を格納する `t_call` 構造体の配列です。

`do_event` は `t_look(3NSL)` を呼び出し、*fd* により指定されたエンドポイント上の XTI/TLI イベントの識別を行います。イベントが接続要求 (`T_LISTEN` イベント) の場合、または切断要求 (`T_DISCONNECT` イベント) の場合、イベントは処理されます。それ以外の場合、サーバーはエラーメッセージを出力し、終了します。

接続要求の場合、do_event は最初の未使用エントリを捜すため未処理接続要求配列の走査を行います。エントリには t_call 構造体が割り当てられ、接続要求は t_listen(3NSL) によって受信されます。配列は同時に扱える未処理接続要求の最大数を保持するのに十分な大きさを持っています。接続要求の処理は延期されます。

接続の切断要求は事前に送られた接続要求と対応していなければなりません。do_event は、要求を受信するために t_discon 構造体を割り当てます。この構造体には以下のフィールドが存在します。

```
struct t_discon {
  struct netbuf udata;
  int reason;
  int sequence;
}
```

udata は接続の切断要求とともに送信されたデータを持っています。reason にはプロトコル固有の接続の切断理由コードが含まれています。sequence が接続の切断要求と対応する接続要求を特定します。

t_rcvdis(3NSL) は接続の切断要求を受信するために呼び出されます。接続要求の配列は、接続の切断要求の sequence 番号と一致する sequence 番号を持つ接続要求を捜すために走査されます。一致する接続要求が見つかった時点で、構造体は解放され、エントリは NULL に設定されます。

トランスポートエンドポイント上にイベントが発見された場合、service_conn_ind がエンドポイント上のすべての待機状態の接続要求の処理を行うために呼び出されます。例 3–12 を参照してください。

例 3–12 すべての接続要求の処理

```
service_conn_ind(slot, fd)
{
   int i;

 for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == (struct t_call *) NULL)
         continue;
      if((conn_fd = t_open( ``/dev/tivc'', O_RDWR,
            (struct t_info *) NULL)) == -1) {
         t_error("open failed");
         exit(15);
      }
      if (t_bind(conn_fd, (struct t_bind *) NULL,
            (struct t_bind *) NULL) == -1) {
         t_error("t_bind failed");
         exit(16);
      }
      if (t_accept(fd, conn_fd, calls[slot][i]) == -1) {
         if (t_errno == TLOOK) {
            t_close(conn_fd);
            return;
```

```
            }
            t_error("t_accept failed");
            exit(167);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = (struct t_call *) NULL;
        run_server(fd);
    }
}
```

それぞれのトランスポートエンドポイントについて、未処理の接続要求の配列が走査されます。サーバーは各要求ごとに、応答するトランスポートエンドポイントを開き、エンドポイントにアドレスを設定し、エンドポイントへの接続を受け取ります。現在の要求を受け取る前に他のイベント (接続要求または切断要求) を受信した場合、`t_accept(3NSL)` は失敗し、`t_errno` に `TLOOK` を設定します (保留状態の接続要求イベントまたは切断要求イベントがトランスポートエンドポイントにある場合は、未処理の接続要求を受け取ることはできません)。

このエラーが発生したときは、応答するトランスポートエンドポイントは閉じられ、`service_conn_ind` はただちに戻ります (現在の接続要求は後で処理するために保存されます)。これにより、サーバーのメイン処理ループに入ることができ、新しいイベントが次の `poll(2)` への呼び出しによって発見されます。この方法で、ユーザーは複数の接続要求を順に処理できます。

結果的にすべてのイベントが処理され、`service_conn_ind` はそれぞれの接続要求を順に受け取ることができます。接続が確立してからは、例 3–5 でサーバーに使用された `run_server` ルーチンが呼び出され、データの転送を管理します。

非同期ネットワーク通信

この項ではリアルタイムアプリケーション用の XTI/TLI を使用した非同期ネットワーク通信の方法を説明しています。SunOS は非同期ネットワークの XTI/TLI イベントの処理を、STREAMS 非同期機能と XTI/TLI ライブラリルーチンの非ブロッキングモードの組み合わせによりサポートしています。

ネットワークプログラミングモデル

ファイルおよびデバイス I/O と同様にネットワーク転送は、プロセスサービス要求による同期または非同期の実行が可能です。

同期ネットワーキング

同期ネットワーキングはファイルおよびデバイス I/O 同様に進行します。write(2) 関数と同様、送信要求はメッセージのバッファー化が行われてから戻されますが、バッファー領域がすぐに確保できない場合は、呼び出し処理を中断する可能性もあります。read(2) 関数と同様、受信要求は必要なデータが到着するまで呼び出し処理の実行を中断させます。 SunOS ではトランスポートサービスの範囲に関する保証は存在しないため、他のデバイスに関連する形でのリアルタイム処理を目的とした使用においての同期ネットワーキングの使用は不適切であるといえます。

非同期ネットワーキング

非同期ネットワーキングは非ブロッキングサービス要求により提供されます。また、データが送信または受信される接続の確立時にアプリケーションが非同期通知を要求することも可能です。

非同期コネクションレスモードサービス

非同期コネクションレスモードネットワーキングはエンドポイントに非ブロッキングサービスを構成し、データの転送時期をポーリングまたは非同期通知によって受信することにより行われます。非同期通知が使用された場合、実際のデータの受信は通常シグナルハンドラ内で行われます。

エンドポイントの非同期化

エンドポイントの確立が t_open(3NSL) により行われ、t_bind(3NSL) により識別が確立された後、エンドポイントを非同期サービスで使用するために構成することが可能です。これは fcntl(2) 関数を使用し、エンドポイント上に O_NONBLOCK フラグを設定することにより可能です。これにより、使用可能なバッファー領域がすぐに確保できない場合、t_sndudata(3NSL) への呼び出しは -1 を戻し、t_errno を TFLOW に設定します。同様に、データが存在しない場合、 t_rcvudata(3NSL) への呼び出しは -1 を戻し、t_errno を TNODATA に設定します。

非同期ネットワーク転送

データの着信、またはエンドポイント上でのデータ受信待機のチェックを定期的に行うためにアプリケーションが poll(2) 関数を使用することは可能ですが、デー

タが着信した場合に非同期通知の受信が必要な場合があります。ioctl(2) 関数の I_SETSIG を使用することにより、エンドポイント上にデータが着信した場合、プロセスに SIGPOLL シグナルの送信要求を行うことが可能です。アプリケーション側では複数のメッセージが単一のシグナルとして送信されないようチェックを行うべきです。

以下の例でアプリケーションによって選択されたトランスポートプロトコルの名前は protocol です。

```
#include <sys/types.h>
#include <tiuser.h>
#include <signal.h>
#include <stropts.h>

int    fd;
struct t_bind    *bind;
void    sigpoll(int);

 fd = t_open(protocol, O_RDWR, (struct t_info *) NULL);

 bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
 ...     /*バインドされるアドレスの設定*/
 t_bind(fd, bind, bin

 /*エンドポイントを非ブロッキング化*/
 fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

 /*SIGPOLL 用のシグナルハンドラを確立*/
 signal(SIGPOLL, sigpoll);

 /*受信データが存在する場合 SIGPOLL を要求*/
 ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

 ...

void sigpoll(int sig)
{
 int     flags;
 struct t_unitdata     ud;

 for (;;) {
  ... /*ud を初期化 */
  if (t_rcvudata(fd, &ud, &flags) < 0) {
   if (t_errno == TNODATA)
    break;  /* メッセージが存在しない */
   ... /*他のエラー状態の処理 */
 }
 ... /*ud 内のメッセージの処理*/
}
```

非同期コネクションモードサービス

コネクションモードサービスでは、データ転送のみではなく、接続の確立そのものを非同期で行うようアプリケーションによって設定することが可能です。操作の

シーケンスは他のプロセスに接続しようとしている場合と、接続を待機している場合とで異なります。

非同期による接続の確立

プロセスは接続を試み、非同期で接続を完了することが可能です。プロセスは最初に接続エンドポイントを作成し、fcntl(2) を使用してエンドポイントを非ブロッキング操作を行うように構成します。コネクションレスデータ転送同様、エンドポイントは接続の確立時とそれ以降のデータ転送に対し非同期通知が行われるよう構成することも可能です。それに続き接続プロセスは t_connect(3NSL) 関数を使用して転送設定の初期化を行います。その後、t_rcvconnect(3NSL) 関数を使用して接続の確立の確認が行われます。

接続の非同期使用

非同期状態で接続の待機を行う場合、プロセスは最初にサービスアドレスにバインドされた非ブロッキングエンドポイントを確立します。poll(2) の結果、または非同期通知によって接続要求の着信が伝えられた場合、プロセスは t_listen(3NSL) 関数を使用して接続要求を取得します。プロセスは接続を受け付けるために t_accept(3NSL) 関数を使用します。応答を行うエンドポイントは非同期データ転送を行うため個別に構成されている必要があります。

以下の例では非同期による接続要求を行う方法を示しています。

```
#include <tiuser.h>
int             fd;
struct t_call   *call;

 fd = .../*非ブロッキングエンドポイントを確立*/

 call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
 .../*call 構造体の初期化*/
 t_connect(fd, call, call);

 /*接続要求は非同期で進行*/

 .../*接続が受け付けられた通知を受信*/
 t_rcvconnect(fd, &call);
```

以下の例では非同期接続の待機の方法を示しています。

```
#include <tiuser.h>
int             fd, res_fd;
struct t_call   call;

 fd = ... /*非ブロッキングエンドポイントを確立*/
```

```
.../*接続要求が着信した通知を受信*/

call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
t_listen(fd, &call);

.../*接続を受け付けるか拒否するかの判定*/
res_fd = ... /*応答用に非ブロッキングエンドポイントを確立*/

t_accept(fd, res_fd, call);
```

非同期オープン

アプリケーションが、リモートホストからマウントされたファイスシステム、または初期化が長期化する可能性のあるデバイス上の通常ファイルを動的に開く必要性が発生する場合があります。このようなファイルオープン作業を行っている場合、アプリケーションは他のイベントに対するリアルタイム応答が行えません。SunOS ではこの問題を解決するために、第 2 のプロセスに実際のファイルオープン作業を行わせ、ファイル記述子をリアルタイム処理に渡す機能が提供されています。

ファイル記述子の転送

SunOS の STREAMS インタフェースでは 1 つのプロセスから別のプロセスへオープンファイル記述子を渡すメカニズムを装備しています。オープンファイル記述子を持つプロセス ioctl(2) 関数の引数 I_SENDFD を使用します。もう 1 つのプロセスは ioctl(2) の引数 I_RECVFD を使用してファイル記述子を取得します。

この例では、親プロセスはテストファイルに関する情報を出力し、パイプを作成します。次に親は、テストファイルを開き、パイプを使用してオープンファイル記述子を親へ戻す子プロセスを作成します。その後、親プロセスは新しいファイル記述子の状態情報を表示します。

例 **3–13** ファイル記述子転送

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <stdio.h>

#define TESTFILE "/dev/null"
main(int argc, char *argv[])
{
 int fd;
 int pipefd[2];
 struct stat statbuf;

 stat(TESTFILE, &statbuf);
 statout(TESTFILE, &statbuf);
 pipe(pipefd);
```

```
 if (fork() == 0) {
  close(pipefd[0]);
  sendfd(pipefd[1]);
 } else {
  close(pipefd[1])
  recvfd(pipefd[0]);
 }
}

sendfd(int p)
{
 int tfd;

 tfd = open(TESTFILE, O_RDWR);
 ioctl(p, I_SENDFD, tfd);
}

recvfd(int p)
{
 struct strrecvfd rfdbuf;
 struct stat statbuf;
 char   fdbuf[32];

 ioctl(p, I_RECVFD, &rfdbuf);
 fstat(rfdbuf.fd, &statbuf);
 sprintf(fdbuf, "recvfd=%d", rfdbuf.fd);
 statout(fdbuf, &statbuf);
}

statout(char *f, struct stat *s)
{
 printf("stat: from=%s mode=0%o, ino=%ld, dev=%lx, rdev=%lx\n",
  f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
 fflush(stdout);
}
```

状態遷移

以下の表に、XTI/TLI 関連のすべての状態遷移を示します。最初に、状態とイベントについて説明します。

XTI/TLI 状態

表 3–8 に、XTI/TLI の状態遷移で経過する状態およびサービスタイプを定義します。

表 3–8 XTI/TLI 状態遷移とサービスタイプ

状態	説明	サービスタイプ
T_UNINIT	初期化未完 – インタフェースの初期状態と終了状態	T_COTS、T_COTS_ORD、T_CLTS
T_UNBND	初期化されているが、バインドされていない	T_COTS、T_COTS_ORD、T_CLTS
T_IDLE	接続が確立していない	T_COTS、T_COTS_ORD、T_CLTS
T_OUTCON	クライアントに対する送信接続が保留中	T_COTS、 T_COTS_ORD
T_INCON	サーバーに対する受信接続が保留中	T_COTS、 T_COTS_ORD
T_DATAXFER	データ転送	T_COTS、T_COTS_ORD
T_OUTREL	送信正常型解放 (正常型解放要求待ち)	T_COTS_ORD
T_INREL	受信正常型解放 (正常型解放要求の送信待ち)	T_COTS_ORD

送信イベント

表 3–9 に記載する送信イベントは、指定のトランスポートルーチンがトランスポートプロバイダに要求または応答を送信したときに返される状態に対応しています。「accept」など、この表に記載する一部のイベントは、それが発生したコンテキストによって意味が変わります。これらのコンテキストは、次の変数の値に基づきます。

- *ocnt* – 未処理の接続要求の数
- *fd* – 現在のトランスポートエンドポイントのファイル記述子
- *resfd* – 接続が受け入れられるトランスポートエンドポイントのファイル記述子

表 3–9　送信イベント

イベント	説明	サービスタイプ
opened	正常に t_open(3NSL) が終了した	T_COTS、T_COTS_ORD、T_CLTS
bind	正常に t_bind(3NSL) が終了した	T_COTS、T_COTS_ORD、T_CLTS
optmgmt	正常に t_optmgmt(3NSL) が終了した	T_COTS、T_COTS_ORD、T_CLTS
unbind	正常に t_unbind(3NSL) が終了した	T_COTS、T_COTS_ORD、T_CLTS
closed	正常に t_close(3NSL) が終了した	T_COTS、T_COTS_ORD、T_CLT
connect1	同期モードの t_connect(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
connect2	非同期モードの t_connect(3NSL) で TNODATA エラーが発生したか、または切断要求がトランスポートエンドポイントに到着したことにより TLOOK エラーが発生した	T_COTS、T_COTS_ORD
accept1	ocnt == 1、fd == resfd で t_accept(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
accept2	ocnt == 1、fd != resfd で t_accept(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
accept3	ocnt > 1 で t_accept(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
snd	正常に t_snd(3NSL) が終了した	T_COTS、T_COTS_ORD
snddis1	ocnt <= 1 で t_snddis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
snddis2	ocnt > 1 で t_snddis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
sndrel	正常に t_sndrel(3NSL) が終了した	T_COTS_ORD
sndudata	正常に t_sndudata(3NSL) が終了した	T_CLTS

表 3–9 送信イベント 続く

受信イベント

受信イベントは、指定のルーチンが正常に終了したときに発生します。これらのルーチンは、トランスポートプロバイダからのデータやイベント情報を返します。ルーチンからの戻り値に直接関連付けられていない入力イベントは、pass_conn だけで、接続が他のエンドポイントに移行するときに発生します。エンドポイントで XTI/TLI ルーチンを呼び出さなくても、接続を渡しているエンドポイントではこのイベントが発生します。

表 3–10 に示す rcvdis イベントは、それぞれ ocnt の値が異なります。ocnt とは、エンドポイントでの未処理接続要求の数です。

表 3–10 受信イベント

イベント	接続	サービスタイプ
listen	正常に t_listen(3NSL) が終了した	T_COTS、T_COTS_ORD
rcvconnect	正常に t_rcvconnect(3NSL) が終了した	T_COTS、T_COTS_ORD
rcv	正常に t_rcv(3NSL) が終了した	T_COTS、T_COTS_ORD
rcvdis1	onct <= 0 で t_rcvdis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
rcvdis2	onct == 1 で t_rcvdis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
rcvdis3	onct > 1 で t_rcvdis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
rcvrel	正常に t_rcvrel(3NSL) が終了した	T_COTS_ORD
rcvudata	正常に t_rcvudata(3NSL) が終了した	T_CLTS

表 3–10　受信イベント　続く

イベント	接続	サービスタイプ
rcvuderr	正常に t_rcvuderr(3NSL) が終了した	T_CLTS
pass_conn	渡された接続を受け取った	T_COTS、T_COTS_ORD

トランスポートユーザーの動作

次に示す一部の状態遷移は、トランスポートユーザーが行うべき動作の一覧を示しています。各動作は、次の一覧に示す値で表現します。

- 未処理の接続要求のカウントにゼロを設定する。
- 未処理の接続要求のカウントを 1 だけ増やす。
- 未処理の接続要求のカウントを 1 だけ減らす。
- t_accept(3NSL) で示されるように、別のトランスポートエンドポイントに接続を移す。

状態テーブル

状態テーブルは、XTI/TLI の状態遷移を示します。状態テーブルの列には現在の状態を、行には現在のイベントを、行と列の交差する部分では次に発生する状態を示しています。次に発生する状態が空の場合は、状態とイベントの組み合わせが無効であることを意味します。また次に発生する状態には、動作一覧が示されている場合もあります。動作は、指定された順序で実行しなければなりません。

状態テーブルを見る場合は、次の点に注意してください。

- t_close(3NSL) によって、コネクション型のトランスポートプロバイダのために、確立された接続が終了します。接続の終了はトランスポートプロバイダでサポートされているサービスタイプに従って、正常型解放の手順をふむことも、異常終了することもあります。t_getinfo(3NSL) を参照してください。
- トランスポートユーザーがシーケンス外の関数を発行すると、その関数は失敗し、t_errno が TOUTSTATE に設定されます。この状態は変更できません。

- t_connect(3NSL) から TLOOK または TNODATA というエラーコードが返されると、98ページの「イベント処理」で説明した状態変更が発生します。以下の状態テーブルでは、XTI/TLI を正しく使用していることを前提としています。
- 関数のマニュアルページに特に指定がない限り、他のトランスポートエラーによって状態が変化することはありません。
- t_getinfo(3NSL)、t_getstate(3NSL)、t_alloc(3NSL)、t_free(3NSL)、t_sync(3NSL)、t_look(3NSL)、t_error(3NSL) といったサポート関数は、状態に影響しないので、この状態テーブルから除外されています。

表 3–11、表 3–12、表 3–13 および表 3–14 では、エンドポイントの確立、コネクションレスモードでのデータの転送、コネクションモードでの接続確立/接続解放/データ転送を示します。

表 3–11 接続確立時における状態

イベント/状態	**T_UNINIT**	**T_UNBND**	**T_IDLE**
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt (TLI のみ)			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

表 3–12 コネクションモードにおける状態 — その 1

イベント/状態	**T_IDLE**	**T_OUTCON**	**T_INCON**	**T_DATAXFER**
connect1	T_DATAXFER			
connect2	T_OUTCON			
rcvconnect		T_DATAXFER		
listen	T_INCON [2]		T_INCON [2]	

表 3–12　コネクションモードにおける状態 — その 1　続く

イベント/状態	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
accept1			T_DATAXFER [3]	
accept2			T_IDLE [3] [4]	
accept3			T_INCON [3] [4]	
snd				T_DATAXFER
rcv				T_DATAXFER
snddis1		T_IDLE	T_IDLE [3]	T_IDLE
snddis2			T_INCON [3]	
rcvdis1		T_IDLE		T_IDLE
rcvdis2			T_IDLE [3]	
rcvdis3			T_INCON [3]	
sndrel				T_OUTREL
rcvrel				T_INREL
pass_conn	T_DATAXFER			
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

表 3–13　コネクションモードにおける状態 — その 2

イベント/状態	T_OUTREL	T_INREL	T_UNBND
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd		T_INREL	
rcv	T_OUTREL		
snddis1	T_IDLE	T_IDLE	
snddis2			
rcvdis1	T_IDLE	T_IDLE	
rcvdis2			
rcvdis3			
sndrel		T_IDLE	
rcvrel	T_IDLE		
pass_conn			T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	

表 3–13　コネクションモードにおける状態 — その 2　続く

表 3–14　コネクションレスモードにおける状態

イベント/状態	**T_IDLE**
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

プロトコルに依存しない処理に関する指針

XTI/TLI が提供する一連のサービスは、多くのトランスポートプロトコルに共通であり、XTI/TLI を使用すると、アプリケーションはプロトコルに依存しない処理が可能になります。しかし、すべてのトランスポートプロトコルが XTI/TLI をサポートしているわけではありません。ソフトウェアをさまざまなプロトコル環境で実行する必要がある場合は、共通のサービスだけを使用してください。なお、次に示すサービスはすべてのトランスポートプロトコルに共通とは限らないので、注意が必要です。

1. コネクションモードのサービスでは、すべてのトランスポートプロバイダで転送サービスデータ単位 (TSDU) がサポートされるとは限りません。接続の際に論理的なデータ境界が保たれることを前提としてはなりません。
2. プロトコルおよび実装に固有のサービス範囲は、t_open(3NSL) および t_getinfo(3NSL) の各ルーチンによって返されます。これらの範囲に基づいて、バッファーを割り当て、プロトコルに固有のトランスポートアドレスおよびオプションを格納します。
3. ユーザーデータは、t_connect(3NSL) および t_snddis(3NSL) などの接続要求や切断要求を使用して送信してはなりません。すべてのトランスポートプロトコルがこのような方法で機能するとは限りません。

4. t_listen(3NSL) に使用する t_call 構造体に含まれるバッファーには、接続確立時にクライントが送信するデータを格納できるだけの大きさが必要です。t_alloc(3NSL) の T_ALL 引数を使用して、最大バッファーサイズを設定し、現在のトランスポートプロバイダのアドレス、オプションおよびユーザーデータを格納します。
5. t_bind(3NSL) に指定するプロトコルアドレスは、クライアント側のエンドポイントで指定してはなりません。トランスポートエンドポイントへの適切なアドレスの割り当ては、トランスポートプロバイダに任せます。サーバーは、トランスポートプロバイダの名前空間を知らなくても、t_bind(3NSL) のプロトコルアドレスを取り込むことができなければなりません。
6. トランスポートアドレスの形式を仮定してはなりません。また、トランスポートアドレスをプログラム内で定数としてはなりません。この詳細については、第 4 章を参照してください。
7. t_rcvdis(3NSL) に関連付けられた理由コードは、プロトコルに依存します。プロトコルからの独立性が重要な問題となる場合は、この情報を使用してはなりません。
8. t_rcvuderr(3NSL) のエラーコードは、プロトコルに依存します。プロトコルからの独立性が問題となる場合は、この情報を使用してはなりません。
9. プログラム内にデバイス名をコーディングしてはなりません。デバイスノードは、特定のトランスポートプロバイダを指定し、プロトコルに依存します。詳細については、第 4 章を参照してください。
10. 複数のプロトコル環境で実行されるプログラムでは、t_sndrel(3NSL) および t_rcvrel(3NSL) のコネクションモードサービスで提供される、オプションの正常型解放機能を使用してはなりません。正常型解放機能は、すべてのコネクション型トランスポートプロトコルでサポートされているわけではありません。この機能を使用すると、解放型システムと正常に通信できなくなることがあります。

XTI/TLI とソケットインタフェース

XTI/TLI とソケットとでは、同じタスクの処理方法が異なります。たいていの場合、両者は、機能的には同様な機構とサービスとを提供しますが、ルーチンや低レベルのサービスで 1 対 1 の互換性があるわけではありません。アプリケーションを移植しようとする場合は、XTI/TLI インタフェースとソケットベースのインタフェースとの間の類似点や相違点をよく知る必要があります。

トランスポートの独立性に関しては、次の問題があります。これらの問題は、RPC アプリケーションにも関係があります。

- 特権ポート *(Privileged ports)* – 特権ポートは、TCP/IP インターネットプロトコルのバークレー版のソフトウェア配布 (BSD) を実装するために開発された機能です。特権ポートは移植可能ではありません。特権ポートの概念は、トランスポートに依存しない環境ではサポートされません。
- あいまいなアドレス *(Opaque addresses)* – ホストを指定するアドレス部分と、そのホスト上でのサービスを指定するアドレス部分とを、トランスポートに依存しない方法で区別することはできません。ネットワークサービスのホストアドレスを認識できることを前提としたコードは必ず変更してください。
- ブロードキャスト *(Broadcast)* – トランスポートに依存しない形態のブロードキャストアドレスはありません。

ソケット関数と XTI/TLI 関数との対応関係

表 3–15 に、XTI/TLI 関数とソケット関数との対応関係をおおまかに示します。コメント欄には、両者の相違点を示します。コメントがない場合は、関数が同じであるか、または一方のインタフェースに等価の関数がないことを意味します。

表 3–15 TLI 関数とソケット関数の対応表

TLI 関数	ソケット関数	コメント
t_open(3NSL)	socket(3SOCKET)	
--	socketpair(3SOCKET)	
t_bind(3NSL)	bind(3SOCKET)	t_bind(3NSL) は、受信ソケットの待ち行列の深さを設定するが、bind(3SOCKET) は設定しない。ソケットでは、listen(3SOCKET) の呼び出しで待ち行列の長さを指定する
t_optmgmt(3NSL)	getsockopt(3SOCKET) setsockopt(3SOCKET)	t_optmgmt(3NSL) では、トランスポートオプションだけを管理する。getsockopt(3SOCKET) と setsockopt(3SOCKET) では、トランスポート層のオプションだけでなく、ソケット層および任意のプロトコル層でのオプションも管理する

表 3–15 TLI 関数とソケット関数の対応表 続く

TLI 関数	ソケット関数	コメント
t_unbind(3NSL)		--
t_close(3NSL)	close(2)	
t_getinfo(3NSL)	getsockopt(3SOCKET)	t_getinfo(3NSL) は、トランスポートに関する情報を返す。getsockopt(3SOCKET) は、トランスポートおよびソケットに関する情報を返すことができる
t_getstate(3NSL)	-	
t_sync(3NSL)	-	
t_alloc(3NSL)	-	
t_free(3NSL)	-	
t_look(3NSL)	-	SO_ERROR オプションを指定した getsockopt(3SOCKET) は、t_look(3NSL) と同じエラー情報を返す
t_error(3NSL)	perror(3C)	
t_connect(3NSL)	connect(3SOCKET)	connect(3SOCKET) は、最初にローカルエンドポイントへのバインドを実行しなくても実行できる。エンドポイントは、t_connect(3NSL) 呼び出しの前にバインドされていなければならない。connect(3SOCKET) は、コネクションレスエンドポイントで実行すると、データグラムのデフォルト宛先アドレスを設定できる。データは、connect(3SOCKET) で送信できる
t_rcvconnect(3NSL)	-	
t_listen(3NSL)	listen(3SOCKET)	t_listen(3NSL) は、接続指示を待つ。listen(3SOCKET) は、待ち行列の深さを設定する
t_accept(3NSL)	accept(3SOCKET)	
t_snd(3NSL)	send(3SOCKET)	

表 3–15 TLI 関数とソケット関数の対応表 続く

TLI 関数	ソケット関数	コメント
	sendto(3SOCKET)	
	sendmsg(3SOCKET)	sendto(3SOCKET) と sendsmg(3SOCKET) は、データグラムモードでもコネクションモードでも機能する
t_rcv(3NSL)	recv(3SOCKET)	
	recvfrom(3SOCKET)	
	recvmsg(3SOCKET)	recvfrom(3SOCKET) および recvmsg(3SOCKET) は、データグラムモードでもコネクションモードでも機能する
t_snddis(3NSL)	-	
t_rcvdis(3NSL)	-	
t_sndrel(3NSL)	shutdown(3SOCKET)	
t_rcvrel(3NSL)	-	
t_sndudata(3NSL)	sendto(3SOCKET)	
	recvmsg(3SOCKET)	
t_rcvuderr(3NSL)	-	
read(2)、write(2)	read(2)、write(2)	XTI/TLI では、read(2) または write(2) を呼び出す前に tirdwr(7M) モジュールをプッシュしておく必要がある。ソケットでは、単に read(2) または write(2) を呼び出すだけでよい

XTI インタフェースへの追加

XNS 5 (Unix98) 標準に新規の XTI インタフェースが導入されました。これらの XTI インタフェースについて、以下に簡単に説明します。詳細については、関連するマニュアルページを参照してください。TLI ユーザーはこれらのインタフェースを使用できません。

データの配布および収集転送インタフェース

`t_sndvudata(3NSL)`	1 つまたは複数の非連続バッファー上のデータ単位を送信する
`t_rcvvudata(3NSL)`	1 つまたは複数の非連続バッファーにデータ単位を受信する
`t_sndv(3NSL)`	接続時に、1 つまたは複数の非連続バッファー上のデータまたは優先データを送信する
`t_rcvv(3NSL)`	接続を経由して受信したデータまたは優先データを、1 つまたは複数の非連続バッファーに格納する

XTI ユーティリティ関数

`t_sysconf(3NSL)`	構成可能な XTI 変数を取得する

追加の接続解放インタフェース

`t_sndreldata(3NSL)`	ユーザーデータを使用して正常型解放を発行したり、応答したりする
`t_rcvreldata(3NSL)`	正常型解放指示やユーザーデータが含まれる確認を受け取る

注 - 追加のインタフェースである `t_sndreldata(3NSL)` と `t_rcvreldata(3NSL)` は、「最小 OSI」と呼ばれる固有のトランスポートでだけ使用されます。最小 OSI は、Solaris プラットフォームではサポートされていません。これらのインタフェースは、インターネットトランスポート (TCP または UDP) と併用することはできません。

第 4 章

トランスポート選択と名前からアドレスへのマッピング

この章では、トランスポートの選択およびネットワークアドレスの解決方法を示します。また、アプリケーションが使用できる通信プロトコルを指定できるようにするインタフェースについて説明します。さらに、名前をネットワークアドレスに直接マッピングする追加機能についても取り上げます。

注 - この章で使用するネットワークおよびトランスポートという用語は、OSI 参照モデルのトランスポート層に準拠するプログラム可能なインタフェースを指す場合、同じ意味です。ネットワークという用語は、何らかの電子媒体を介して接続できるコンピュータの物理的な集まりを指す場合にも使用されます。

マルチスレッドに対して安全なトランスポート選択

この章で取り上げるインタフェースは、マルチスレッドに対して安全です。このことは、トランスポートの選択機能呼び出しを行うアプリケーションを、マルチスレッド対応アプリケーション内で自由に使用できることを意味します。ただし、アプリケーションの多重度は、特定されていません。

トランスポート選択

分散アプリケーションを各種のプロトコルに移植可能にするには、分散アプリケーションでトランスポートサービスの標準インタフェースを使用する必要があります。トランスポート選択サービスが提供するインタフェースを使用すると、アプリケーションは、使用するプロトコルを選択できます。これによって、アプリケーションは、「プロトコル」と「媒体」に依存しなくなります。

トランスポート選択機能を使用すると、クライアントアプリケーションは、サーバーとの通信を確立するまでに、どのトランスポートが使用できるかを簡単に試すことができます。一方、サーバーアプリケーションは、複数のトランスポートに関する要求を受け入れ、複数のプロトコルを経由して通信を行うことができます。どのトランスポートが使用できるかは、ローカルなデフォルトシーケンスで指定された順序、またはユーザーが指定した順序で試すことができます。

使用可能なトランスポートのうち、どれを選択するかを決定するのは、アプリケーションの役割です。トランスポート選択機構を使用すると、選択が統一的な方法で簡単に行えます。

トランスポート選択のしくみ

トランスポートの選択は、次に基づいて行われます。

- ネットワーク構成データベース (/etc/netconfig ファイル)。このデータベースには、システム上の各ネットワークに関するエントリが含まれています。
- 環境変数 NETPATH (使用は任意)

環境変数 NETPATH は、ユーザーが設定します。この環境変数は、トランスポート識別子を順番に指定したリストと同じです。トランスポート識別子は、netconfig network ID フィールドに一致し、netconfig(4) ファイル内のレコードとリンクしています。netconfig(4) ファイルについては、141ページの「/etc/netconfig ファイル」を参照してください。ネットワーク選択インタフェースは、ネットワーク構成データベースへの一連のアクセスルーチンから構成されます。

ライブラリルーチン一式は、環境変数 NETPATH で指定される /etc/netconfig のエントリにだけアクセスします。

setnetpath(3NSL)	NETPATH の検索を初期化する
getnetpath(3NSL)	NETPATH の次の要素に対応する netconfig(4) エントリへのポインタを返す
endnetpath(3NSL)	処理の完了時に NETPATH の要素へのデータベースポインタを解放する

これらのルーチンについては、144ページの「NETPATH を経由した netconfig(4) データへのアクセス」および getnetpath(3NSL) を参照してください。これらのルーチンを使用すると、アプリケーションが使用するトランスポート選択にユーザーが影響を与えることができます。

トランスポート選択へのユーザーの影響を避けるには、netconfig(4) データベースに直接アクセスするルーチンを使用します。これらのルーチンについては、146ページの「netconfig(4) へのアクセス」および getnetconfig(3NSL) を参照してください。

setnetconfig(3NSL)	データベースの最初のインデックスへのレコードポインタを初期化する
getnetconfig(3NSL)	netconfig(4) データベース内の現在のレコードへのポインタを返し、次のレコードを指すようにポインタを 1 だけ増加する
endnetconfig(3NSL)	処理の完了時にデータベースポインタを解放する

次の 2 つのルーチンは、netconfig(4) エントリおよびそれが表すデータ構造体を操作します。これらのルーチンについては、146ページの「netconfig(4) へのアクセス」を参照してください。

getnetconfigent(3NSL)	netid に対応する netconfig 構造体へのポインタを返す
freenetconfigent(3NSL)	getnetconfigent(3NSL) で返される構造体を解放する

/etc/netconfig ファイル

netconfig(4) ファイルには、ホスト上のすべてのトランスポートプロトコルが記述されています。表 4–1 で、netconfig(4) ファイルのエントリについて簡単に説明しています。詳細については、netconfig(4) のマニュアルページを参照してください。

表 4–1 netconfig(4) ファイル

エントリ	説明
network ID	tcp のようなトランスポート名のローカルな表現。このフィールドに tcp や udp のようなよく知られた名前が設定されるとは限らない。また、同じトランスポートについて、2 つのシステムが同じ名前を使用するとも限らない
semantics	特定のトランスポートプロトコルの意味。指定できる意味は次のとおり ■ tpi_clts – コネクションレス ■ tpi_cots – コネクション型 ■ tpi_cots_ord – 正常型解放機能を備えたコネクション型
flags	v またはハイフン (-) だけを指定できるが、(-v) という可視フラグだけが定義されている
protocol family	トランスポートプロバイダのプロトコルファミリ名 (たとえば、inet または loopback)
protocol name	トランスポートプロバイダのプロトコル名。たとえば、*protocol family* が inet の場合、*protocal name* は、tcp、udp または icmp。それ以外の場合は、*protocal name* の値は、ハイフン (-)
network device	トランスポートプロバイダにアクセスする場合に開く、デバイスファイルの完全パス名
name-to-address translation libraries	共有オブジェクトの名前。このフィールドには、名前からアドレスへのマッピングルーチンが格納されている共有オブジェクトのファイル名をコンマで区切って指定する。共有オブジェクトは、環境変数 LD_LIBRARY_PATH のパスに格納される。このフィールドに「-」を指定すると、ホストおよびサービスに関するネームサービス切り換えポリシーが使用されることを意味する

例 4–1 に、netconfig(4) ファイルのサンプルを示します。inet トランスポートについては、このサンプルファイルのコメント部分に示すように、netconfig(4) ファイルの使用方法が変更されました。この変更については、149ページの「名前からアドレスへのマッピング」も参照してください。

例 4–1 netconfig(4) サンプルファイル

```
# The ``Network Configuration'' File.
#
# Each entry is of the form:
```

```
#
#<net <semantics>  <flags> <proto    <proto   <device>     <nametoaddr_libs>
# id>                       family>   name>
#
# The "-" in <nametoaddr_libs> for inet family transports indicates redirection
# to the name service switch policies for "hosts" and "services. The "-" may be
# replaced by nametoaddr libraries that comply with the SVR4 specs, in which
# case the name service switch will be used for netdir_getbyname, netdir_
# getbyaddr, gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
# There are no nametoaddr_libs for the inet family in Solaris anymore.
#
udp       tpi_clts     v   inet       udp      /dev/udp        -
#
tcp       tpi_cots_ord v   inet       tcp      /dev/tcp        -
#
icmp      tpi_raw      -   inet       icmp     /dev/icmp       -
#
rawip     tpi_raw      -   inet       -        /dev/rawip      -
#
ticlts    tpi_clts     v   loopback   -        /dev/ticlts     straddr.so
#
ticots    tpi_cots     v   loopback   -        /dev/ticots     straddr.so
#
ticotsord tpi_cots_ord v   loopback   -        /dev/ticotsord  straddr.so
#
```

ネットワーク選択ライブラリルーチンは、netconfig エントリへのポインタを返します。例 4–2 に、netconfig 構造体を示します。

例 4–2 netconfig 構造体

```
struct netconfig {
   char  *nc_netid;                /* ネットワーク識別子 */
   unsigned  int   nc_semantics;   /* プロトコルの意味 */
   unsigned  int   nc_flag;        /* プロトコルのフラグ */
   char *nc_protofmly;             /* ファミリ名 */
   char *nc_proto;                 /* プロトコル固有 */
   char *nc_device;                /* ネットワーク ID に相当するデバイス名 */
   unsigned  int   nc_nlookups;    /* nc_lookups 内のエントリ数 */
   char **nc_lookups;              /* ルックアッププライブラリのリスト */
   unsigned  int   nc_unused[8];
};
```

有効なネットワーク ID は、システム管理者が定義します。システム管理者は、必ず、ネットワーク ID をローカルに一意にしなければなりません。一意でないと、一部のネットワーク選択ルーチンが正しく機能しません。たとえば、udp というネットワーク ID が指定された netconfig エントリが 2 つあると、getnetconfigent ("udp") がどちらのネットワークを使用するかがわかりません。

システム管理者は、netconfig(4) データベース内のエントリの順序も設定します。/etc/netconfig 内のエントリを見つけるルーチンは、ファイルの先頭から順番に走査し、エントリを返します。netconfig(4) ファイル内のトランスポートの

記述順序は、ルーチンがトランスポートを検索する際のデフォルト順序になります。ループバックエントリは、ファイルの終わりに設定する必要があります。

netconfig(4) ファイルおよび netconfig 構造体については、netconfig(4) のマニュアルページでさらに詳細に説明しています。

環境変数 NETPATH

アプリケーションは、通常、システム管理者が設定したデフォルトのトランスポート検索パスを使用して、使用可能なトランスポートを検索します。ただし、ユーザーがアプリケーションのトランスポート選択に関与したい場合は、環境変数 NETPATH と、144ページの「NETPATH を経由した netconfig(4) データへのアクセス」で説明するルーチンとを使用することによって、アプリケーションがインタフェースを変更できます。これらのルーチンは、環境変数 NETPATH に指定されているトランスポートにだけアクセスします。

NETPATH は、PATH 変数と同様に、トランスポート ID をコロンで区切ったリストです。環境変数 NETPATH 内の各トランスポート ID は、netconfig(4) ファイル内のレコードのネットワーク ID フィールドに対応します。NETPATH については、environ(4) のマニュアルページを参照してください。

デフォルトトランスポートセットは、環境変数 NETPATH (次の節で説明する) を経由して netconfig(4) にアクセスするルーチンと、netconfig(4) に直接アクセスするルーチンとで異なります。環境変数 NETPATH を介して netconfig(4) にアクセスするルーチンのデフォルトトランスポートセットは、netconfig(4) ファイルに定義された可視のトランスポートからなります。一方、netconfig(4) に直接アクセスするルーチンのデフォルトトランスポートセットは、netconfig(4) ファイル全体になります。トランスポートが可視になるのは、システム管理者がそのトランスポートの netconfig(4) エントリの flags フィールドに v フラグを設定した場合です。

NETPATH を経由した netconfig(4) データへのアクセス

環境変数 NETPATH を経由して間接的にネットワーク構成データベースにアクセスするルーチンは、3 つあります。この環境変数では、アプリケーションが使用するトランスポート (複数も可) と、使用できるトランスポートを試す順序とを指定しま

す。NETPATH の構成要素は、左から右へと読み込まれます。これらの関数は、次のインタフェースを使用します。

```
#include <netconfig.h>

void *setnetpath(void);
struct netconfig *getnetpath(void *);
int endnetpath(void *);
```

setnetpath(3NSL) の呼び出しは、NETPATH の検索を初期化します。また、環境変数 NETPATH で指定したエントリが含まれるデータベースへのポインタを返します。このポインタはハンドルと呼ばれ、getnetpath(3NSL) を使用してデータベース内を移動する際に使用します。setnetpath(3NSL) 関数は、getnetpath(3NSL) を最初に呼び出す前に呼び出しておく必要があります。

getnetpath(3NSL) は、最初に呼び出されると、環境変数 NETPATH の最初の構成要素に対応する netconfig(4) ファイル内のエントリへのポインタを返します。以降の getnetpath(3NSL) 呼び出しでは、環境変数 NETPATH の次の構成要素に対応する netconfig(4) ファイル内のエントリへのポインタを返します。NETPATH に構成要素がなくなると、getnetpath(3NSL) は NULL を返します。setnetpath(3NSL) を最初に呼び出さずに getnetpath(3NSL) を呼び出すと、エラーが発生します。getnetpath(3NSL) には、setnetpath(3NSL) で返されたポインタを引数に指定する必要があります。

getnetpath(3NSL) は、無効な NETPATH 構成要素を無視するだけで、メッセージを出力しません。無効な NETPATH 構成要素とは、netconfig(4) データベースに対応するエントリがないものです。

環境変数 NETPATH が設定されていない場合、getnetpath(3NSL) は、netconfig(4) データベースのデフォルトトランスポートまたは可視トランスポートの順序が NETPATH に設定されているように振る舞います。

endnetpath(3NSL) は、処理の完了時に呼び出され、環境変数 NETPATH 内の要素へのデータベースポインタを解放します。setnetpath(3NSL) が事前に呼び出されていないと、endnetpath(3NSL) は失敗します。例 4–3 に、setnetpath(3NSL)、getnetpath(3NSL)、および endnetpath(3NSL) の各ルーチンを示します。

例 4–3 setnetpath(3NSL)、getnetpath(3NSL) および endnetpath(3NSL) の各関数

```
#include <netconfig.h>

void *handlep;
struct netconfig *nconf;
```

```
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}

while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * nconf がトランスポートプロバイダ情報を示す
    */
}
endnetpath(handlep);
```

getnetpath(3NSL) 経由で取得した netconfig(4) 構造体は、endnetpath(3NSL) の実行後無効になります。構造体内のデータを保持するには、getnetconfigent(nconf->nc_netid) を使用して、それらを新しいデータ構造体にコピーします。

netconfig(4) へのアクセス

/etc/netconfig にアクセスし、netconfig(4) 内のエントリを検索する関数は、3 つあります。これら 3 つのルーチンである setnetconfig(3NSL)、getnetconfigent(3NSL) および endnetconfig(3NSL) は、次のインタフェースを使用します。

```
#include <netconfig.h>

void *setnetconfig(void);
struct netconfig *getnetconfig(void *);
int endnetconfig(void *);
```

setnetconfig(3NSL) 呼び出しは、データベースの最初のインデックスへのレコードポインタを初期化します。setnetconfig(3NSL) は、最初に getnetconfig(3NSL) を使用する前に使用する必要があります。setnetconfig(3NSL) は、getnetconfig(3NSL) ルーチンが使用する一意のハンドル (データベースへのポインタ) を返します。getnetconfig(3NSL) の各呼び出しは、netconfig(4) データベース内の現在のレコードへのポインタを返し、次のレコードを指すようにポインタを 1 だけ増加させます。また、getnetconfig(3NSL) は netconfig(4) データベース全体の検索にも使用できます。getnetconfig(3NSL) は、ファイルの終わりに NULL を返します。

処理の完了時にデータベースポインタを解放するには、endnetconfig(3NSL) を使用する必要があります。endnetconfig(3NSL) は、setnetconfig(3NSL) の前に呼び出してはなりません。

例 4–4　setnetconfig(3NSL)、getnetconfig(3NSL)、および endnetconfig(3NSL) の各関数

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig()) == (void *)NULL){
   nc_perror(argv[0]);
   exit(1);
}
/*
 * トランスポートプロバイダ情報は、nconf に記載されている。
 * process_transport は、ユーザーが提供するルーチンであり、
 * トランスポート nconf を経由してサーバーへの接続を試みる。
 */
while ((nconf = getnetconfig(handlep)) != (struct netconfig *)NULL){
    if (process_transport(nconf) == SUCCESS)
      break;
}
endnetconfig(handlep);
```

getnetconfigent(3NSL) および freenetconfigent(3NSL) の各関数は、次のインタフェースを使用します。

```
#include <netconfig.h>
struct netconfig *getnetconfigent(char *);
int freenetconfigent(struct netconfig *);
```

getnetconfigent(3NSL) は、netid に対応した netconfig 構造体へのポインタを返します。netid が無効の場合は、NULL を返します。freenetconfigent(3NSL) は、getnetconfigent(3NSL) の前に呼び出してはなりません。

freenetconfigent(3NSL) は、getnetconfigent(3NSL) で返された構造体を解放します。例 4–5 に、getnetconfigent(3NSL) と freenetconfigent(3NSL) の各ルーチンを示します。

例 4–5　getnetconfigent(3NSL) と freenetconfigent(3NSL) の関数

```
/* udp がこのホストの netid と仮定する */
struct netconfig *nconf;

if ((nconf = getnetconfigent(''udp'')) == (struct netconfig *)NULL){
   nc_perror(''no information about udp'');
   exit(1);
}
process_transport(nconf);
freenetconfigent(nconf);
```

可視の全 netconfig(4) エントリ間のループ

setnetconfig(3NSL) 呼び出しを使用して、netconfig(4) データベースで可視とマークされたすべてのトランスポート (flags フィールドに v フラグが設定されているもの) 間を移動します。トランスポート選択ルーチンは、netconfig(4) ポインタを返します。

ユーザー定義の netconfig(4) エントリ間のループ

環境変数 NETPATH にコロンで区切ったトランスポート名のリストを設定することによって、ループを制御できます。NETPATH は、次のように設定します。

```
NETPATH=tcp:udp
```

ループは、最初に tcp エントリを返し、次に udp エントリを返します。NETPATH が定義されていないと、ループは、netconfig(4) ファイル内に存在するすべての可視エントリを格納順に返します。環境変数 NETPATH を使用すると、クライアント側アプリケーションがサービスに接続しようとする順番をユーザーが定義できるだけでなく、サービスが待機できるトランスポートの数をサーバーの管理者が制限できます。

getnetpath(3NSL) と setnetpath(3NSL) を使用して、ネットワークパス変数を取得したり、変更したりします。例 4–6 では、その形式と使用方法を示します。これは、getnetconfig(3NSL) と setnetconfig(3NSL) のルーチンに似ています。

例 4–6 可視のトランスポート間のループ

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig() == (void *) NULL) {
    nc_perror(``setnetconfig'');
   exit(1);
}
while (nconf = getnetconfig(handlep))
    if (nconf->nc_flag & NC_VISIBLE)
      doit(nconf);
(void) endnetconfig(handlep);
```

名前からアドレスへのマッピング

名前からアドレスへのマッピングによって、使用されるトランスポートに関係なく、アプリケーションは指定のホスト上で実行されるサービスのアドレスを取得できます。名前からアドレスへのマッピングでは、次の関数を使用します。

netdir_getbyname(3NSL)	ホスト名およびサービス名を一連のアドレスに対応づける
netdir_getbyaddr(3NSL)	アドレスを、ホスト名およびサービス名に対応づける
netdir_free(3NSL)	名前からアドレスへの変換ルーチンによって割り当てられた構造体を解放する
taddr2uaddr(3NSL)	アドレスを変換し、トランスポートに依存しないアドレスの文字表現を返す
uaddr2taddr(3NSL)	汎用アドレスを netbuf 構造体に変換する
netdir_options(3NSL)	ブロードキャストアドレス、TCP や UDP の予約ポート機能など、トランスポートに固有の機能へのインタフェースをとる

各ルーチンの最初の引数では、トランスポートを示す netconfig(4) 構造体を指します。これらのルーチンは、netconfig(4) 構造体内にあるディレクトリルックアップ用のライブラリパスの配列を使用して、変換が正常終了するまで各パスを呼び出します。

表 4–2 で、ライブラリについて説明します。151ページの「名前からアドレスへのマッピングルーチンの使用」に示すルーチンは、netdir(3NSL) のマニュアルページに定義されています。

注 - tcpip.so、switch.so および nis.so というライブラリは、Solaris 2 環境で廃止されました。この変更の詳細については、nsswitch.conf(4) のマニュアルページおよび gethostbyname(3NSL) マニュアルページの NOTES セクションを参照してください。

表 4–2　名前からアドレスへのマッピングを行うライブラリ

ライブラリ	トランスポートファミリ	説明
-	inet	inet プロトコルファミリを使用するネットワークでは、名前からアドレスへのマッピングは、nsswitch.conf(4) ファイルに定義されている *hosts* と *services* の各エントリに基づくネームサービス切り換え機能が行う。inet 以外のファミリを使用するネットワークに「-」を指定すると、名前からアドレスへのマッピング機能が存在しないことを示す
straddr.so	loopback	ループバックトランスポートのように、文字列をアドレスとして受け入れるプロトコルの、名前からアドレスへのマッピングルーチンが含まれている

straddr.so ライブラリ

straddr.so ライブラリで使用する名前からアドレスへの変換ファイルは、システム管理者が作成し、保守します。straddr.so ファイルには、/etc/net/*transport-name*/hosts と /etc/net/*transport-name*/services があります。*transport-name* は、文字列アドレス (/etc/netconfig ファイルの *network ID* フィールドに指定したもの) を受け入れるトランスポートのローカル名です。たとえば、ticlts のホストファイルは、/etc/net/ticlts/hosts となり、ticlts のサービスファイルは、/etc/net/ticlts/services となります。

たいていの文字列アドレスは、ホストとサービスを区別しませんが、文字列をホスト部分とサービス部分とに分けると、他のトランスポートとの間で一貫性が保てます。/etc/net/*transport-name*/hosts ファイルには、ホストアドレスと見なされるテキスト文字列に続いて、ホスト名を定義します。たとえば、次のように定義します。

```
joyluckaddr joyluck
carpediemaddr carpediem
thehopaddr thehop
pongoaddr pongo
```

ルックアップトランスポートの場合、他のホストを記述することには意味がありません。なぜなら、サービスは、それがインストールされているホスト以外で実行することはできないからです。

/etc/net/*transport-name*/services には、サービス名に続いて、サービスアドレスを特定する文字列を定義します。たとえば、次のように定義します。

```
rpcbind rpc
listen serve
```

ルーチンは、ホストアドレス、ピリオド (.)、およびサービスアドレスを結合して完全な文字列アドレスを作成します。たとえば、pongo での listen サービスのアドレスは、pongoaddr.serve になります。

このライブラリを使用するトランスポート上で、アプリケーションが特定のホスト上のサービスアドレスを要求する場合は、/etc/net/*transport*/hosts にホスト名が、/etc/net/*transport*/services にサービス名がそれぞれ定義されていなければなりません。どちらか一方でも欠けると、名前からアドレスへの変換が失敗します。

名前からアドレスへのマッピングルーチンの使用

この節では、どのようなルーチンが使用できるかについて簡単に説明します。ルーチンは、ネットワーク名を返すか、またはネットワーク名を対応するネットワークアドレスに変換します。なお、netdir_getbyname(3NSL)、netdir_getbyaddr(3NSL)、taddr2uaddr(3NSL) は、データへのポインタを返しますが、これらのポインタは、netdir_free(3NSL) 呼び出しで解放する必要があります。

```
int netdir_getbyname(struct netconfig *nconf,
  struct nd_hostserv *service, struct nd_addrlist **addrs);
```

netdir_getbyname(3NSL) は、*service* に指定されたホスト名とサービス名を、*nconf* で指定されたトランスポートに一致したアドレスセットに対応づけます。nd_hostserv と nd_addrlist の各構造体は、netdir(3NSL) のマニュアルページに定義されています。アドレスへのポインタは、*addrs* に返されます。

使用可能なすべてのトランスポート上で、ホストおよびサービスのすべてのアドレスを取得するには、getnetpath(3NSL) または getnetconfig(3NSL) のいずれかで返される各 netconfig(4) 構造体を使用して netdir_getbyname(3NSL) を呼び出します。

```
int netdir_getbyaddr(struct netconfig *nconf,
  struct nd_hostservlist **service, struct netbuf *netaddr);
```

netdir_getbyaddr(3NSL) は、アドレスをホスト名とサービス名に対応付けます。この関数は、*netaddr* に指定したアドレスを使用して呼び出され、ホスト名とサービス名のペアのリストを *service* に返します。nd_hostservlist 構造体は、netdir(3NSL) に定義されています。

```
void netdir_free(void *ptr, int struct_type);
```

netdir_free(3NSL) ルーチンは、名前からアドレスへの変換ルーチンによって割り当てられた構造体を解放します。表 4–3 に、引数がとる値を示します。

表 4–3 netdir_free(3NSL) ルーチン

struct_type	ptr
ND_HOSTSERV	nd_hostserv 構造体へのポインタ
ND_HOSTSERVLIST	nd_hostservlist 構造体へのポインタ
ND_ADDR	netbuf 構造体へのポインタ
ND_ADDRLIST	nd_addrlist 構造体へのポインタ

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

taddr2uaddr(3NSL) は、*addr* が指すアドレスを変換し、トランスポートに依存しない文字列表現 (「汎用アドレス」) を返します。*nconf* には、アドレスが有効なトランスポートを指定します。汎用アドレスは、free(3C) で解放できます。

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

uaddr が指す「汎用アドレス」は、netbuf 構造体に変換されます。*nconf* には、アドレスが有効なトランスポートを指定します。

```
int netdir_options(struct netconfig *nconf,
   int option, int fd,
   char *point_to_args);
```

netdir_options(3NSL) は、ブロードキャストアドレス、TCP や UDP の予約ポート機能など、トランスポートに固有の機能とインタフェースをとります。*nconf* にはトランスポートを、*option* にはトランスポート固有の動作をそれぞれ指定しま

す。*fd* は、*option* の値次第で指定してもしなくてもかまいません。4 つ目の引数は、操作固有のデータを指します。

表 4–4 に、option に指定できる値を示します。

表 4–4 netdir_options に指定できる値

オプション	説明
ND_SET_BROADCAST	ブロードキャスト用のトランスポートを設定する (トランスポートがブロードキャスト機能をサポートしている場合)
ND_SET_RESERVEDPORT	アプリケーションが予約ポートにバインドできるようにする (トランスポートがそのようなバインドを許可している場合)
ND_CHECK_RESERVEDPORT	アドレスが予約ポートに対応しているかどうかを検証する (トランスポートが予約ポートをサポートしている場合)
ND_MERGEADDR	ローカルに意味のあるアドレスを、クライアントホストが接続できるアドレスに変換する

netdir_perror(3NSL) は、名前からアドレスへのマッピングルーチンの 1 つが失敗した場合に、その理由を示すメッセージを stderr に出力します。

```
void netdir_perror(char *s);
```

netdir_sperror(3NSL) は、名前からアドレスへのマッピングルーチンの 1 つが失敗した場合に、その理由を示すエラーメッセージが含まれた文字列を返します。

```
char *netdir_sperror(void);
```

例 4–7 に、ネットワーク選択および名前からアドレスへのマッピングを示します。

例 4–7 ネットワーク選択および名前からアドレスへのマッピング

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* ホストとサービスの情報 */
struct nd_addrlist *nd_addrlistp; /* サービスのアドレスリスト */
struct netbuf *netbufp;           /* サービスのアドレス */
struct netconfig *nconf;          /* トランスポート情報 */
int i;                            /* アドレスの数 */
char *uaddr;                      /* サービスの汎用アドレス */
```

```
void *handlep;                          /* ネットワーク選択用のハンドル */
/*
 * 「gandalf」というホスト上の「日付」サービスを参照する
 * ホスト構造体の設定
 */
nd_hostserv.h_host = "gandalf";
nd_hostserv.h_serv = "date";
/*
 * ネットワーク選択機構の初期化
 */
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}
/*
 * トランスポートプロバイダ間のループ
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * netconfig 構造体で指定したトランスポートプロバイダに
    * 関連付けられた情報を出力する。
    */
   printf("Transport provider name: %s\n", nconf->nc_netid);
   printf("Transport protocol family: %s\n", nconf->nc_protofmly);
   printf("The transport device file: %s\n", nconf->nc_device);
   printf("Transport provider semantics: ");
    switch (nconf->nc_semantics) {
   case NC_TPI_COTS:
      printf("virtual circuit\n");
      break;
   case NC_TPI_COTS_ORD:
      printf("virtual circuit with orderly release\n");
      break;

   case NC_TPI_CLTS:
      printf("datagram\n");
      break;
   }
   /*
    * netconfig 構造体で指定したトランスポートプロバイダ
    * を経由して、「gandalf」というホスト上の「日付」
    * サービスのアドレスの取得
    */
   if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp) != ND_OK) {
      printf("Cannot determine address for service\n");
      netdir_perror(argv[0]);
      continue;
   }
   printf("<%d> addresses of date service on gandalf:\n",
      nd_addrlistp->n_cnt);
   /*
    * 現在のトランスポートプロバイダ上で、「gandalf」
    * というホスト上にある「日付」サービスの全アドレスの出力
    */
   netbufp = nd_addrlistp->n_addrs;
   for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
      uaddr = taddr2uaddr(nconf,netbufp);
      printf("%s\n",uaddr);
      free(uaddr);
```

```
    }
     netdir_free( nd_addrlistp, ND_ADDRLIST );

}
endnetconfig(handlep);
```

付録 **A**

UNIX ドメインソケット

はじめに

UNIX ドメインソケットには、UNIX パスの名前がつきます。たとえば、ソケット名には `/tmp/foo` などがあります。UNIX ドメインソケットは、単一ホスト上のプロセス間でだけ交信します。UNIX ドメイン上のソケットは、単一ホスト上のプロセス間の交信にしか使用できないため、ネットワークプロトコルの一部とは見なされません。

ソケットタイプでは、ユーザーが認識できる通信プロパティを定義します。インターネットドメインソケットを使用すると、TCP/IP トランスポートプロトコルにアクセスできます。インターネットドメインは、`AF_INET` という値で識別します。ソケットは、同じドメイン内にあるソケットとだけデータをやりとりします。

ソケットの作成

`socket(3SOCKET)` 呼び出しを使用すると、指定のファミリおよび指定のタイプのソケットを作成できます。

```
s = socket(family, type, protocol);
```

プロトコルが未指定 (値が `0`) の場合、要求したソケットタイプをサポートするプロトコルがシステムによって選択され、ソケットハンドル (ファイル記述子) が返されます。

ファミリは、sys/socket.h に定義されている定数の 1 つで指定します。AF_*suite* という定数には、表 2–1 に示されている名前を解釈する際に使用するアドレス形式を指定します。

次のコードでは、マシン内部で使用されるデータグラムソケットを作成します。

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

プロトコルは、通常、デフォルトのもの (*protocol* 引数に 0 を指定する) を使用します。

ローカル名のバインド

ソケットは、その作成時には名前がありません。遠隔プロセスは、ソケットにアドレスがバインドされるまでソケットを参照できません。通信プロセスは、アドレスを介して接続されます。UNIX ファミリでは、接続は、通常、1 つまたは 2 つのパス名からなります。UNIX ファミリのソケットは、必ずしも名前にバインドされる必要はありませんが、バインドされると、local pathname または foreign pathname といった順序セットを複製することができません。パス名では、既存のファイルを参照できません。

bind(3SOCKET) 呼び出しを使用すると、プロセスは、ソケットのローカルアドレスを指定できます。これによって、local pathname が決定します。一方、connect(3SOCKET) と accept(3SOCKET) は、遠隔側アドレスを固定することでソケットの接続を完了します。bind(3SOCKET) は、次のように使用します。

```
bind (s, name, namelen);
```

s は、ソケットハンドルです。バインド名は、バイト文字列であり、サポートするプロトコル (複数も可) がこれを解釈します。UNIX ファミリ名には、パス名とファミリが含まれます。例では、UNIX ファミリソケットに /tmp/foo という名前をバインドしています。

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
  strlen(addr.sun_path) + sizeof (addr.sun_family));
```

AF_UNIX というソケットアドレスの大きさを判定する場合には、ヌル (null) バイトがカウントされません。このため、strlen(3C) の使用をお勧めします。

addr.sun_path で参照されるファイル名は、システムファイル名空間でソケットとして作成されます。呼び出し側は、addr.sun_path が作成されるディレクトリに書き込み許可を持っていなければなりません。このファイルは、不要になったら呼び出し側が削除しなければなりません。AF_UNIX ソケットは、unlink(1M) で削除できます。

接続の確立

通常、接続の確立は非対称に行われます。1 つのプロセスは、クライアントとして機能し、もう一方のプロセスはサーバーとして機能します。サーバーは、サービスに関連付けられた既知のアドレスにソケットをバインドし、接続要求のためにソケットをブロックします。すると、無関係のプロセスがサーバーに接続できます。クライアントは、サーバーのソケットへの接続を起動することでサーバーにサービスを要求します。クライアント側では、connect(3SOCKET) 呼び出しによって接続を起動できます。UNIX ファミリは、これを次のように表現します。

```
struct sockaddr_un server;
  server.sun.family = AF_UNIX;
   ...
  connect(s, (struct sockaddr *)&server, strlen(server.sun_path)
         + sizeof (server.sun_family));
```

接続エラーについては、32ページの「接続エラー」を参照してください。33ページの「データ転送」では、データの転送方法を示します。33ページの「ソケットを閉じる」では、ソケットのクローズ方法を示します。

付録 **B**

実際のコード例

実際のコード例

この付録では、基本的な IPv4 のクライアントおよびサーバーと、クライアントおよびサーバーのさまざまな IPv6 ポートについて例を示して説明します。それぞれの例は、コンパイルと実行ができることがわかっています。各クライアントは、各サーバーと自由に組み合わせて使用できます。これらの例は情報目的のためだけに提供されているので、Sun は、これらの例の使用に対して責任を負いません。

第 1 の例は、オリジナルで無修正の IPv4 クライアント、myconnect です。

例 **B–1** IPv4 クライアント

```
#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;

 /* ソケットを開く */
 sock = socket(AF_INET, SOCK_STREAM, 0);
 if (sock == -1) {
  perror("socket");
  return (-1);
 }
```

```
 /* ホストアドレスの取得 */
 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 */
 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);
}
```

第 2 の例は、IPv4 と IPv6 の両方のサーバーに接続できる、myconnect の最小のポート、myconnect2 です。

例 B–2 IPv4 と IPv6 のクライアントへの最小ポート

```
#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_in6 sin;
 struct hostent  *hp;
 int   sock, errnum;

 /* ソケットを開く */
 sock = socket(AF_INET6, SOCK_STREAM, 0);
 if (sock == -1) {
  perror("socket");
  return (-1);
 }

 /* ホストアドレスの取得。IPv4 射影 IPv6 アドレス可 */
 hp = getipnodebyname(hostname, AF_INET6, AI_DEFAULT, &errnum);
 if (hp == NULL) {
  (void) fprintf(stderr, "Unknown host \"%s\"\n", hostname);
  (void) close(sock);
  return (-1);
 }

 /* すべての sockaddr_in6 フィールドが 0 であることを確認 */
 bzero(&sin, sizeof (sin));
 sin.sin6_family = hp->h_addrtype;
 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 */
 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 = 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);
}
```

第 3 の例は、第 2 の例の拡張版で、1 つが接続するまですべてのアドレスを試行しようとします。これは myconnect2all です。第 3 の例は、同一の機能を持つ第 4 の例よりも複雑であることに注意してください。

例 **B–3**　IPv4 と IPv6 のクライアントへの拡張最小ポート

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int
myconnect2all(char *hostname, int port)
{
 struct sockaddr_in6 sin;
 struct hostent  *hp;
 int   sock, errnum;
 int   h_addr_index;

 /* ソケットを開く */
 sock = socket(AF_INET6, SOCK_STREAM, 0);
 if (sock == -1) {
  perror("socket");
  return (-1);
 }

 /* ホストアドレスの取得。IPv4 射影 IPv6 アドレスが可 */
 hp = getipnodebyname(hostname, AF_INET6, AI_DEFAULT|AI_ALL, &errnum);
 if (hp == NULL) {
```

```
  (void) fprintf(stderr, "Unknown host \"%s\"\n", hostname);
  (void) close(sock);
  return (-1);
 }

 /* すべての sockaddr_in6 フィールドが 0 であることを確認 */
 bzero(&sin, sizeof (sin));
 sin.sin6_family = hp->h_addrtype;
 sin.sin6_port = htons(port);

 /* 1 つが接続するまで戻されたすべてのアドレスを試行 */
 h_addr_index = 0;
 while (hp->h_addr_list[h_addr_index] != NULL) {
  bcopy(hp->h_addr_list[h_addr_index], &sin.sin6_addr,
      hp->h_length);
  /* ホストに接続 */
  if (connect(sock, (struct sockaddr *)&sin,
      sizeof (sin)) == -1) {
   if (hp->h_addr_list[++h_addr_index] != NULL) {
    /* 次のアドレスを試行 */
    continue;
   }
   perror("connect");
   freehostent(hp);
   (void) close(sock);
   return (-1);
  }
  break;
 }
 freehostent(hp);
 return (sock);
}

main(int argc, char *argv[])
{
 int  sock;
 char  buf[BUFSIZ];
 int  cc;

 switch (argc) {
 case 2:
  sock = myconnect2all(argv[1], IPPORT_ECHO);
  break;
 case 3:
  sock = myconnect2all(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);
}
```

第 4 の例は、新しい getaddrinfo(3SOCKET) インタフェースを使用する myconnect のポートです。このルーチンは、1 つが接続するまですべてのアドレスを試行しようとします。これは myconnect3 です。このプログラムは、myconnect2all よりも単純であり、IPv4 から IPv6 へこのポートを実行する方法としてお奨めします。

例 **B–4** IPv6 クライアントへの推奨ポート

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

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

 /* ホストアドレスの取得。どのタイプのアドレスも可 */
 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);
 }
 /* 1 つが接続するまですべてのアドレスを試行 */
 for (aip = res; aip != NULL; aip = aip->ai_next) {
  /*
   * ソケットを開く。このアドレスタイプは、
   * getaddrinfo() の返す値に依存
   */
  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 */
  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);
}
```

第 5 の例は、オリジナルで無修正の IPv4 サーバー、myserver です。

例 **B–5**　IPv4 サーバー

```
#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    /* 保留状態の接続の最大数 */

void do_work(int sock);

int
myserver(int port)
{
 struct sockaddr_in laddr, faddr;
 int   sock, new_sock, sock_opt;
 socklen_t  faddrlen;

 /* 接続待機するソケットの設定 */
 laddr.sin_family = AF_INET;
 laddr.sin_port = htons(port);
 laddr.sin_addr.s_addr = INADDR_ANY;

 sock = socket(AF_INET, SOCK_STREAM, 0);
 if (sock == -1) {
  perror("socket");
  return (-1);
 }

 /* ローカルアドレスを再使用できるようにシステムに通知 */
 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);
 }

 /* 接続要求のための待機 */
 for (;;) {
  faddrlen = sizeof (faddr);
  new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
  if (new_sock == -1) {
   if (errno != EINTR && errno != ECONNABORTED) {
    perror("accept");
   }
   continue;
  }

  (void) printf("Connection from %s/%d\n",
      inet_ntoa(faddr.sin_addr), ntohs(faddr.sin_port));
```

```
  do_work(new_sock);  /* 作業の実行 */
 }
 /*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) myserver(htons(atoi(argv[1])));
 return (0);
}
```

第 6 の例は、IPv4 と IPv6 の両方のクライアントとの接続を処理する myserver の最小ポートです。この例では単一のソケットを使うので、IPv4 接続は IPv4 射影アドレスとして表示されます。

例 B–6　IPv6 サーバーへの最小ポート

```
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    /* 保留状態の接続の最大数 */

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];

 /*
  * 接続待機するソケットの設定
  * すべての sockaddr_in6 フィールドが 0 であることを確認
  */
 bzero(&laddr, sizeof (laddr));
 laddr.sin6_family = AF_INET6;
 laddr.sin6_port = htons(port);
 laddr.sin6_addr = in6addr_any; /* 構造体割り当て */

 sock = socket(AF_INET6, SOCK_STREAM, 0);
 if (sock == -1) {
  perror("socket");
  return (-1);
 }

 /* ローカルアドレスを再使用できるようにシステムに通知 */
 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);
 }

 /* 接続要求のための待機 */
 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);  /* 作業の実行 */
 }
 /*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);
}
```

第 7 の例は、getnameinfo(3SOCKET) を使って対等アドレスと対等名のログを単純化する、IPv6 への myserver のポートです。これは、IPv4 から IPv6 へこのポートを実行する方法としてお奨めします。

例 **B–7**　IPv6 サーバーへの推奨ポート

```
#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    /* 保留状態の接続の最大数 */

void do_work(int sock);

int
myserver3(char *servicename)
{
 struct addrinfo  *aip;
 struct addrinfo  hints;
 struct sockaddr_storage faddr;
 int   sock, new_sock, sock_opt;
 socklen_t  faddrlen;
 int   error;
 char   hname[NI_MAXHOST];
 char   sname[NI_MAXSERV];

 /* 接続待機するソケットの設定 */
 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);
 }

 /* ローカルアドレスを再使用できるようにシステムに通知 */
 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, aip->ai_addr, aip->ai_addrlen) == -1) {
```

```
  perror("bind");
  (void) close(sock);
  return (-1);
 }

 if (listen(sock, BACKLOG) == -1) {
  perror("listen");
  (void) close(sock);
  return (-1);
 }

 /* 接続要求のための待機 */
 for (;;) {
  faddrlen = sizeof (faddr);
  new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
  if (new_sock == -1) {
   if (errno != EINTR && errno != ECONNABORTED) {
    perror("accept");
   }
   continue;
  }
  error = getnameinfo((struct sockaddr *)&faddr, faddrlen,
      hname, sizeof (hname), sname, sizeof (sname),
      NI_NUMERICHOST|NI_NUMERICSERV);
  if (error) {
   (void) fprintf(stderr, "getnameinfo: %s\n",
       gai_strerror(error));
  } else {
   (void) printf("Connection from (addr) %s/%s\n",
       hname, sname);
  }
  error = getnameinfo((struct sockaddr *)&faddr, faddrlen,
      hname, sizeof (hname), sname, sizeof (sname), 0);
  if (error) {
   (void) fprintf(stderr, "getnameinfo: %s\n",
       gai_strerror(error));
  } else {
   (void) printf("Connection from (name) %s/%s\n",
       hname, sname);
  }
  do_work(new_sock);  /* 作業の実行 */
 }
 /*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 <servicename>\n", argv[0]);
  exit(1);
 }
 (void) myserver3(argv[1]);
 return (0);
}
```

第 8 の例は、射影アドレスの代わりに IPv4 と IPv6 用の個別のソケットを使用する myserver のポートです。この設定は poll(2) を使う必要があるので、さらに複雑です。

例 **B–8** IPv4 ソケットと IPv6 ソケットを使った myserver のポート

```
#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    /* 保留状態の接続の最大数 */
#define NUMSOCKS 2   /* ポーリングするソケット数 */

void do_work(int sock);

int
myserver4(int port)
{
 struct sockaddr_in laddr4, faddr4;
 struct sockaddr_in6 laddr6, faddr6;
 int   sock4, sock6, new_sock, sock_opt;
 socklen_t  faddrlen;
 char   addrbuf[INET6_ADDRSTRLEN];
 struct pollfd  pfd[NUMSOCKS];

 /*
  * 接続待機するソケットの設定。1 つは IPv4 に、
  * もう 1 つは IPv6 に。
  */
```

```
laddr4.sin_family = AF_INET;
laddr4.sin_port = htons(port);
laddr4.sin_addr.s_addr = INADDR_ANY;

sock4 = socket(AF_INET, SOCK_STREAM, 0);
if (sock4 == -1) {
 perror("socket");
 return (-1);
}

/* すべての sockaddr_in6 フィールドが 0 であることを確認 */
bzero(&laddr6, sizeof (laddr6));
laddr6.sin6_family = AF_INET6;
laddr6.sin6_port = htons(port);
laddr6.sin6_addr = in6addr_any; /* 構造体割り当て */

sock6 = socket(AF_INET6, SOCK_STREAM, 0);
if (sock6 == -1) {
 perror("socket");
 (void) close(sock4);
 return (-1);
}

/* ローカルアドレスを再使用できるようにシステムに通知 */
sock_opt = 1;
if (setsockopt(sock4, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,
    sizeof (sock_opt)) == -1 ||
    setsockopt(sock6, SOL_SOCKET, SO_REUSEADDR, (void *)&sock_opt,
    sizeof (sock_opt))) {
 perror("setsockopt(SO_REUSEADDR)");
 (void) close(sock4);
 (void) close(sock6);
 return (-1);
}

if (bind(sock4, (struct sockaddr *)&laddr4, sizeof (laddr4)) == -1 ||
    bind(sock6, (struct sockaddr *)&laddr6, sizeof (laddr6)) == -1) {
 perror("bind");
 (void) close(sock4);
 (void) close(sock6);
 return (-1);
}

if (listen(sock4, BACKLOG) == -1 || listen(sock6, BACKLOG) == -1) {
 perror("listen");
 (void) close(sock4);
 (void) close(sock6);
 return (-1);
}

/* 接続要求のための待機 */
pfd[0].fd = sock4;
pfd[1].fd = sock6;
pfd[0].events = pfd[1].events = POLLIN;

for (;;) {
 if (poll(pfd, NUMSOCKS, -1) == -1) {
  if (errno != EINTR) {
   perror("poll");
   (void) close(sock4);
```

```
    (void) close(sock6);
    return (-1);
   }
   continue;
  }

  if (pfd[0].revents & POLLIN) {  /* IPv4 */
   faddrlen = sizeof (faddr4);
   new_sock = accept(sock4, (struct sockaddr *)&faddr4,
       &faddrlen);
   if (new_sock == -1) {
    if (errno != EINTR && errno != ECONNABORTED) {
     perror("accept");
    }
    continue;
   }
   (void) printf("Connection from %s/%d\n",
       inet_ntop(AF_INET, (void *)&faddr4.sin_addr,
       addrbuf, sizeof (addrbuf)),
       ntohs(faddr4.sin_port));
   do_work(new_sock);  /* 作業の実行 */
  }

  if (pfd[1].revents & POLLIN) {  /* IPv6 */
   faddrlen = sizeof (faddr6);
   new_sock = accept(sock6, (struct sockaddr *)&faddr6,
       &faddrlen);
   if (new_sock == -1) {
    if (errno != EINTR && errno != ECONNABORTED) {
     perror("accept");
    }
    continue;
   }
   (void) printf("Connection from %s/%d\n",
       inet_ntop(AF_INET6, (void *)&faddr6.sin6_addr,
       addrbuf, sizeof (addrbuf)),
       ntohs(faddr6.sin6_port));
   do_work(new_sock);  /* 作業の実行 */
  }
 }
 /*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) myserver4(htons(atoi(argv[1])));
 return (0);
}
```

この make ファイルは、この付録の例の一部またはすべてをコンパイルおよびリンクします。また、lint も行います。

```
SRC  = myconnect.c myconnect2.c myconnect3.c myconnect2all.c \
    myserver.c myserver2.c myserver3.c myserver4.c

EXES  = $(SRC:%.c=%)
LINTOUT  = $(SRC:%.c=%.lint)

#TOOLSDIR = /ws/on28-tools/SUNWspro/SC5.0/bin
#CC       = $(TOOLSDIR)/cc
#LINT.c   = $(TOOLSDIR)/lint

CFLAGS   = -g
LDLIBS   = -lsocket -lnsl
LFLAGS   = $(CFLAGS)
LINTOPTS = $(LFLAGS) $(LIBS)

all: $(EXES)
lint: $(LINTOUT)

myconnect.o: myconnect.c
 $(CC) -c -o $@ $(CFLAGS) myconnect.c

myconnect2.o: myconnect2.c
 $(CC) -c -o $@ $(CFLAGS) myconnect2.c

myconnect2all.o: myconnect2all.c
 $(CC) -c -o $@ $(CFLAGS) myconnect2all.c

myconnect3.o: myconnect3.c
 $(CC) -c -o $@ $(CFLAGS) myconnect3.c

myserver.o: myserver.c
 $(CC) -c -o $@ $(CFLAGS) myserver.c

myserver2.o: myserver2.c
```

```
 $(CC) -c -o $@ $(CFLAGS) myserver2.c

myserver3.o: myserver3.c
 $(CC) -c -o $@ $(CFLAGS) myserver3.c

myconnect.lint:
 $(LINT.c) $(LINTOPTS) myconnect.c

myconnect2.lint:
 $(LINT.c) $(LINTOPTS) myconnect2.c

myconnect2all.lint:
 $(LINT.c) $(LINTOPTS) myconnect2all.c

myconnect3.lint:
 $(LINT.c) $(LINTOPTS) myconnect3.c

myserver.lint:
 $(LINT.c) $(LINTOPTS) myserver.c

myserver2.lint:
 $(LINT.c) $(LINTOPTS) myserver2.c

myserver3.lint:
 $(LINT.c) $(LINTOPTS) myserver3.c

myserver4.lint:
 $(LINT.c) $(LINTOPTS) myserver4.c

clean:
 rm -f *.o core
```

索引

O

P

R

S

T

U

X

い

か

く

こ

さ

す

せ

そ