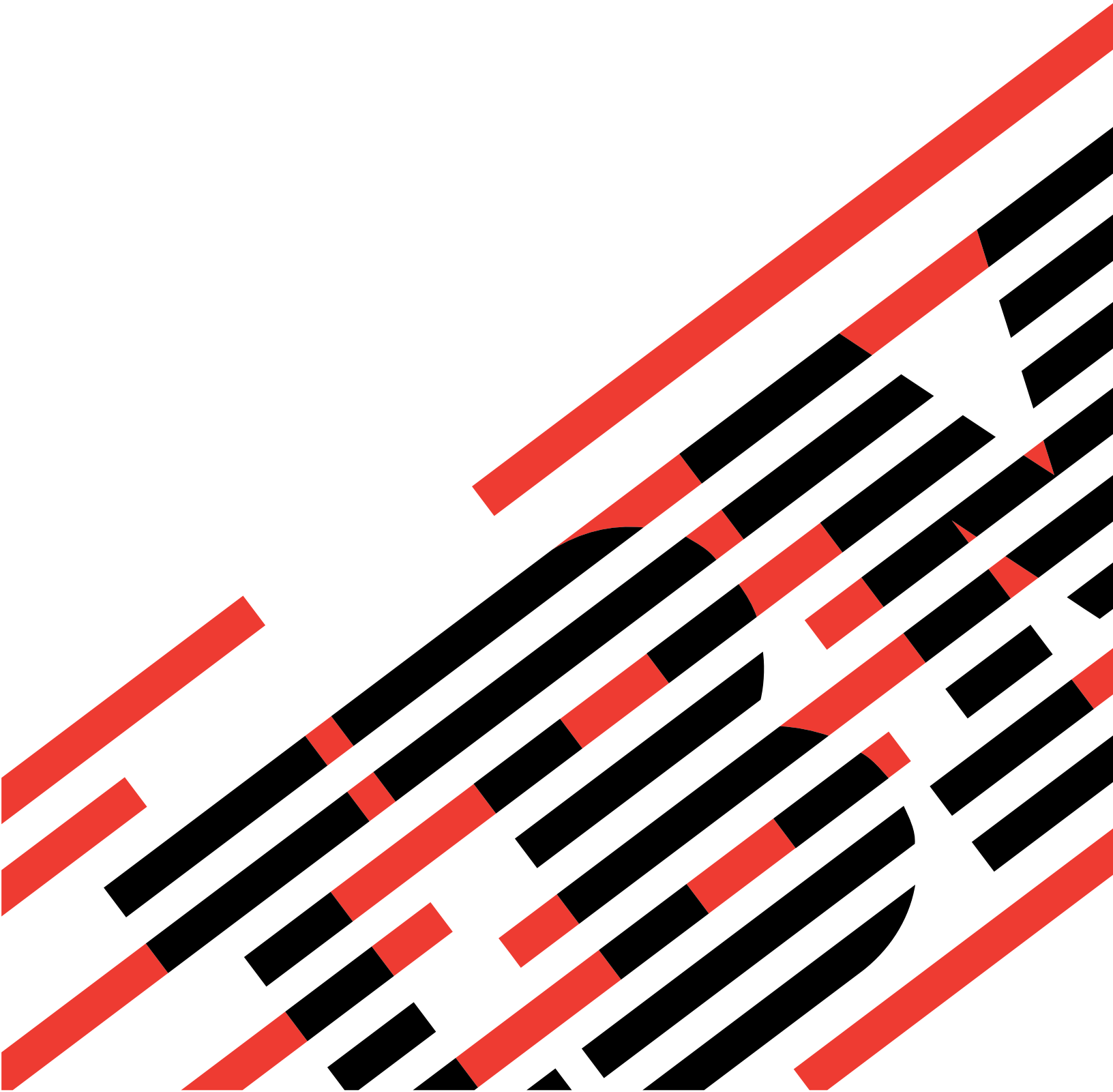
IBM

eserver

iSeries

ソケット・プログラミング

バージョン 5 リリース 3



IBM

@server

iSeries

ソケット・プログラミング

バージョン 5 リリース 3

ご注意

本書および本書で紹介する製品をご使用になる前に、 191 ページの『特記事項』に記載されている情報をお読みください。

本書は、Operating System/400® (5722-SS1) のバージョン 5、リリース 3、モディフィケーション 0、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また CISC モデルでは稼働しません。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

`http://www.ibm.com/jp/manuals/main/mail.html`

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

`http://www.ibm.com/jp/manuals/` の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原　典： iSeries
Socket programming
Version 5 Release 3

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2005.8

この文書では、平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™ W7を使用しています。この(書体*)は、(財）日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、平成角ゴシック体™W7

目次

ソケット・プログラミング

ソケットとは、ネットワーク上で名前付けやアドレス指定が可能な通信接続ポイント (端点) のことです。ソケットを使用するプロセスは、同じシステムまたは異なるネットワークの異なるシステムに置くことができます。ソケットは、スタンドアロン・アプリケーションでもネットワーク・アプリケーションでも有効です。ソケットを使用すれば、同一マシン上の、またはネットワークを介した複数のプロセス間で、情報を交換したり、最も効率のよいマシンに作業を配布することができ、中央データに簡単にアクセスすることもできます。ソケット・アプリケーション・プログラム・インターフェース (API) は、TCP/IP のネットワーク標準です。さまざまな種類のオペレーティング・システムが、ソケット API をサポートしています。OS/400® ソケットは、多重トランスポートとネットワーク・プロトコルをサポートしています。ソケット・システム関数とソケット・ネットワーク関数はスレッド・セーフです。

「ソケット・プログラミング」では、ソケット API を使用して、リモート・プロセスとローカル・プロセスの間に通信リンクを確立する方法について説明します。統合化言語環境® (ILE) C を使用するプログラマーは、この情報を使用してソケット・アプリケーションを作成することができます。 RPG などの他の ILE 言語を使って、ソケット API でコーディングすることもできます。ILE RPG の詳細については、IBM® レッドブック Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More. を参照してください。

ソケット・プログラミング・インターフェースは、Java™ でもサポートされています。詳細については、Information Center で Java に関するトピックを参照してください。

ソケット・プログラミングについてのトピック

以下のトピックで、ソケット・アプリケーションの開発に役立つ概念、設計の推奨事項、および例について説明します。以下を参照してください。

- **トピックの印刷**
  このページは、ソケット・プログラミングに関する情報の PDF 版を印刷またはダウンロードするために使用します。
- **ソケット・プログラミングの前提条件**
  このトピックでは、ソケット API を使ってアプリケーションを作成する前に完了しておく必要のある必須タスクについて説明します。
- **ソケットの基本設計**
  このトピックでは、ソケットの最も基本的なタイプ用のサンプル・プログラムを概説します。ソケットの基本設計戦略を説明する例については、サンプル・プログラムへのリンクを使用してください。
- **ソケットの概念**
  このトピックでは、非同期入出力 (I/O) やグローバル・セキュア・ツールキット (GSkit) などのさらに高度なソケット概念について説明します。トピック内のリンクを使用して、これらの概念に関連したサンプル・プログラムを参照してください。
- **ソケットのシナリオ: IPv4 クライアントと IPv6 クライアントを受け入れるアプリケーションの作成**
  このトピックでは、AF_INET6 アドレス・ファミリーを使用することのできる典型的な状況について説明します。V5R2 から、このアドレス・ファミリーはインターネット・プロトコル バージョン 6 (IPv6) をサポートするようになりました。IPv6 は 128 ビット IP アドレスをサポートします。このトピックに

は計画情報が載せられています。また、プログラム例へのリンクもあり、AF_INET6 アドレス・ファミリーを使用するソケット・アプリケーションを実装する場合に使用できます。

- **ソケット・アプリケーション設計の推奨事項**
  このトピックでは、より効果的なソケット・アプリケーションを設計するためのヒントを提供します。
- **例: ソケット・アプリケーション設計**
  このトピックでは、ソケット・アプリケーションを作成するために使用できるサンプル・ソケット・プログラムを提供します。

  注: 本書はサンプル・コードを含んでいます。これらのサンプル・プログラムの使用に関する詳細については、『コードの特記事項情報』を参照してください。
- **Xsocket ツール**
  このトピックでは、Xsocket ツールについて説明します。ソケット・プログラマーは、ソケット・アプリケーションの開発にこのツールを役立てることができます。トピック内のリンクを使用して、このツールのインストールと使用方法について説明を参照してください。
- **保守容易性ツール**
  このトピックでは、ソケットの保守容易性のツールについて説明します。
- **関連情報**
  このトピックでは、ソケットに関する他の情報へのリンクおよび説明を提供します。

トピックの印刷

この文書の PDF 版を参照用または印刷用にダウンロードし、表示することができます。 PDF ファイルを表示したり印刷したりするには、Adobe® Acrobat® Reader が必要です。これは、 Adobe Web サイト

http://www.adobe.com/prodindex/acrobat/readstep.html から、ダウンロードできます。

PDF 版をダウンロードし、表示するには、『ソケット・プログラミング』を選択します (約 1,345 KB、202 ページ)。

表示用または印刷用の PDF ファイルをワークステーションに保存するには、次のようにします。

1. ブラウザーで PDF を開く (上記のリンクをクリックする)。
2. ブラウザーのメニューから「**ファイル**」をクリックする。
3. 「**名前を付けて保存**」をクリックする。
4. PDF を保存したいディレクトリーに進む。
5. 「**保存**」をクリックする。

ソケット・プログラミングの前提条件

ソケット・アプリケーションを作成する前に、以下のステップを完了する必要があります。

コンパイラーの要件

1. QSYSINC ライブラリーをインストールする。このライブラリーは、ソケット・アプリケーションのコンパイル時に必要なヘッダー・ファイルを提供します。
2. C Compiler ライセンス・プログラム (5722-CX2) をインストールする。

AF_INET および AF_INET6 アドレス・ファミリーの要件

コンパイラーの要件を満たすことに加えて、次の事柄を実行する必要があります。

1. TCP/IP 計画。
2. TCP/IP のインストール。
3. 最初に行う TCP/IP の構成。
4. IPv6 用の TCP/IP の構成。このステップはオプションです。AF_INET6 アドレス・ファミリーを使用するアプリケーションを作成する予定の場合、TCP/IP 用に IPv6 インターフェースを構成してください。

Secure Sockets Layer (SSL) およびグローバル・セキュア・ツールキット (GSKit) API の要件

コンパイラーおよび AF_INET、AF_INET6 アドレスの要件を満たすことに加え、セキュア・ソケットを処理するためには以下のタスクを実行する必要があります。
1. ディジタル証明書マネージャー・ライセンス・プログラム (5722-SS1 オプション 34) をインストールして構成する。詳細については、Information Center の『ディジタル証明書管理』を参照してください。
2. Cryptographic Access Provider ライセンス・プログラム (5722-AC3) をインストールする。
3. 暗号化ハードウェアを使って SSL を利用する場合は、2058 Cryptographic Accelerator for iSeries™ または 4758 PCI 暗号化コプロセッサーのどちらかをインストールして構成する。2058 Cryptographic Accelerator を使用すると、SSL 暗号化処理をカードにオフロードして、オペレーティング・システムの負荷を軽減できます。2058 Cryptographic Accelerator とそのフィーチャーの詳細な説明は、『2058 暗号化アクセラレーター』を参照してください。4758 暗号化コプロセッサーも SSL 暗号化処理に用いることができますが、2058 とは異なり、このカードは暗号化鍵や復号鍵のような、より暗号化に特化した機能を提供します。このカードのフィーチャーと構成のステップに関しては、『4758 PCI 暗号化コプロセッサー』を参照してください。

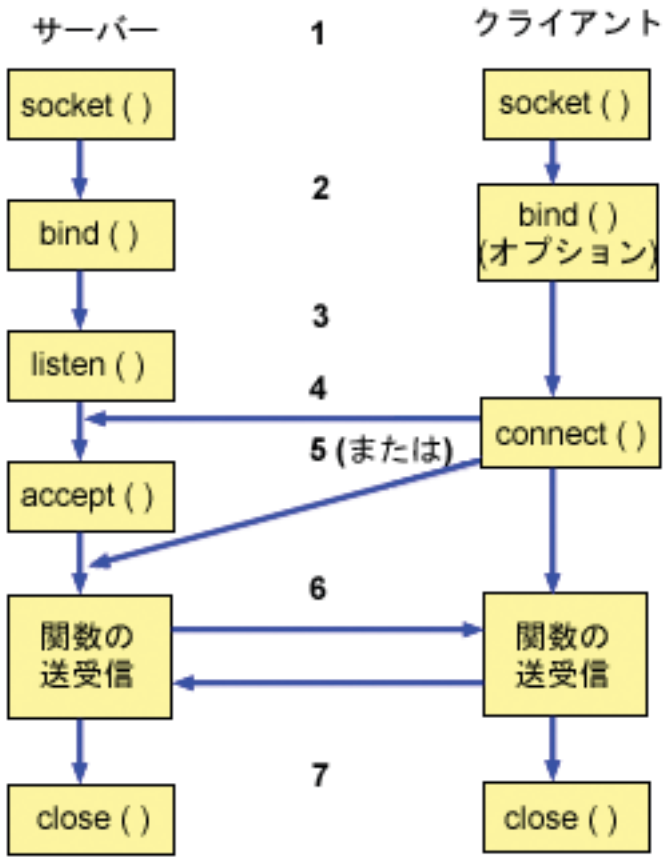
ソケットの仕組み

ソケットは一般にクライアント/サーバーの対話で使用されます。通常のシステム構成では、一方のマシンにサーバーを、もう一方のマシンにクライアントを置きます。クライアントはサーバーに接続して情報を交換し、その後切断します。

ソケットには定型のイベント・フローがあります。コネクション型クライアント/サーバー・モデルでは、サーバー・プロセス上のソケットはクライアントからの要求を待ちます。これを行うため、サーバーはまず、クライアントがサーバーを探せるようにアドレスを確立 (バインド) します。アドレスが確立されると、サーバーはクライアントがサービスを要求してくるのを待ちます。クライアントとサーバーとの間のデータ交換は、クライアントがソケットを経由してサーバーに接続しているときに行われます。サーバーは、クライアントの要求を実行し、クライアントに応答を送信し返します。

注: 現在、IBM は、大部分のソケット API について、2 つのバージョンをサポートしています。デフォルトの OS/400 ソケットは、バークレー・ソケット・ディストリビューション (BSD) 4.3 の構造と構文を使用します。基本 OS/400 ソケットと BSD 4.3 の相違点は、『バークレー・ソケット・ディストリビューション (BSD) との互換性』に概説されています。もう一方のバージョンのソケットは、BSD 4.4 および UNIX® 98 プログラミング・インターフェース仕様と互換性のある構文および構造を使用します。プログラマーは、_XOPEN_SOURCE マクロを指定することにより、UNIX98 と互換性のあるインターフェースを使用することができます。これらの API および構造上の相違点については、『UNIX 98 互換性』を参照してください。

以下の図は、コネクション型ソケット・セッションの典型的なイベント・フロー (および関数が発行される順序) を表しています。各イベントの説明が、図の後に続きます。

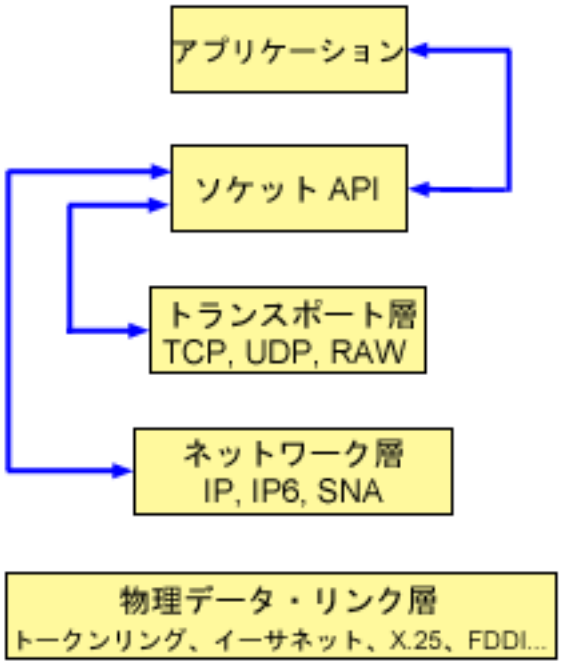


コネクション型ソケットの一般的なイベント・フロー

1. **socket()** 関数は、通信用の端点を作成し、端点を表すソケット記述子を戻します。
2. アプリケーションがソケット記述子をもつと、アプリケーションはソケットに固有な名前をバインドできます。サーバーは、ネットワークからのアクセスを可能にするために、名前をバインドする必要があります。
3. **listen()** 関数は、クライアントの接続要求を受け入れる態勢を示しています。 **listen()** がソケットに対して発行されると、そのソケットは接続要求を積極的には始めません。 **listen()** API は、あるソケットが **socket()** 関数で割り当てられた後で、かつ **bind()** 関数でそのソケットに名前がバインドされた後に発行されます。 **listen()** 関数は **accept()** 関数の発行前に発行されていなければなりません。
4. ストリーム・ソケットの **connect()** 関数は、クライアント・アプリケーションがサーバーへの接続を確立するのに使用されます。
5. サーバー・アプリケーションは **accept()** 関数を使用して、クライアント接続要求を受け入れます。サーバーは、**accept()** を発行する前に、**bind()** 関数と **listen()** 関数を正常に発行している必要があります。
6. ストリーム・ソケット間 (クライアントとサーバーの間) に接続が確立されると、ソケット API データ転送関数をどれでも使用できるようになります。クライアントとサーバーには、選択可能な多くのデータ転送関数があります。たとえば、**send()**、**recv()**、**read()**、**write()** などです。
7. サーバーまたはクライアントが操作を終了したい場合は、ソケットが獲得したシステム・リソースを解放するために **close()** 関数を発行する必要があります。

注:

ソケット API は、通信モデルの中でアプリケーション層とトランスポート層の間に位置します。ソケット API は、通信モデルの中の層ではありません。ソケット API を使用することにより、アプリケーションは一般的な通信モデルのトランスポート層やネットワーク層と対話できます。以下の図の矢印は、ソケットの位置とソケットが提供する通信層とを示しています。



一般に、ネットワーク構成では、セキュア内部ネットワークと非セキュア外部ネットワークとを接続することはできません。しかし、ソケットがファイアウォール (高度なセキュア・ホスト) の外部にあるシステムのサーバー・プログラムと通信できるようにすることもできます。

ソケットはマルチプロトコル・トランスポート・ネットワーキング (MPTN) 体系を支える、IBM の AnyNet® 実装の一部でもあります。MPTN 体系は、追加のトランスポート・ネットワークの中から 1 つのトランスポート・ネットワークを操作できるようにしたり、異なるタイプのトランスポート・ネットワーク間でもアプリケーション・プログラムを接続したりできるようにします。

Information Center では、何通りかの方法で API 参照情報にアクセスできます。トピック『Socket API』では、ソケット関数と構造に関する概要を示しています。特定の API を検索する場合や、API のカテゴリーを検索する場合に備えて、API 情報には対話型の API ファインダー (API finder) が用意されています。

ソケットの特性

ソケットは以下の特性を共有します。

- ソケットは、整数によって表されます。その整数のことを**ソケット記述子**といいます。
- ソケットは、プロセスがソケットへのオープン・リンクを保持している間、存在します。
- 通信ドメイン内では、特定のソケットを指定し、それを他のソケットとの通信に使用することができます。

- ソケットが通信を行うのは、サーバーがソケットからの接続を受け入れるとき、またはサーバーがソケットとメッセージを交換するときです。
- ソケットは対で作成できます (AF_UNIX アドレス・ファミリーのソケットのみ)。

ソケットが提供する通信の種類には、コネクション型とコネクションレス型があります。**コネクション型**通信とは接続が確立されており、プログラム間の対話が続いて行われるものです。サービスを提供するプログラム (サーバー・プログラム) が、着信接続要求を受け入れることができる使用可能なソケットを設定します。オプションで、サーバーは提供するサービスに名前を割り当てることができます。これによってクライアントは、サービスの取得先とそのサービスへの接続方法を識別できます。サービスのクライアント (クライアント・プログラム) は、サーバー・プログラムのサービスを要求しなければなりません。クライアントは、固有名に接続することによって、またはサーバー・プログラムが指定した固有名に関連した属性に接続することによって、このことを行います。これは、電話番号 (ID) をダイヤルし、サービスを提供する相手 (たとえば、プロバイダー) につなげるようなものです。呼の受信側 (サーバー、この例ではプロバイダー) が電話に応答すれば、接続は確立します。プロバイダーは、クライアントが正しい相手に接続できたかどうかを確認でき、その接続は通話している両者が必要とする限り保たれます。

コネクションレス型通信とは、対話またはデータの転送を行う上での接続が確立されていないものです。代わりに、サーバー・プログラムが受信先の名前 (私書箱によく似ている) を指定します。私書箱に手紙を出しても、相手がその手紙を受け取るという絶対的な保証はありません。返信の手紙を待つ必要があることでしょう。同じようにこの通信では、データの交換で、アクティブなリアルタイム接続はありません。

ソケット特性の判別方法

アプリケーションは **socket()** 関数を用いてソケットを作成する場合、以下のパラメーターを指定してソケットを識別しなければなりません。

- ソケットのアドレス・ファミリーによって、ソケットのアドレス構造の形式が決まります。このトピックには、各アドレス・ファミリーのアドレス構造の例があります。ソケット・アドレスの構造の一般定義については、『ソケットのアドレス構造』を参照してください。
- ソケット・タイプによって、ソケットの希望する通信形式が決まります。
- ソケット・サポート・プロトコルによって、ソケットが使用するプロトコルが決まります。

これらのパラメーターまたは特性によって、ソケット・アプリケーションと、それが他のソケット・アプリケーションと相互運用する方法とが定義されます。ソケットのアドレス・ファミリーによって、異なるソケット・タイプおよびソケット・プロトコルを選択できます。以下の表は、対応するアドレス・ファミリーとそれに関連したソケット・タイプおよびソケット・プロトコルを示しています。

表 1. ソケット特性の要約

アドレス・ファミリー	ソケット・タイプ	ソケット・プロトコル
AF_UNIX	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP、ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6、ICMP6
AF_TELEPHONY	SOCK_STREAM	N/A

表 1. ソケット特性の要約 (続き)

AF_UNIX_CCSID	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A

これらのソケット特性またはパラメーターに加えて、QSYSINC ライブラリーに付属のネットワーク・ルーチンとヘッダー・ファイルには、定数値が定義されています。ヘッダー・ファイルの説明については、Information Center の『Socket API』にリストされている個別の API を参照してください。各 API では、API の説明の使用法のセクションに適切なヘッダー・ファイルがリストされています。

ソケット・ネットワーク・ルーチンを使用して、ソケット・アプリケーションは、DNS、ホスト、プロトコル、サービス、およびネットワーク・ファイルから情報を獲得することができます。これらのルーチンの説明については、『ソケット・ネットワーク関数』を参照してください。

ソケットのアドレス構造

ソケットは **sockaddr** アドレス構造を使用してアドレスの受け渡しを行います。この構造では、アドレス形式を識別するためのソケット API は必要ありません。現在、OS/400 はバークレー・ソフトウェア・ディストリビューション (BSD) 4.3 および X/Open Single Unix Specification (UNIX 98) をサポートしています。基本 OS/400 API は、BSD 4.3 の構造と構文を使用します。 _XOPEN_SOURCE マクロの値を 520 以上に定義すると、UNIX 98 互換のインターフェースを選択できます。使用される BSD 4.3 のソケットの各アドレス構造が、UNIX 98 の構造に対応するものになります。

表 2. BSD 4.3 と UNIX 98/BSD 4.4 のソケット・アドレス構造の比較

BSD 4.3 構造	**BSD 4.4/UNIX 98 互換の構造**

`struct sockaddr{`
`  u_short  sa_family;`
`  char  sa_data [14];`
`};`
 	
`struct sockaddr_storage{`
`  sa_family_t  ss_family;`
`  char  _ss_pad1[_SS_PAD1SIZE];`
`  char*  _ss_align;`
`  char  _ss_pad2[_SS_PAD2SIZE];`
`};`
	`struct sockaddr {`
	`  uint8_t  sa_len;`
	`  sa_family_t  sa_family`
	`  char  sa_data[14]`
	`};`
	
	`struct sockaddr_storage {`
	`  uint8_t  ss_len;`
	`  sa_family_t  ss_family;`
	`  char  _ss_pad1[_SS_PAD1SIZE];`
	`  char*  _ss_align;`
	`  char  _ss_pad2[_SS_PAD2SIZE];`
	`};`

表 3. アドレス構造

アドレス構造フィールド	**定義**
sa_len	このフィールドには UNIX 98 仕様のアドレス長が入ります。 **注:** sa_len フィールドは、BSD 4.4 との互換性を保つために存在しているに過ぎません。BSD 4.4/UNIX 98 互換性を使用する場合であっても、このフィールドを使用する必要はありません。入力アドレスの場合、このフィールドは無視されます。
sa_family	このフィールドには、アドレス・ファミリーを定義します。これは、**socket()** 呼び出しでアドレス・ファミリーに指定される値です。

表 3. アドレス構造 (続き)

sa_data	このフィールドには、アドレスの保持用に確保されている 14 バイトが入ります。 **注:** sa_data の 14 バイトの長さというのはアドレスのプレースホルダー分です。アドレスはこの長さを超えてしまう場合があります。この構造はアドレスの形式を定義しないので汎用性があります。アドレスの形式はトランスポートのタイプによって定義され、ソケットはトランスポートのタイプに合わせて作成されます。それぞれのトランスポート・プロバイダーは、固有のアドレス要件に適した形式を、類似のアドレス構造で定義します。トランスポートは、**socket()** API のプロトコル・パラメーター値によって識別されます。
sockaddr_storage	あらゆるアドレス・ファミリーのアドレスのためのストレージを宣言します。この構造は、プロトコル特有のどのような構造に対しても大きさが十分で、位置合わせも正しく行われます。API で使用するために sockaddr 構造にキャストされる場合もあります。sockaddr_storage の ss_family フィールドは、プロトコル特有のどのような構造のファミリー・フィールドとも、必ず正しく位置合わせされます。

ソケットのアドレス・ファミリー

socket() のアドレス・ファミリー・パラメーターによって、ソケット関数で使用するアドレス構造の形式が決まります。アドレス・ファミリー・プロトコルは、ネットワーク内のあるアプリケーションから別のアプリケーションへ (または、同じマシン内のあるプロセスから別のプロセスへ) アプリケーション・データを移送できるようにします。アプリケーションは、ソケットのプロトコル・パラメーターにネットワーク・トランスポート・プロバイダーを指定します。

socket() 関数のアドレス・ファミリー・パラメーター (address_family) は、ソケット関数で使用されるアドレス構造を指定します。以下のトピックでは、これらのアドレス・ファミリー、その使用方法、関連プロトコル、および関係する構造の例について説明しています。

- AF_INET アドレス・ファミリー
- AF_INET6 アドレス・ファミリー
- AF_UNIX アドレス・ファミリー
- AF_UNIX_CCSID アドレス・ファミリー
- AF_TELEPHONY アドレス・ファミリー

AF_INET アドレス・ファミリー

このアドレス・ファミリーは、同一システムまたは異なるシステム上で実行される複数のプロセス間で、プロセス間通信を行えるようにします。AF_INET ソケットのアドレスは、IP アドレスおよびポート番号です。AF_INET ソケットの IP アドレスは、IP アドレス (130.99.128.1 など) または 32 ビット形式 (X'82638001') のどちらかで指定します。

インターネット・プロトコル バージョン 4 (IPv4) を使用するソケット・アプリケーションの場合、AF_INET アドレス・ファミリーは **sockaddr_in** アドレス構造を使用します。_XOPEN_SOURCE マクロを使用すると、AF_INET アドレス構造は BSD 4.4/UNIX 98 仕様と互換性を持つように変化します。以下の表に sockaddr_in アドレス構造の相違点を要約します。

表 4. BSD 4.3 と BSD 4.4/UNIX 98 の間の sockaddr_in アドレス構造の相違点

BSD 4.3 の sockaddr_in アドレス構造	**BSD 4.4/UNIX 98 の sockaddr_in アドレス構造**

`struct sockaddr_in {`
`  short  sin_family;`
`  u_short  sin_port;`
`  struct in_addr sin_addr;`
`  char  sin_zero[8];`
`};`
	`struct sockaddr_in {`
	`  uint8_t  sin_len;`
	`  sa_family_t  sin_family;`
	`  u_short  sin_port;`
	`  struct in_addr  sin_addr;`
	`  char  sin_zero[8];`
	`};`

表 5. AF_INET アドレス構造

アドレス構造フィールド	**定義**
sin_len	このフィールドには UNIX 98 仕様のアドレス長が入ります。 **注:** sin_len フィールドは、BSD 4.4 との互換性を保つために存在しているに過ぎません。BSD 4.4/UNIX 98 互換性を使用する場合であっても、このフィールドを使用する必要はありません。入力アドレスの場合、このフィールドは無視されます。
sin_family	このフィールドにはアドレス・ファミリーが入りますが、これは TCP または UDP が使用されるときは常に AF_INET です。
sin_port	このフィールドにはポート番号が入ります。
sin_addr	このフィールドには IP アドレスが入ります。
sin_zero	このフィールドは予約済みです。このフィールドは 16 進数のゼロに設定してください。

AF_INET の使用法と AF_INET アドレス・ファミリーを使用するサンプル・プログラムについては、『AF_INET アドレス・ファミリーの使用』を参照してください。

AF_INET6 アドレス・ファミリー

このアドレス・ファミリーは、インターネット・プロトコル バージョン 6 (IPv6) をサポートします。AF_INET6 アドレス・ファミリーは、128 ビット (16 バイト) のアドレスを使用します。このアドレスの基本体系には、64 ビットのネットワーク番号と、64 ビットのホスト番号が組み込まれています。AF_INET6 アドレスは、x:x:x:x:x:x:x:x という形式で指定できます。8 個の 'x' は、それぞれ 16 ビットのアドレスを 16 進値で表したものです。たとえば、FEDC:BA98:7654:3210:FEDC:BA98:7654:3210 は有効なアドレスです。

TCP、UDP または RAW を使用するソケット・アプリケーションの場合、AF_INET6 アドレス・ファミリーは、**sockaddr_in6** アドレス構造を使用します。BSD 4.4/UNIX 98 仕様を実装するために _XOPEN_SOURCE マクロを使用する場合、このアドレス構造は変化します。以下の表に sockaddr_in6 アドレス構造の相違点を要約します。

表 6. BSD 4.3 と BSD 4.4/UNIX 98 の間の sockaddr_in6 アドレス構造の相違点

BSD 4.3 の sockaddr_in6 アドレス構造	BSD 4.4/UNIX 98 の sockaddr_in6 アドレス構造

struct sockaddr_in6 {
sa_family_t sin6_family;
in_port_t sin6_port;
uint32_t sin6_flowinfo;
struct in6_addr sin6_addr;
uint32_t sin6_scope_id;
};
	struct sockaddr_in6 {
	uint8_t sin6_len;
	sa_family_t sin6_family;
	in_port_t sin6_port;
	uint32_t sin6_flowinfo;
	struct in6_addr sin6_addr;
	uint32_t sin6_scope_id;
	};

表 7. AF_INET6 アドレス構造

アドレス構造フィールド	定義
sin6_len	このフィールドには UNIX 98 仕様のアドレス長が入ります。 **注:** sin6_len フィールドは、BSD 4.4 との互換性を保つために存在しているに過ぎません。BSD 4.4/UNIX 98 互換性を使用する場合であっても、このフィールドを使用する必要はありません。入力アドレスの場合、このフィールドは無視されます。
sin6_family	このフィールドには、AF_INET6 アドレス・ファミリーを指定します。
sin6_port	このフィールドには、トランスポート層ポートが入ります。
sin6_flowinfo	このフィールドには、トラフィック・クラスとフロー・ラベルという 2 つの情報が入ります。 **注:** このフィールドは現在サポートされていません。上位互換性のため、ゼロに設定してください。
sin6_addr	このフィールドには、IPv6 アドレスを指定します。
sin6_scope_id	このフィールドは、**sin6_addr** フィールドに入れられるアドレスの有効範囲に適切な、一連のインターフェースを指定します。 **注:** このフィールドは現在サポートされていません。上位互換性のため、ゼロに設定してください。

AF_UNIX アドレス・ファミリー

このアドレス・ファミリーは、ソケット API を使用する同一システムでプロセス間通信を行えるようにします。このアドレスは、ファイル・システムの項目への実際のパス名になります。ソケットは、ルート・ディレクトリー内、またはすべてのオープン・ファイル・システムで作成できます。ただし、QSYS または QDOC などのファイル・システムは除きます。プログラムは、データグラムを受け取るために、AF_UNIX、SOCK_DGRAM ソケットを名前に結合する必要があります。さらに、プログラムは **unlink()** API を使用して、ソケットのクローズ時にファイル・システム・オブジェクトを明示的に除去する必要があります。

アドレス・ファミリー AF_UNIX を指定したソケットは、**sockaddr_un** アドレス構造を使用します。BSD 4.4/UNIX 98 仕様を実装するために _XOPEN_SOURCE マクロを使用する場合、このアドレス構造は変化します。以下の表に sockaddr_un アドレス構造の相違点を要約します。

表 8. BSD 4.3 と BSD 4.4/UNIX 98 の間の sockaddr_un アドレス構造の相違点

BSD 4.3 の sockaddr_un アドレス構造	BSD 4.4/UNIX 98 の sockaddr_un アドレス構造

struct sockaddr_un {
short sun_family;
char sun_path[126];
};
	struct sockaddr_un {
	uint8_t sun_len;
	sa_family_t sun_family;
	char sun_path[126];
	};

表 9. AF_UNIX アドレス構造

アドレス構造フィールド	定義
sun_len	このフィールドには UNIX 98 仕様のアドレス長が入ります。 **注:** sun_len フィールドは、BSD 4.4 との互換性を保つために存在しているに過ぎません。BSD 4.4/UNIX 98 互換性を使用する場合であっても、このフィールドを使用する必要はありません。入力アドレスの場合、このフィールドは無視されます。
sun_family	このフィールドには、アドレス・ファミリーが入ります。
sun_path	このフィールドには、ファイル・システムの項目へのパス名が入ります。

AF_UNIX アドレス・ファミリーでは、プロトコル標準が関係しないため、プロトコル指定は適用されません。この 2 つのプロセスが使用する通信機構はマシン固有です。

AF_UNIX の使用法とこのアドレス・ファミリーを使用するサンプル・プログラムについては、『AF_UNIX アドレス・ファミリーの使用』を参照してください。

AF_UNIX_CCSID アドレス・ファミリー

AF_UNIX_CCSID ファミリーは AF_UNIX アドレス・ファミリーと互換性があり、同じ制限もあります。これらの両方のファミリーはどちらも、コネクションレス型またはコネクション型のどちらかにすることができ、2 つのプロセスを接続する外部通信関数はありません。異なっているのは、アドレス・ファミリー AF_UNIX_CCSID を指定したソケットが、**sockaddr_unc** アドレス構造を使用するという点です。このアドレス構造は sockaddr_un と類似していますが、**Qlg_Path_Name_T** 形式を使用して、UNICODE または任意の CCSID でパス名を指定できます。Information Center の『path name format』を参照してください。

ただし、AF_UNIX ソケットは、AF_UNIX_CCSID ソケットのパス名を AF_UNIX アドレス構造に戻すことがあるので、パス・サイズには制限があります。AF_UNIX がサポートするのは 126 文字だけなので、AF_UNIX_CCSID も 126 文字に制限されます。

ユーザーは、1 つのソケットで AF_UNIX アドレスと AF_UNIX_CCSID アドレスを交換できません。**socket()** 呼び出しで AF_UNIX_CCSID を指定すると、以降の API 呼び出しですべてのアドレスが **sockaddr_unc** でなければなりません。

```
struct sockaddr_unc {
     short             sunc_family;
     short             sunc_format;
     char              sunc_zero[12];
     Qlg_Path_Name_T   sunc_qlg;
     union {
     char              unix[126];
     wchar_t           wide[126];
```

```
    char*             p_unix;
    wchar_t*          p_wide;
   }                  sunc_path;
   } ;
```

表 10. AF_UNIX_CCSID アドレス構造

アドレス構造フィールド	定義
sunc_family	このフィールドにはアドレス・ファミリーが入りますが、これは常に AF_UNIX_CCSID です。
sunc_format	このフィールドには、パス名の形式用の以下の 2 つの定義済みの値が入ります。 • **SO_UNC_DEFAULT** は、統合ファイル・システム・パス名用の現在のデフォルト CCSID を使用する、長いパス名を示します。sunc_qlg フィールドは無視されます。 • **SO_UNC_USE_QLG** は、**sunc_qlg** フィールドがパス名の形式と CCSID とを定義していることを示します。
sunc_zero	このフィールドは予約済みです。このフィールドは 16 進数のゼロに設定してください。
sunc_qlg	このフィールドはパス名形式を指定します。
sunc_path	このフィールドにはパス名が入ります。このパス名は、最大 126 文字で、単一バイトでも 2 バイトでも構いません。パス名は、sunc_path フィールドに入れても、別々に割り振って sunc_path が指すようにしても構いません。形式は sunc_format と sunc_qlg によって決まります。

AF_UNIX_CCSID ソケットの詳細については、『AF_UNIX_CCSID アドレス・ファミリーの使用』を参照してください。このトピックでは、サンプル・プログラムが提供されます。

AF_TELEPHONY アドレス・ファミリー

AF_TELEPHONY アドレス・ファミリーを使用すると、ユーザーは、標準のソケット API を使用して ISDN 電話ネットワークを介したダイヤル呼び出しや通話への応答を行うことができます。このドメインでの接続の端点を形成するソケットは、実際に通話の受信者と送信者になります。このアドレス・ファミリーのアドレスは、40 桁の電話番号で表されます。このアドレス・ファミリーが最もよく使用されるのは、FAX サポートの開発時です。

システムは、AF_TELEPHONY ソケットをコネクション型ソケット (ソケット・タイプ SOCK_STREAM) としてのみサポートします。電話ドメイン・ソケットでの接続で提供されるのは、下層の電話接続の信頼性のみです。確実な転送を行う場合は、このアドレス・ファミリーを使用する FAX アプリケーションなどの、サービスを提供するアプリケーションを処理する必要があります。

AF_TELEPHONY アドレス・ファミリーを指定したソケットは、以下のように **sockaddr_tel** アドレス構造を使用します。

```
struct sockaddr_tel {
     short stel_family;
     struct tel_addr stel_addr;
     char stel_zero[4];
};
```

電話アドレスは 2 バイト長で構成され、最大 40 桁 (0 から 9) の電話番号がそれに続きます。

```
struct tel_addr {
     unsigned short  t_len
     char            t_addr[40];
};
```

表 11. AF_TELEPHONY アドレス構造

アドレス構造フィールド	定義
stel_family	このフィールドには、アドレス・ファミリーが入ります。
stel_addr	このフィールドには電話アドレスが入ります。
stel_zero	このフィールドは予約フィールドです。

AF_TELEPHONY アドレス・ファミリーの詳細については、『AF_TELEPHONY アドレス・ファミリーの使用』を参照してください。このトピックでは、AF_TELEPHONY アドレス・ファミリーを使用する環境の構成方法についてのステップを説明しています。

ソケット・タイプ

ソケット呼び出しの 2 番目のパラメーターによって、ソケットのタイプが決まります。ソケット・タイプによって、あるマシンから別のマシンまたはあるプロセスから別のプロセスへのデータのトランスポート用に使用可能にされる、接続のタイプと特性が識別されます。次のリストに、iSeries がサポートするソケット・タイプを示します。

ストリーム (SOCK_STREAM)
このソケット・タイプはコネクション型です。 **bind()**、**listen()**、**accept()**、および **connect()** 関数を使用して端末相互間接続を確立してください。SOCK_STREAM はエラーや重複なしでデータを送信し、送信時の順序でデータを受信します。SOCK_STREAM は、データのオーバーランを防ぐためにフロー制御を構築します。データ上にレコード境界は設けられません。SOCK_STREAM はデータをバイトのストリームと見なします。iSeries 実装では、ストリーム・ソケットを、伝送制御プロトコル (TCP)、システム・ネットワーク体系 (SNA)、AF_UNIX、AF_UNIX_CCSID、および AF TELEPHONY ソケットで使用することができます。またストリーム・ソケットを使用して、セキュア・ホスト (ファイアウォール) の外部にあるシステムと通信することもできます。

データグラム (SOCK_DGRAM)
インターネット・プロトコル用語では、データ転送の基本単位を**データグラム**といいます。基本的にはいくつかのデータを伴ったヘッダーを指します。データグラム・ソケットはコネクションレス型です。トランスポート・プロバイダー (プロトコル) との端末相互間接続が確立されることはありません。ソケットはデータグラムを独立パケットとして、配送の保証がないまま送信します。データの喪失または重複が起きたり、データグラムが壊れて着信することもあります。データグラムのサイズは、1 回のトランザクションで送信できるサイズに限定されています。いくつかのトランスポート・プロバイダーに対して、それぞれのデータグラムがネットワーク内の異なる経路を使用できます。このソケット・タイプでは **connect()** 関数を発行できますが、プログラムを送受信するための宛先アドレスは **connect()** 関数で指定しなければなりません。iSeries 実装の場合、データグラム・ソケットは、ユーザー・データグラム・プロトコル (UDP)、SNA で使用することができ、AF_UNIX および AF_UNIX_CCSID アドレス・ファミリーと共に使用することもできます。

ロー (SOCK_RAW)
このソケット・タイプでは、インターネット・プロトコル (IPv4 または IPv6) やインターネット制御メッセージ・プロトコル (ICMP または ICMP6) のような下位層プロトコルに直接アクセスできます。トランスポート・プロバイダーが用いるプロトコル・ヘッダー情報を管理することになるため、SOCK_RAW を扱

うためには多くのプログラミングの専門知識が必要になります。トランスポート・プロバイダーはこのレベルでデータの形式を決定し、セマンティクスを特定します。

ソケット・プロトコル

プロトコルによって、ネットワーク内のあるマシンから別のマシンへ (または、同じマシン内のあるプロセスから別のプロセスへ) アプリケーション・データを移送できます。アプリケーションは、**socket()** 関数の **protocol** パラメーターにトランスポート・プロバイダーを指定します。

AF_INET アドレス・ファミリーでは、複数のトランスポート・プロバイダーを使用できます。 SNA、TCP/IP、または UDP/IP のプロトコルは、同時に同じソケット上で活動状態になります。ALWANYNET (ANYNET サポートを可能にする) ネットワーク属性を指定すれば、AF_INET ソケット・アプリケーションに TCP/IP 以外のトランスポートも使用できるかどうかを顧客が選択できるようになります。このネットワーク属性は ***YES** または ***NO** です。デフォルト値は ***NO** です。

たとえば、現行状況 (デフォルト状況) が *NO の場合、SNA トランスポート上での AF_INET の使用は活動状態になりません。AF_INET ソケットを TCP/IP トランスポートでのみ使用する場合は、CPU 使用率を改善するために ALWANYNET 状況を *NO に設定する必要があります。

注: ALWANYNET ネットワーク属性は、TCP/IP サポート上の APPC にも影響を与えます。

APPC 構成オプションについては、『APPC、APPN、HPR の構成』を参照してください。

TCP/IP を介した AF_INET ソケットでは、SOCK_RAW というタイプを指定することもできます。これは、ソケットがインターネット・プロトコル (IP) というネットワーク層と直接通信することを示しています。TCP または UDP トランスポート・プロバイダーは、通常はこの層と通信します。 SOCK_RAW ソケットを使用する場合、アプリケーション・プログラムは 0 から 255 のプロトコルをどれでも指定できます (TCP および UDP プロトコルの場合を除く)。ネットワーク上でマシンが通信している場合、IP ヘッダーにこのプロトコル数が入ります。アプリケーション・プログラムは、UDP または TCP トランスポートが通常提供するすべてのトランスポート・サービスを提供しなければならないので、そのアプリケーション・プログラムが事実上トランスポート・プロバイダーとなります。

AF_UNIX、AF_UNIX_CCSID、および AF_TELEPHONY アドレス・ファミリーでは、プロトコル規格は関係しないので、プロトコルを指定しても実際には意味がありません。同一マシン上の 2 つのプロセス間の通信機構はマシン固有です。

ソケットの基本設計

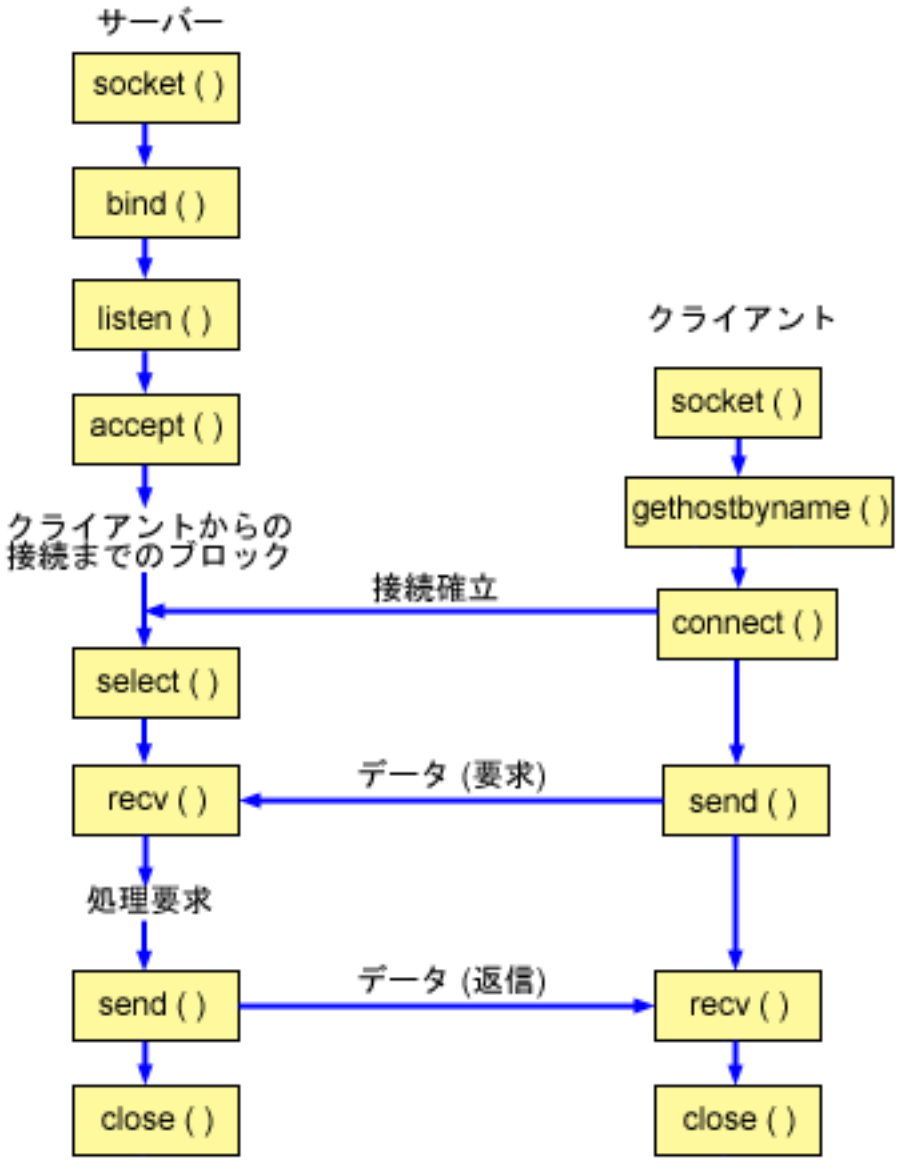
このトピックでは、最も基本的な設計を使用するソケット・プログラムの例を示します。これらの例は、より複雑なソケット設計の基本となるものです。これらの例には、前述のトピックで強調した基本概念のいくつかが実装されています。以下のサンプル・プログラムは、ソケット・プログラムの最も一般的なタイプの例を示しています。

- コネクション型ソケットの作成
- コネクションレス型ソケットの作成
- アドレス・ファミリーを使用したアプリケーションの設計

コネクション型ソケットの作成

以下のサーバーとクライアントの例は、伝送制御プロトコル (TCP) などのコネクション型プロトコル用に書き込まれたソケット API サーバーを示しています。

以下の図は、コネクション型プロトコル用ソケット API のクライアント/サーバー関係を表します。



ソケットのイベントのフロー: コネクション型サーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、コネクション型設計におけるサーバーとクライアント・アプリケーションの関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: コネクション型サーバー』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. **setsockopt()** 関数により、必要な待ち時間が満了する前にサーバーを再始動した場合に、ローカル・アドレスを再利用できるようになります。

3. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。この例では、ユーザーは s_addr をゼロに設定します。これにより、ポート 3005 を指定するあらゆる IPv4 クライアントが接続を確立できるようになります。
4. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。この例では、バックログが 10 に設定されています。これは、待ち行列に入れられた着信接続が 10 個になると、システムが着信要求を拒否するようになるということです。
5. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続の成功を待機して、無期限にブロックします。
6. **select()** 関数により、プロセスがイベントの発生を待機して、イベントが発生するとウェイクアップするようになります。この例では、データが読み取り可能な場合にのみ、システムはプロセスに通知します。この select の呼び出しでは、タイムアウトは 30 秒です。
7. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。この例では、クライアントが 250 バイトのデータを送信してくることが分かっているものとします。これを踏まえて、SO_RCVLOWAT ソケット・オプションを使用し、250 バイトのデータがすべて到着するまで **recv()** がウェイクアップしないように指定できます。
8. **send()** 関数が、クライアントにデータを送り返します。
9. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

ソケットのイベントのフロー: コネクション型クライアント

『例: コネクション型クライアント』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. クライアントのプログラム例では、**inet_addr()** 関数に渡されるサーバー・ストリングがドット 10 進 IP アドレスでなければ、それをサーバーのホスト名であると見なします。その場合は、**gethostbyname()** 関数を使用して、サーバーの IP アドレスを検索します。
3. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。
4. **send()** 関数が、250 バイトのデータをサーバーに送信します。
5. **recv()** 関数が、サーバーから 250 バイトのデータを送り返してくるのを待機します。この例では、こちらから送信したのとまったく同じ 250 バイトをそのままサーバーが送り返してくることが分かっているものとします。クライアントの例の場合、250 バイトのデータが別々のパケットで到着する可能性があるため、250 バイトが全部到着するまで、**recv()** 関数を繰り返し使用します。
6. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

例: コネクション型サーバー

以下のコード例は、コネクション型サーバーをどのように作成できるかを示しています。この例を使用して、独自のソケット・サーバー・アプリケーションを作成できます。コネクション型サーバー設計は、ソケット・アプリケーションの最も一般的なモデルの 1 つです。コネクション型設計では、サーバー・アプリケーションは、クライアント要求を受け入れるためのソケットを作成します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* This sample program provides a code for a connection-oriented server.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program .                     */
/*********************************************************************/
#include <stdio.h>
```

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

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /*********************************************************************/
   /* Variable and structure definitions.                               */
   /*********************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length, on=1;
   char   buffer[BUFFER_LENGTH];
   fd_set read_fd;
   struct timeval timeout;
   struct sockaddr_in serveraddr;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The */
   /* close() of each of the socket descriptors is only done once at the */
   /* very end of the program.                                          */
   /*********************************************************************/
   do
   {
      /*********************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the INET        */
      /* (Internet Protocol) address family with the TCP transport        */
      /* (SOCK_STREAM) will be used for this socket.                       */
      /*********************************************************************/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /*********************************************************************/
      /* The setsockopt() function is used to allow the local address to  */
      /* be reused when the server is restarted before the required wait  */
      /* time expires.                                                     */
      /*********************************************************************/
      rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
      if (rc < 0)
      {
         perror("setsockopt(SO_REUSEADDR) failed");
         break;
      }

      /*********************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.  In this example, the user sets the  */
      /* s_addr to zero, which allows connections to be established from  */
      /* any client that specifies port 3005.                              */
      /*********************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sin_family      = AF_INET;
      serveraddr.sin_port        = htons(SERVER_PORT);
      serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if (rc < 0)
{
   perror("bind() failed");
   break;
}

/*********************************************************************/
/* The listen() function allows the server to accept incoming       */
/* client connections.  In this example, the backlog is set to 10.  */
/* This means that the system will queue 10 incoming connection     */
/* requests before the system starts rejecting the incoming         */
/* requests.                                                        */
/*********************************************************************/
rc = listen(sd, 10);
if (rc< 0)
{
   perror("listen() failed");
   break;
}

printf("Ready for client connect().\n");

/*********************************************************************/
/* The server uses the accept() function to accept an incoming      */
/* connection request.  The accept() call will block indefinitely   */
/* waiting for the incoming connection to arrive.                   */
/*********************************************************************/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
   perror("accept() failed");
   break;
}

/*********************************************************************/
/* The select() function allows the process to wait for an event to */
/* occur and to wake up the process when the event occurs.  In this */
/* example, the system notifies the process only when data is       */
/* available to read.  A 30 second timeout is used on this select   */
/* call.                                                            */
/*********************************************************************/
timeout.tv_sec  = 30;
timeout.tv_usec = 0;

FD_ZERO(&read_fd);
FD_SET(sd2, &read_fd);

rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
if (rc < 0)
{
   perror("select() failed");
   break;
 }

if (rc == 0)
{
   printf("select() timed out.\n");
   break;
}

/*********************************************************************/
/* In this example we know that the client will send 250 bytes of   */
/* data over.  Knowing this, we can use the SO_RCVLOWAT socket      */
/* option and specify that we don't want our recv() to wake up until*/
/* all 250 bytes of data have arrived.                              */
/*********************************************************************/
```

```
      length = BUFFER_LENGTH;
      rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                          (char *)&length, sizeof(length));
      if (rc < 0)
      {
         perror("setsockopt(SO_RCVLOWAT) failed");
         break;
      }

      /*********************************************************************/
      /* Receive that 250 bytes data from the client                       */
      /*********************************************************************/
      rc = recv(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("recv() failed");
         break;
      }

      printf("%d bytes of data were received\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("The client closed the connection before all of the\n");
         printf("data was sent\n");
         break;
      }

      /*********************************************************************/
      /* Echo the data back to the client                                  */
      /*********************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("send() failed");
         break;
      }

      /*********************************************************************/
      /* Program complete                                                  */
      /*********************************************************************/

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
   if (sd2 != -1)
      close(sd2);
}
```

例: コネクション型クライアント

以下の例は、コネクション型設計で、コネクション型サーバーに接続するソケット・クライアント・プログラムを作成する方法を示したものです。サービスのクライアント (クライアント・プログラム) は、サーバー・プログラムのサービスを要求しなければなりません。このコード例を使用して、独自のクライアント・アプリケーションを作成できます。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* This sample program provides a code for a connection-oriented client.  */
/*********************************************************************/

/*********************************************************************/
```

```
/* Header files needed for this sample program                        */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
/* SERVER_NAME.                             */
void main(int argc, char *argv[])
{
   /*********************************************************************/
   /* Variable and structure definitions.                               */
   /*********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char   buffer[BUFFER_LENGTH];
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   struct sockaddr_in serveraddr;
   struct hostent *hostp;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The */
   /* close() of the socket descriptor is only done once at the very end */
   /* of the program.                                                   */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing  */
      /* an endpoint.  The statement also identifies that the INET       */
      /* (Internet Protocol) address family with the TCP transport       */
      /* (SOCK_STREAM) will be used for this socket.                     */
      /*******************************************************************/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* If an argument was passed in, use this as the server, otherwise */
      /* use the #define that is located at the top of this program.     */
      /*******************************************************************/
      if (argc > 1)
         strcpy(server, argv[1]);
      else
         strcpy(server, SERVER_NAME);

      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sin_family      = AF_INET;
      serveraddr.sin_port        = htons(SERVER_PORT);
      serveraddr.sin_addr.s_addr = inet_addr(server);
      if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
```

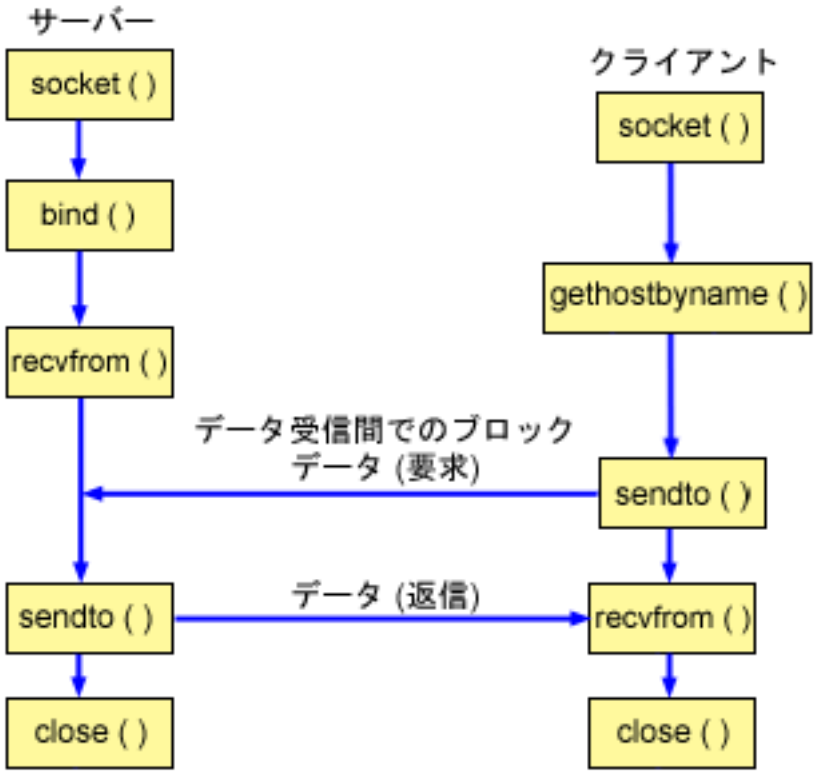
```
{
   /*****************************************************************/
   /* The server string that was passed into the inet_addr()        */
   /* function was not a dotted decimal IP address.  It must        */
   /* therefore be the hostname of the server.  Use the             */
   /* gethostbyname() function to retrieve the IP address of the    */
   /* server.                                                       */
   /*****************************************************************/

   hostp = gethostbyname(server);
   if (hostp == (struct hostent *)NULL)
   {
      printf("Host not found --> ");
      printf("h_errno = %d\n", h_errno);
      break;
   }

   memcpy(&serveraddr.sin_addr,
          hostp->h_addr,
          sizeof(serveraddr.sin_addr));
}

/*********************************************************************/
/* Use the connect() function to establish a connection to the       */
/* server.                                                           */
/*********************************************************************/
rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if (rc < 0)
{
   perror("connect() failed");
   break;
}

/*********************************************************************/
/* Send 250 bytes of a's to the server                               */
/*********************************************************************/
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("send() failed");
   break;
}

/*********************************************************************/
/* In this example we know that the server is going to respond with  */
/* the same 250 bytes that we just sent.  Since we know that 250     */
/* bytes are going to be sent back to us, we could use the           */
/* SO_RCVLOWAT socket option and then issue a single recv() and      */
/* retrieve all of the data.                                         */
/*                                                                   */
/* The use of SO_RCVLOWAT is already illustrated in the server       */
/* side of this example, so we will do something different here.     */
/* The 250 bytes of the data may arrive in separate packets,         */
/* therefore we will issue recv() over and over again until all      */
/* 250 bytes have arrived.                                           */
/*********************************************************************/
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
   rc = recv(sd, & buffer[bytesReceived],
             BUFFER_LENGTH - bytesReceived, 0);
   if (rc < 0)
   {
      perror("recv() failed");
      break;
   }
```

```
         else if (rc == 0)
         {
            printf("The server closed the connection\n");
            break;
         }

         /*********************************************************************/
         /* Increment the number of bytes that have been received so far  */
         /*********************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

コネクションレス型ソケットの作成

コネクションレス型ソケットは、データを転送するための接続を確立しません。その代わり、サーバー・アプリケーションはクライアントが要求を送信できるよう、自分の名前を指定します。コネクションレス型ソケットは、TCP/IP の代わりにユーザー・データグラム・プロトコル (UDP) を使用します。『例: コネクションレス型サーバー』と『例: コネクションレス型クライアント』は、ユーザー・データグラム・プロトコル (UDP) 用に作成されたソケット API を示しています。

以下の図は、コネクションレス型ソケット設計用のコード例で使用される、ソケット API のクライアント/サーバーの関係を表しています。



ソケットのイベントのフロー: コネクションレス型サーバー

以下のソケット呼び出しのシーケンスは、図およびこの後のプログラム例の説明となっています。これはまた、コネクションレス型設計におけるサーバーとクライアント・アプリケーションの関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: コネクションレス型サーバー』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと UDP トランスポート (SOCK_DGRAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数が、ソケットの固有名を取得します。この例では、ユーザーは s_addr をゼロに設定します。これは、UDP ポート 3555 が、システム上のすべての IPv4 アドレスにバインドされるということです。
3. サーバーが、データを受信するために **recvfrom()** 関数を使用します。 **recvfrom()** 関数は、データの到着を無期限に待機します。
4. **sendto()** 関数がデータをクライアントに送り返します。
5. **close()** 関数が、オープンしているソケット記述子をすべて終了させます。

ソケットのイベントのフロー: コネクションレス型クライアント

『例: コネクションレス型クライアント』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと UDP トランスポート (SOCK_DGRAM) を使用することも示します。
2. クライアントのプログラム例では、inet_addr() 関数に渡されるサーバー・ストリングがドット 10 進 IP アドレスでなければ、それをサーバーのホスト名であると見なします。その場合は、**gethostbyname()** 関数を使用して、サーバーの IP アドレスを検索します。
3. **sendto()** 関数を使用して、データをサーバーに送信します。
4. **recvfrom()** 関数を使用して、サーバーが送り返してきたデータを受信します。
5. **close()** 関数が、オープンしているソケット記述子をすべて終了させます。

例: コネクションレス型サーバー

この例を使用して、独自のコネクションレス型サーバー設計を作成することができます。この例では UDP を使用してコネクションレス型ソケット・サーバー・プログラムを作成します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This sample program provides a code for a connectionless server.          */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*****************************************************************************/
/* Constants used by this program                                            */
/*****************************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
```

```
#define FALSE              0

void main()
{
   /*********************************************************************/
   /* Variable and structure definitions.                               */
   /*********************************************************************/
   int    sd=-1, rc;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_in serveraddr;
   struct sockaddr_in clientaddr;
   int    clientaddrlen = sizeof(clientaddr);

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The */
   /* close() of each of the socket descriptors is only done once at the */
   /* very end of the program.                                          */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing */
      /* an endpoint.  The statement also identifies that the INET      */
      /* (Internet Protocol) address family with the UDP transport      */
      /* (SOCK_DGRAM) will be used for this socket.                     */
      /*******************************************************************/
      sd = socket(AF_INET, SOCK_DGRAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.  In this example, the user sets the */
      /* s_addr to zero, which means that the UDP port of 3555 will be  */
      /* bound to all IP addresses on the system.                       */
      /*******************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sin_family      = AF_INET;
      serveraddr.sin_port        = htons(SERVER_PORT);
      serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

      rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
      if (rc < 0)
      {
         perror("bind() failed");
         break;
      }

      /*******************************************************************/
      /* The server uses the recvfrom() function to receive that data.  */
      /* The recvfrom() function waits indefinitely for data to arrive. */
      /*******************************************************************/
      rc = recvfrom(sd, buffer, sizeof(buffer), 0,
                    (struct sockaddr *)&clientaddr,
                    &clientaddrlen);
      if (rc < 0)
      {
         perror("recvfrom() failed");
         break;
      }

      printf("server received the following: <%s>\n", buffer);
      printf("from port %d and address %s\n",
             ntohs(clientaddr.sin_port),
```

```
            inet_ntoa(clientaddr.sin_addr));

      /*******************************************************************/
      /* Echo the data back to the client                                */
      /*******************************************************************/
      rc = sendto(sd, buffer, sizeof(buffer), 0,
                  (struct sockaddr *)&clientaddr,
                  sizeof(clientaddr));
      if (rc < 0)
      {
         perror("sendto() failed");
         break;
      }

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

例: コネクションレス型クライアント

以下の例は、UDP を使用してコネクションレス型ソケット・クライアント・プログラムをサーバーに接続する方法について示します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* This sample program provides a code for a connectionless client.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
/* SERVER_NAME                             */
void main(int argc, char *argv[])
{
   /*******************************************************************/
   /* Variable and structure definitions.                             */
   /*******************************************************************/
   int    sd, rc;
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   char   buffer[BUFFER_LENGTH];
```

```
struct hostent *hostp;
struct sockaddr_in serveraddr;
int    serveraddrlen = sizeof(serveraddr);

/*********************************************************************/
/* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
/* close() of the socket descriptor is only done once at the very end  */
/* of the program.                                                     */
/*********************************************************************/
do
{
   /******************************************************************/
   /* The socket() function returns a socket descriptor representing   */
   /* an endpoint.  The statement also identifies that the INET        */
   /* (Internet Protocol) address family with the UDP transport        */
   /* (SOCK_STREAM) will be used for this socket.                      */
   /******************************************************************/
   sd = socket(AF_INET, SOCK_DGRAM, 0);
   if (sd < 0)
   {
      perror("socket() failed");
      break;
   }

   /******************************************************************/
   /* If an argument was passed in, use this as the server, otherwise  */
   /* use the #define that is located at the top of this program.      */
   /******************************************************************/
   if (argc > 1)
      strcpy(server, argv[1]);
   else
      strcpy(server, SERVER_NAME);

   memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sin_family      = AF_INET;
      serveraddr.sin_port        = htons(SERVER_PORT);
      serveraddr.sin_addr.s_addr = inet_addr(server);
      if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
   {
      /*****************************************************************/
      /* The server string that was passed into the inet_addr()        */
      /* function was not a dotted decimal IP address.  It must        */
      /* therefore be the hostname of the server.  Use the             */
      /* gethostbyname() function to retrieve the IP address of the    */
      /* server.                                                       */
      /*****************************************************************/
      hostp = gethostbyname(server);
      if (hostp == (struct hostent *)NULL)
      {
         printf("Host not found --> ");
         printf("h_errno = %d\n", h_errno);
         break;
      }

      memcpy(&serveraddr.sin_addr,
             hostp->h_addr,
             sizeof(serveraddr.sin_addr));
   }

   /******************************************************************/
   /* Initialize the data block that is going to be sent to the server */
   /******************************************************************/
   memset(buffer, 0, sizeof(buffer));
   strcpy(buffer, "A CLIENT REQUEST");

   /******************************************************************/
   /* Use the sendto() function to send the data to the server.        */
```

```
      /*******************************************************************/
      rc = sendto(sd, buffer, sizeof(buffer), 0,
                  (struct sockaddr *)&serveraddr,
                  sizeof(serveraddr));
      if (rc < 0)
      {
         perror("sendto() failed");
         break;
      }

      /*******************************************************************/
      /* Use the recvfrom() function to receive the data back from the    */
      /* server.                                                         */
      /*******************************************************************/
      rc = recvfrom(sd, buffer, sizeof(buffer), 0,
                    (struct sockaddr *)&serveraddr,
                    & serveraddrlen);
      if (rc < 0)
      {
         perror("recvfrom() failed");
         break;
      }

      printf("client received the following: <%s>\n", buffer);
      printf("from port %d, from address %s\n",
             ntohs(serveraddr.sin_port),
             inet_ntoa(serveraddr.sin_addr));

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

アドレス・ファミリーを使用したアプリケーションの設計

以下のトピックでは、それぞれのソケット・アドレス・ファミリーを例示するサンプル・プログラムを示します。

- AF_INET アドレス・ファミリーの使用
- AF_INET6 アドレス・ファミリーの使用
- AF_UNIX アドレス・ファミリーの使用
- AF_TELEPHONY アドレス・ファミリーの使用
- AF_UNIX_CCSID アドレス・ファミリーの使用

AF_INET アドレス・ファミリーの使用

AF_INET アドレス・ファミリー・ソケットは、コネクション型 (タイプ SOCK_STREAM) とコネクションレス型 (タイプ SOCK_DGRAM) のいずれかにすることができます。コネクション型 AF_INET ソケットは、TCP をトランスポート・プロトコルとして使用します。コネクションレス型 AF_INET ソケットは、UDP をトランスポート・プロトコルとして使用します。 AF_INET ドメイン・ソケットの作成時に、ソケット・プログラムのアドレス・ファミリーに AF_INET を指定します。 AF_INET ソケットもタイプ SOCK_RAW を使用することができます。このタイプを設定すると、アプリケーションは IP 層に直接接続し、TCP または UDP トランスポートのいずれも使用しません。

AF_INET アドレス・ファミリーを使用する環境の設定に関する詳細は、『ソケット・プログラミングの前提条件』を参照してください。

AF_INET アドレス・ファミリーを使用するサンプル・プログラムについては、『例: コネクション型サーバー』および『例: コネクション型クライアント』を参照してください。

AF_INET6 アドレス・ファミリーの使用

AF_INET6 ソケットは、インターネット・プロトコル バージョン 6 (IPv6) の 128 ビット (16 バイト) アドレス構造をサポートします。プログラマーは AF_INET6 アドレス・ファミリーを使用してアプリケーションを作成し、IPv4 ノードまたは IPv6 ノードのどちらかから、あるいは IPv6 ノードのみから、クライアント要求を受け入れることができます。

AF_INET ソケット同様、AF_INET6 ソケットにも、コネクション型 (タイプ SOCK_STREAM) とコネクションレス型 (タイプ SOCK_DGRAM) があります。コネクション型 AF_INET6 ソケットは、TCP をトランスポート・プロトコルとして使用します。コネクションレス型 AF_INET6 ソケットは、UDP をトランスポート・プロトコルとして使用します。 AF_INET6 ドメイン・ソケットの作成時に、ソケット・プログラムのアドレス・ファミリーに AF_INET6 を指定します。 AF_INET6 ソケットもタイプ SOCK_RAW を使用することができます。このタイプを設定すると、アプリケーションは IP 層に直接接続し、TCP または UDP トランスポートのいずれも使用しません。 AF_INET6 アドレス・ファミリーを使用する環境の設定に関する詳細は、『ソケット・プログラミングの前提条件』を参照してください。

IPv6 アプリケーションと IPv4 アプリケーションの互換性

AF_INET6 アドレス・ファミリーを使ってソケット・アプリケーションを作成すれば、インターネット・プロトコル バージョン 6 (IPv6) アプリケーションは、インターネット・プロトコル バージョン 4 (IPv4) アプリケーション (AF_INET アドレス・ファミリーを使用するアプリケーション) と一緒に機能できるようになります。この機能により、ソケット・プログラマーは IPv4 マップ IPv6 アドレス形式を使用できます。このアドレス形式は、IPv4 ノードの IPv4 アドレスを IPv6 アドレスとして表します。 IPv4 アドレスは IPv6 アドレスの下位 32 ビットにエンコードされ、高位 96 ビットは 0:0:0:0:0:FFFF という接頭部で固定されます。 IPv4 マップ・アドレスの一例を挙げます。

```
::FFFF:192.1.1.1
```

指定されたホストが IPv4 アドレスしか持っていない場合、**getaddrinfo()** 関数でこのようなアドレスを自動的に生成できます。

AF_INET6 ソケットを使用して IPv4 ノードへの TCP 接続をオープンするアプリケーションを作成することができます。そのためには、宛先の IPv4 アドレスを IPv4 マップ IPv6 アドレスにエンコードし、**connect()** または **sendto()** 呼び出しの中で、そのアドレスを sockaddr_in6 構造に渡すことができます。アプリケーションが AF_INET6 ソケットを使用して、IPv4 ノードからの TCP 接続を受け入れたり、IPv4 ノードから UDP パケットを受信したりする場合、システムは同様の方法でエンコードされた sockaddr_in6 構造を使用し、相手システムのアドレスを、**accept()**、**recvfrom()**、または **getpeername()** 呼び出しによってアプリケーションに戻します。

アプリケーションは、**bind()** 関数によって、UDP パケットや TCP 接続の送信元の IP アドレスを選択することができますが、システムに送信元アドレスを選択してもらいたい場合もあります。そのような場合、IPv4 であれば INADDR_ANY マクロを使用しますが、それと同様の方法でアプリケーションは in6addr_any を使用できます。この方法のバインドに追加された機能は、AF_INET6 ソケットが、IPv4 ノードと IPv6 ノードの両方と通信できるようになったということです。たとえば、in6addr_any にバインドされた listen 中のソケット上で **accept()** を発行するアプリケーションは、IPv4 または IPv6 ノードのどちらか一方からの接続を受け入れることができます。この動作は、IPPROTO_IPV6 レベルのソケット・オプシ

ョン IPV6_V6ONLY を使用して変更が可能です。相互運用しているノードのタイプを知る必要のあるアプリケーションはほとんどないと考えられますが、それを知る必要のあるアプリケーションのために、<netinet/in.h> で定義されている IN6_IS_ADDR_V4MAPPED() マクロが提供されています。

IPv4 と IPv6 をより詳細に比較したい場合は、Information Center の『IPv4 と IPv6 との比較』を参照してください。ここでは、これらの 2 つのプロトコルの機能を比較しています。

AF_INET6 ソケットが IPv4 ノードと IPv6 ノードの両方と通信する場合のプログラム例と状況の説明に関しては、『ソケットのシナリオ: IPv4 クライアントと IPv6 クライアントを受け入れるアプリケーションの作成』を参照してください。

IPv6 の制約事項

現在、OS/400 の IPv6 サポートは、V5R2 の特定の機能に限られています。以下の表には、これらの制約事項と、それがソケットのプログラマーにどのような影響を与えるかについての説明が列挙されています。

表 12. IPv6 の制約事項とその影響

制約事項	影響
IPv6 はフラグメントをサポートしていません。	AF_INET6 (SOCK_DGRAM) は、インターフェースの MTU からヘッダーのサイズを引いたものより大きなデータグラムを送信しようとしてはなりません。
IPv6 エニーキャストはサポートされていません。	エニーキャスト・アドレスに接続または送信することはできません。
IPv6 マルチキャストはサポートされていません。	マルチキャスト・データグラムを送信または受信することはできません。
iSeries ホスト・テーブルは、IPv6 アドレスをサポートしていません。	**getaddrinfo()** API および **getnameinfo()** API は、ホスト・テーブルで IPv6 アドレスを見つけることができません。これらの API は、DNS でアドレスを見つけることしかできません。
gethostbyname() API および **gethostbyaddr()** API は、IPv4 アドレス解決のみサポートしています。	IPv6 アドレス解決が必要な場合には、**getaddrinfo()** API と **getnameinfo()** API を使用します。

AF_UNIX アドレス・ファミリーの使用

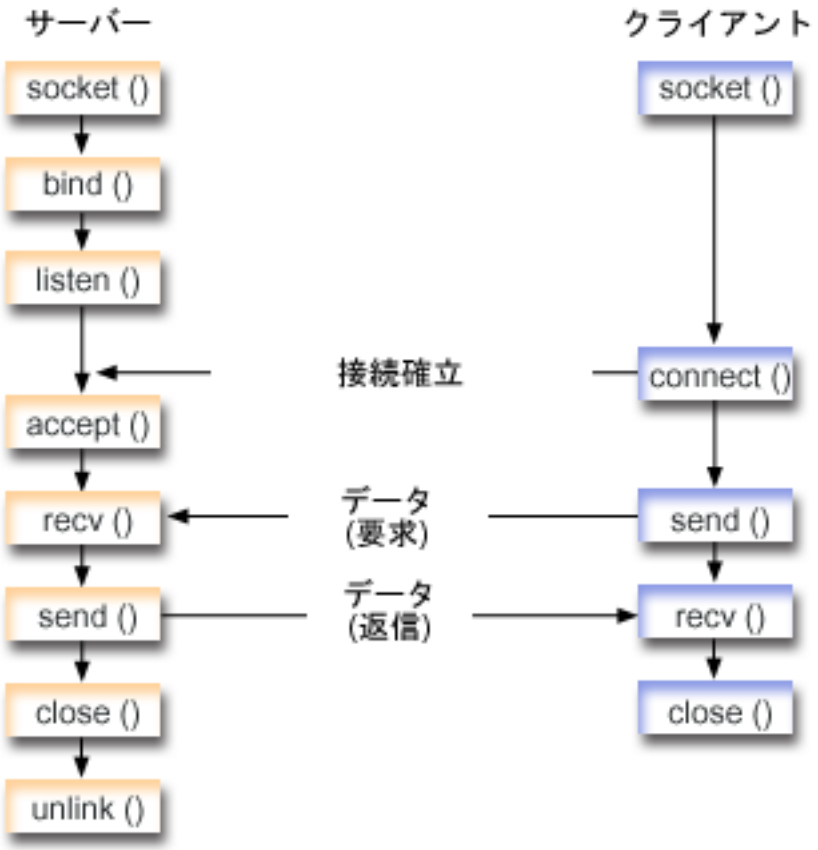
AF_UNIX アドレス・ファミリー (AF_UNIX または AF_UNIX_CCSID アドレス・ファミリーを使用するソケット) には、コネクション型 (SOCK_STREAM タイプ) またはコネクションレス型 (SOCK_DGRAM タイプ) があります。この 2 つのタイプは両方とも、2 つのプロセスを接続する外部通信関数がないので信頼性があります。

UNIX ドメイン・データグラム・ソケットは UDP データグラム・ソケットとは異なる働きをします。システムが自動的に未使用のポート番号を割り当てるため、UDP データグラム・ソケットでは、クライアント・プログラムが **bind()** 関数を呼び出す必要はありません。サーバーはポート番号にデータグラムを送信し返すことができます。ただし、UNIX ドメイン・データグラム・ソケットを使用すると、システムはクライアントに自動的にパス名を割り当てません。そこで、UNIX ドメイン・データグラムを使用するすべてのクライアント・プログラムは、**bind()** 関数を呼び出さなければなりません。クライアントの **bind()** に指定された正確なパス名は、サーバーに渡されるものです。よって、クライアントが相対パス名 (先頭に / で修飾されているパス名) を指定する場合は、それが同じ現行ディレクトリーで実行していない限り、サーバーはクライアントにデータグラムを送信することはできません。

アプリケーションがこのアドレス・ファミリーに使用できるパス名の例として、/tmp/myserver または servers/thatserver が挙げられます。servers/thatserver では、完全修飾ではないパス名 (/ が指定されていない) になります。つまり、ファイル・システム階層での項目のロケーションは現行作業ディレクトリーに関連して判別されます。

注: ファイル・システムのパス名は NLS 化されています。

以下の図は、AF_UNIX アドレス・ファミリーのクライアント/サーバー関係を表します。AF_UNIX アドレス・ファミリーを使用する環境の設定に関する詳細は、『ソケット・プログラミングの前提条件』を参照してください。



ソケットのイベントのフロー: AF_UNIX アドレス・ファミリーを使用するサーバー・アプリケーション
『例: AF_UNIX アドレス・ファミリーを使用するサーバー・アプリケーション』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために UNIX アドレス・ファミリーとストリーム・トランスポート (SOCK_STREAM) を使用することも示します。関数は、端点を表すソケット記述子を戻します。さらに、**socketpair()** 関数を使用して UNIX ソケットを初期化することもできます。

   AF_UNIX または AF_UNIX_CCSID は、唯一 **socketpair()** 関数をサポートしているアドレス・ファミリーです。**socketpair()** 関数は、名前がなく接続されている 2 つのソケット記述子を戻します。
2. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。

UNIX ドメイン SOCKET のネーム・スペースはパス名で構成されています。ソケット・プログラムが **bind()** 関数を呼び出すと、ファイル・システム・ディレクトリーに項目が作成されます。パス名がすでに存在する場合、**bind()** は失敗します。そこで、UNIX ドメイン SOCKET プログラムが常に **unlink()** 関数を呼び出して、終了時にディレクトリーの項目を除去する必要があります。

3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。この例では、バックログが 10 に設定されています。これは、待ち行列に入れられた着信接続が 10 個になると、システムが着信要求を拒否するようになるということです。
4. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。この例では、クライアントが 250 バイトのデータを送信してくることが分かっているものとします。これを踏まえて、SO_RCVLOWAT ソケット・オプションを使用し、250 バイトのデータがすべて到着するまで **recv()** がウェイクアップしないように指定できます。
5. **send()** 関数が、クライアントにデータを送り返します。
6. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。
7. **unlink()** 関数が、UNIX パス名をファイル・システムから除去します。

ソケットのイベントのフロー: AF_UNIX アドレス・ファミリーを使用するクライアント・アプリケーション

『例: AF_UNIX アドレス・ファミリーを使用するクライアント・アプリケーション』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために UNIX アドレス・ファミリーとストリーム・トランスポート (SOCK_STREAM) を使用することも示します。関数は、端点を表すソケット記述子を戻します。さらに、**socketpair()** 関数を使用して UNIX ソケットを初期化することもできます。

   AF_UNIX または AF_UNIX_CCSID は、唯一 **socketpair()** 関数をサポートしているアドレス・ファミリーです。**socketpair()** 関数は、名前がなく接続されている 2 つのソケット記述子を戻します。
2. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。
3. **send()** 関数が、サーバー・アプリケーションの SO_RCVLOWAT ソケット・オプションに指定されている 250 バイトのデータを送信します。
4. 250 バイトのデータが全部到着するまで、**recv()** 関数がループします。
5. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

例: AF_UNIX アドレス・ファミリーを使用するサーバー・アプリケーション: この例は、AF_UNIX アドレス・ファミリー用のサンプル・サーバーです。AF_UNIX アドレス・ファミリーは、他のアドレス・ファミリーと同じソケット呼び出しの多くを使用します。ただし、サーバー・アプリケーションを識別するためにパス名構造を使用するという点が異なっています。以下のサンプル・プログラムでは、AF_UNIX アドレス・ファミリーを使用しています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This sample program provides code for aserver application that uses     */
/* AF_UNIX address family                                                    */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
```

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

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /*******************************************************************/
   /* Variable and structure definitions.                             */
   /*******************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_un serveraddr;

   /*******************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of each of the socket descriptors is only done once at the  */
   /* very end of the program.                                        */
   /*******************************************************************/
   do
   {
      /****************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the UNIX        */
      /* address family with the stream transport (SOCK_STREAM) will be   */
      /* used for this socket.                                        */
      /****************************************************************/
      sd = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /****************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.                                  */
      /****************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sun_family = AF_UNIX;
      strcpy(serveraddr.sun_path, SERVER_PATH);

      rc = bind(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
      if (rc < 0)
      {
         perror("bind() failed");
         break;
      }

      /****************************************************************/
      /* The listen() function allows the server to accept incoming       */
      /* client connections.  In this example, the backlog is set to 10.  */
      /* This means that the system will queue 10 incoming connection     */
      /* requests before the system starts rejecting the incoming         */
      /* requests.                                                    */
      /****************************************************************/
      rc = listen(sd, 10);
      if (rc< 0)
      {
         perror("listen() failed");
         break;
      }
```

```
      printf("Ready for client connect().\n");

      /*****************************************************************/
      /* The server uses the accept() function to accept an incoming   */
      /* connection request.  The accept() call will block indefinitely*/
      /* waiting for the incoming connection to arrive.                */
      /*****************************************************************/
      sd2 = accept(sd, NULL, NULL);
      if (sd2 < 0)
      {
         perror("accept() failed");
         break;
      }

      /*****************************************************************/
      /* In this example we know that the client will send 250 bytes of*/
      /* data over.  Knowing this, we can use the SO_RCVLOWAT socket   */
      /* option and specify that we don't want our recv() to wake up   */
      /* until all 250 bytes of data have arrived.                     */
      /*****************************************************************/
      length = BUFFER_LENGTH;
      rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                        (char *)&length, sizeof(length));
      if (rc < 0)
      {
      }

      printf("%d bytes of data were received\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("The client closed the connection before all of the\n");
         printf("data was sent\n");
         break;
      }

      /*****************************************************************/
      /* Echo the data back to the client                              */
      /*****************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("send() failed");
         break;
      }

      /*****************************************************************/
      /* Program complete                                              */
      /*****************************************************************/

   } while (FALSE);

   /*******************************************************************/
   /* Close down any open socket descriptors                          */
   /*******************************************************************/
   if (sd != -1)
      close(sd);

   if (sd2 != -1)
      close(sd2);

    /******************************************************************/
   /* Remove the UNIX path name from the file system                  */
   /*******************************************************************/
   unlink(SERVER_PATH);
}
```

例: AF_UNIX アドレス・ファミリーを使用するクライアント・アプリケーション: この例は、AF_UNIX アドレス・ファミリー用のサンプル・クライアント・アプリケーションです。AF_UNIX アドレス・ファミリーは、他のアドレス・ファミリーと同じソケット呼び出しの多くを使用します。ただし、サーバー・アプリケーションを識別するためにパス名構造を使用するという点が異なっています。以下のサンプル・プログラムでは、AF_UNIX アドレス・ファミリーを使用して、サーバーへのクライアント接続を作成します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This sample program provides code for a client application that uses     */
/* AF_UNIX address family                                                    */
/*****************************************************************************/
/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*****************************************************************************/
/* Constants used by this program                                            */
/*****************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE    */
/* string, or set the server path in the   */
/* #define SERVER_PATH which is a CCSID    */
/* 500 string.                             */
void main(int argc, char *argv[])
{
   /**************************************************************************/
   /* Variable and structure definitions.                                    */
   /**************************************************************************/
   int    sd=-1, rc, bytesReceived;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_un serveraddr;

   /**************************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The      */
   /* close() of the socket descriptor is only done once at the very end     */
   /* of the program.                                                        */
   /**************************************************************************/
   do
   {
      /***********************************************************************/
      /* The socket() function returns a socket descriptor representing      */
      /* an endpoint.  The statement also identifies that the UNIX_CCSID     */
      /* address family with the stream transport (SOCK_STREAM) will be      */
      /* used for this socket.                                               */
      /***********************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /***********************************************************************/
      /* If an argument was passed in, use this as the server, otherwise     */
      /* use the #define that is located at the top of this program.         */
      /***********************************************************************/
```

```
   memset(&serveraddr, 0, sizeof(serveraddr));
   serveraddr.sun_family = AF_UNIX;
   if (argc > 1)
      strcpy(serveraddr.sun_path, argv[1]);
   else
      strcpy(serveraddr.sun_path, SERVER_PATH);

   /*******************************************************************/
   /* Use the connect() function to establish a connection to the     */
   /* server.                                                         */
   /*******************************************************************/
   rc = connect(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
   if (rc < 0)
   {
      perror("connect() failed");
      break;
   }

   /*******************************************************************/
   /* Send 250 bytes of a's to the server                             */
   /*******************************************************************/
   memset(buffer, 'a', sizeof(buffer));
   rc = send(sd, buffer, sizeof(buffer), 0);
   if (rc < 0)
   {
      perror("send() failed");
                  break;
   }

   /*******************************************************************/
   /* In this example we know that the server is going to respond with */
   /* the same 250 bytes that we just sent.  Since we know that 250   */
   /* bytes are going to be sent back to us, we could use the         */
   /* SO_RCVLOWAT socket option and then issue a single recv() and    */
   /* retrieve all of the data.                                       */
   /*                                                                 */
   /* The use of SO_RCVLOWAT is already illustrated in the server     */
   /* side of this example, so we will do something different here.   */
   /* The 250 bytes of the data may arrive in separate packets,       */
   /* therefore we will issue recv() over and over again until all    */
   /* 250 bytes have arrived.                                         */
   /*******************************************************************/
   bytesReceived = 0;
   while (bytesReceived < BUFFER_LENGTH)
   {
      rc = recv(sd, & buffer[bytesReceived],
                BUFFER_LENGTH - bytesReceived, 0);
      if (rc < 0)
      {
         perror("recv() failed");
         break;
      }
      else if (rc == 0)
      {
         printf("The server closed the connection\n");
         break;
      }

      /****************************************************************/
      /* Increment the number of bytes that have been received so far */
      /****************************************************************/
      bytesReceived += rc;
   }

} while (FALSE);

/*********************************************************************/
```

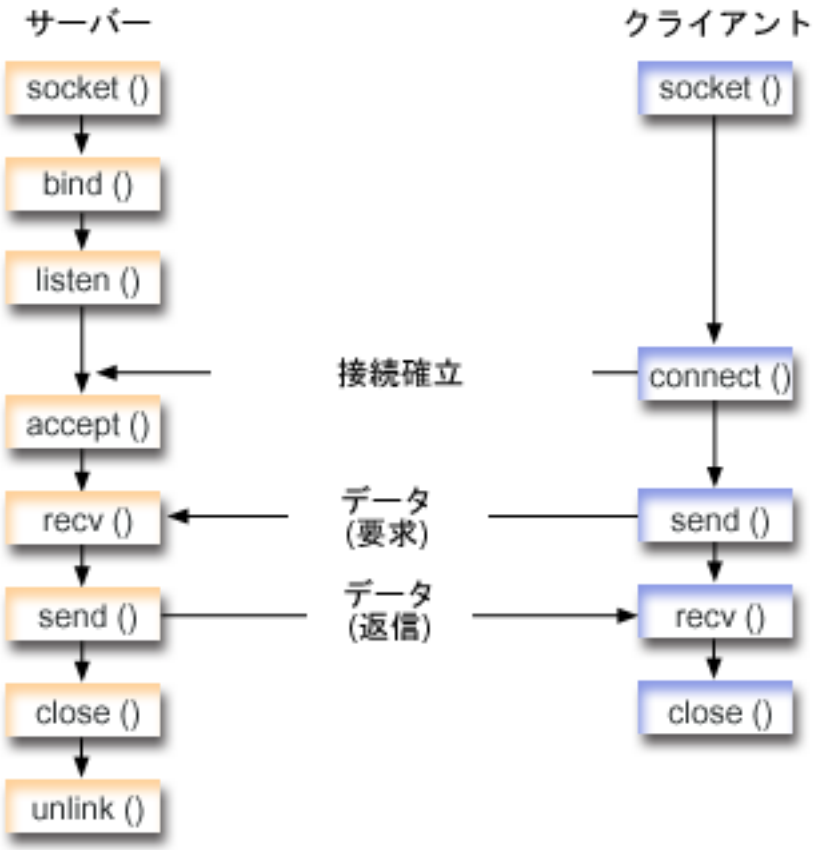
```
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

AF_UNIX_CCSID アドレス・ファミリーの使用

AF_UNIX_CCSID アドレス・ファミリー・ソケットの仕様は、AF_UNIX アドレス・ファミリー・ソケットの仕様と同じです。これらは、コネクション型またはコネクションレス型に使用でき、同一システム上で通信を行えるようにします。詳細については、『AF_UNIX アドレス・ファミリーの使用』を参照してください。

AF_UNIX_CCSID ソケット・アプリケーションを処理する前に、出力形式を判別するために **Qlg_Path_Name_T** 構造に精通している必要があります。詳細については、Information Center の「API Reference」の『Path name structures』を参照してください。

出力アドレス構造 (**accept()**、**getsockname()**、**getpeername()**、**recvfrom()**、および **recvmsg()** から戻されるものなど) を処理する場合、アプリケーションはソケット・アドレス構造 (sockaddr_unc) を調べて、その形式を判別する必要があります。sunc_format および sunc_qlg フィールドによって、パス名の出力形式が決まります。しかし、ソケットは、アプリケーションが入力アドレスで使用したのと同じ値を必ずしも出力で使用するわけではありません。



ソケットのイベントのフロー: AF_UNIX_CCSID アドレス・ファミリーを使用するサーバー・アプリケーション

『例: AF_UNIX_CCSID アドレス・ファミリーを使用するサーバー・アプリケーション』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために UNIX_CCSID アドレス・ファミリーとストリーム・トランスポート (SOCK_STREAM) を使用することも示します。さらに、**socketpair()** 関数を使用して UNIX ソケットを初期化することもできます。

   AF_UNIX または AF_UNIX_CCSID は、唯一 **socketpair()** 関数をサポートしているアドレス・ファミリーです。**socketpair()** 関数は、名前がなく接続されている 2 つのソケット記述子を戻します。
2. ソケット記述子が作成された後、**bind()** 関数が、ソケットの固有名を取得します。

   UNIX ドメイン SOCKET のネーム・スペースはパス名で構成されています。ソケット・プログラムが **bind()** 関数を呼び出すと、ファイル・システム・ディレクトリーに項目が作成されます。パス名がすでに存在する場合、**bind()** は失敗します。そこで、UNIX ドメイン SOCKET プログラムが常に **unlink()** 関数を呼び出して、終了時にディレクトリーの項目を除去する必要があります。
3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。この例では、バックログが 10 に設定されています。これは、待ち行列に入れられた着信接続が 10 個になると、システムが着信要求を拒否するようになるということです。
4. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続を待機して、無期限にブロックします。
5. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。この例では、クライアントが 250 バイトのデータを送信してくることが分かっているものとします。これを踏まえて、SO_RCVLOWAT ソケット・オプションを使用し、250 バイトのデータがすべて到着するまで **recv()** がウェイクアップしないように指定できます。
6. **send()** 関数が、クライアントにデータを送り返します。
7. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。
8. **unlink()** 関数が、UNIX パス名をファイル・システムから除去します。

ソケットのイベントのフロー: AF_UNIX_CCSID アドレス・ファミリーを使用するクライアント・アプリケーション

『例: AF_UNIX_CCSID アドレス・ファミリーを使用するクライアント・アプリケーション』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために UNIX アドレス・ファミリーとストリーム・トランスポート (SOCK_STREAM) を使用することも示します。関数は、端点を表すソケット記述子を戻します。さらに、**socketpair()** 関数を使用して UNIX ソケットを初期化することもできます。

   AF_UNIX または AF_UNIX_CCSID は、唯一 **socketpair()** 関数をサポートしているアドレス・ファミリーです。**socketpair()** 関数は、名前がなく接続されている 2 つのソケット記述子を戻します。
2. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。
3. **send()** 関数が、サーバー・アプリケーションの SO_RCVLOWAT ソケット・オプションに指定されている 250 バイトのデータを送信します。
4. 250 バイトのデータが全部到着するまで、**recv()** 関数がループします。
5. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

例: AF_UNIX_CCSID アドレス・ファミリーを使用するサーバー・アプリケーション: 以下のサンプル・プログラムでは、AF_UNIX_CCSID アドレス・ファミリーを使用しています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This sample program provides code for a server application for         */
/* AF_UNIX_CCSID address family.                                             */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*****************************************************************************/
/* Constants used by this program                                            */
/*****************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /**************************************************************************/
   /* Variable and structure definitions.                                    */
   /**************************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_unc serveraddr;

   /**************************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The      */
   /* close() of each of the socket descriptors is only done once at the     */
   /* very end of the program.                                               */
   /**************************************************************************/
   do
   {
      /***********************************************************************/
      /* The socket() function returns a socket descriptor representing      */
      /* an endpoint.  The statement also identifies that the UNIX_CCSID     */
      /* address family with the stream transport (SOCK_STREAM) will be      */
      /* used for this socket.                                               */
      /***********************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
      if (sd < 0)
      {
         perror("socket() failed");
         break;
      }

      /***********************************************************************/
      /* After the socket descriptor is created, a bind() function gets a    */
      /* unique name for the socket.                                         */
      /***********************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sunc_family          = AF_UNIX_CCSID;
      serveraddr.sunc_format          = SO_UNC_USE_QLG;
      serveraddr.sunc_qlg.CCSID       = 500;
      serveraddr.sunc_qlg.Path_Type   = QLG_PTR_SINGLE;
      serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
      serveraddr.sunc_path.p_unix     = SERVER_PATH;
```

```
rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if (rc < 0)
{
   perror("bind() failed");
   break;
}

/*********************************************************************/
/* The listen() function allows the server to accept incoming       */
/* client connections.  In this example, the backlog is set to 10.  */
/* This means that the system will queue 10 incoming connection     */
/* requests before the system starts rejecting the incoming         */
/* requests.                                                        */
/*********************************************************************/
rc = listen(sd, 10);
if (rc< 0)
{
   perror("listen() failed");
   break;
}

printf("Ready for client connect().\n");

/*********************************************************************/
/* The server uses the accept() function to accept an incoming      */
/* connection request.  The accept() call will block indefinitely   */
/* waiting for the incoming connection to arrive.                   */
/*********************************************************************/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
   perror("accept() failed");
   break;
}

/*********************************************************************/
/* In this example we know that the client will send 250 bytes of   */
/* data over.  Knowing this, we can use the SO_RCVLOWAT socket      */
/* option and specify that we don't want our recv() to wake up      */
/* until all 250 bytes of data have arrived.                        */
/*********************************************************************/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                    (char *)&length, sizeof(length));
if (rc < 0)
{
   perror("setsockopt(SO_RCVLOWAT) failed");
   break;
}


/*********************************************************************/
/* Receive that 250 bytes data from the client                      */
/*********************************************************************/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("recv() failed");
   break;
}

printf("%d bytes of data were received\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
   printf("The client closed the connection before all of the\n");
   printf("data was sent\n");
```

```
         break;
      }

      /*******************************************************************/
      /* Echo the data back to the client                                */
      /*******************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("send() failed");
         break;
      }

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*******************************************************************/
   /* Close down any open socket descriptors                          */
   /*******************************************************************/
   if (sd != -1)
      close(sd);

   if (sd2 != -1)
      close(sd2);

   /*******************************************************************/
   /* Remove the UNIX path name from the file system                  */
   /*******************************************************************/
   unlink(SERVER_PATH);
}
```

例: AF_UNIX_CCSID アドレス・ファミリーを使用するクライアント・アプリケーション: 以下のサンプル・プログラムでは、AF_UNIX_CCSID アドレス・ファミリーを使用しています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* This sample program provides code for a client application for    */
/* AF_UNIX_CCSID address family.                                     */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <wcstr.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE    */
/* string, or set the server path in the   */
/* #define SERVER_PATH which is a CCSID    */
/* 500 string.                             */
void main(int argc, char *argv[])
{
```

```
/*********************************************************************/
/* Variable and structure definitions.                               */
/*********************************************************************/
int    sd=-1, rc, bytesReceived;
char   buffer[BUFFER_LENGTH];
struct sockaddr_unc serveraddr;

/*********************************************************************/
/* A do/while(FALSE) loop is used to make error cleanup easier.  The */
/* close() of the socket descriptor is only done once at the very end */
/* of the program.                                                   */
/*********************************************************************/
do
{
   /*********************************************************************/
   /* The socket() function returns a socket descriptor representing   */
   /* an endpoint.  The statement also identifies that the UNIX_CCSID  */
   /* address family with the stream transport (SOCK_STREAM) will be   */
   /* used for this socket.                                            */
   /*********************************************************************/
   sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
   if (sd < 0)
   {
      perror("socket() failed");
      break;
   }

   /*********************************************************************/
   /* If an argument was passed in, use this as the server, otherwise  */
   /* use the #define that is located at the top of this program.      */
   /*********************************************************************/
   memset(&serveraddr, 0, sizeof(serveraddr));
   serveraddr.sunc_family = AF_UNIX_CCSID;
   if (argc > 1)
   {
      /* The argument is a UNICODE path name.  Use the default format  */
      serveraddr.sunc_format = SO_UNC_DEFAULT;
      wcscpy(serveraddr.sunc_path.wide, (wchar_t *) argv[1]);
   }
   else
   {
      /* The local #define is CCSID 500.  Set the Qlg_Path_Name to use */
      /* the character format                                          */
      serveraddr.sunc_format          = SO_UNC_USE_QLG;
      serveraddr.sunc_qlg.CCSID       = 500;
      serveraddr.sunc_qlg.Path_Type   = QLG_CHAR_SINGLE;
      serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
      strcpy((char *)&serveraddr.sunc_path, SERVER_PATH);
   }
   /*********************************************************************/
   /* Use the connect() function to establish a connection to the      */
   /* server.                                                           */
   /*********************************************************************/
   rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
   if (rc < 0)
   {
      perror("connect() failed");
      break;
   }

   /*********************************************************************/
   /* Send 250 bytes of a's to the server                               */
   /*********************************************************************/
   memset(buffer, 'a', sizeof(buffer));
   rc = send(sd, buffer, sizeof(buffer), 0);
   if (rc < 0)
   {
```

```
         perror("send() failed");
         break;
      }

      /*******************************************************************/
      /* In this example we know that the server is going to respond with */
      /* the same 250 bytes that we just sent.  Since we know that 250    */
      /* bytes are going to be sent back to us, we could use the          */
      /* SO_RCVLOWAT socket option and then issue a single recv() and     */
      /* retrieve all of the data.                                        */
      /*                                                                  */
      /* The use of SO_RCVLOWAT is already illustrated in the server      */
      /* side of this example, so we will do something different here.    */
      /* The 250 bytes of the data may arrive in separate packets,        */
      /* therefore we will issue recv() over and over again until all     */
      /* 250 bytes have arrived.                                          */
      /*******************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("recv() failed");
            break;
         }
         else if (rc == 0)
         {
            printf("The server closed the connection\n");
            break;
         }

         /*******************************************************************/
         /* Increment the number of bytes that have been received so far  */
         /*******************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

AF_TELEPHONY アドレス・ファミリーの使用

電話アドレス・ファミリー (AF_TELEPHONY アドレス・ファミリーを使用するソケット) は、標準的なソケット API を使用する ISDN 回線ネットワークを通じて、ユーザーが通話を開始 (ダイヤル) および完了 (応答) するのを許可します。このドメインでの接続の端点を形成するソケットは、実際に通話の受信者 (受動端点) と送信者 (活動端点) になります。AF_TELEPHONY アドレスは最大 40 桁 (0 から 9) の電話番号で、sockaddr_tel アドレス構造に含まれます。

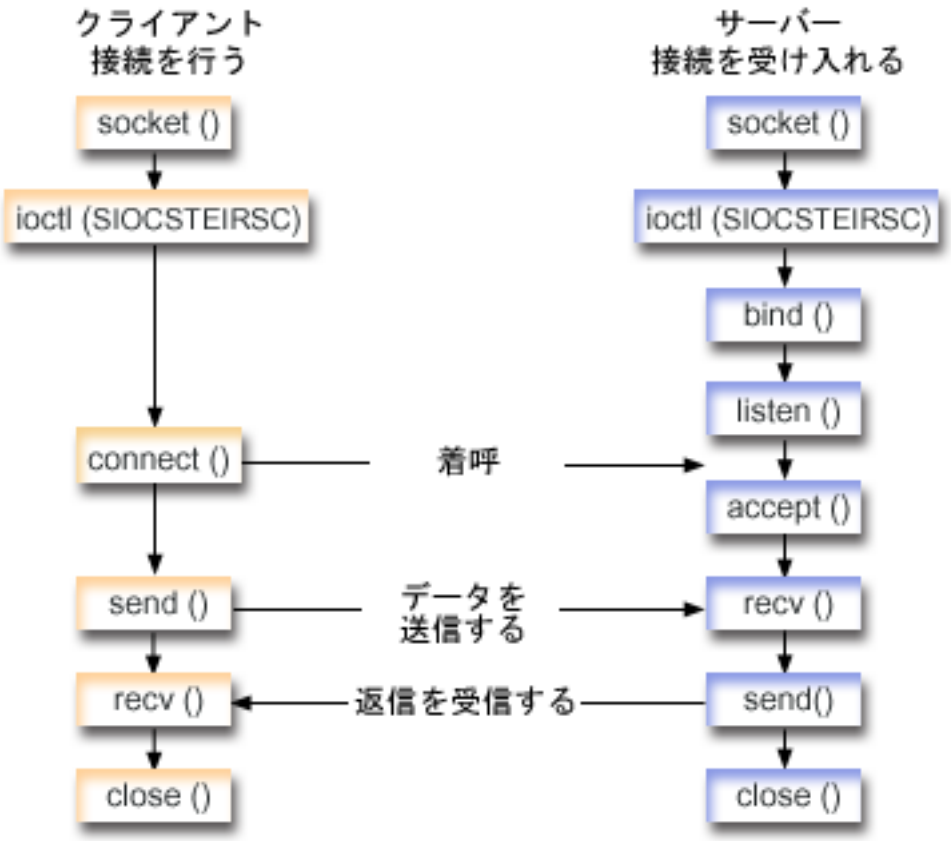
AF_TELEPHONY ソケットは、コネクション型 (タイプ SOCK_STREAM) ソケットとしてのみサポートされています。これらのソケットには、他のコネクション型プロトコルのものと類似したセマンティクスおよび関数があります。主な相違点は、電話ドメインでの接続の信頼性は、基盤となる電話接続の信頼性よりも低いということです。保証された配送が必要な場合は、アプリケーション・レベルで (たとえば、このファミリーを使用する FAX アプリケーションで) そのことを行う必要があります。また、アウト・オブ・バンド・データの概念は、電話アドレス・ファミリーではサポートされていません。

AF_TELEPHONY ソケットをネットワーク電話装置 (論理的には電話) に関連付けなければ、接続を開始または完了できません。この関連付けを行うには、**ioctl()** コマンド、**SIOCSTELRSC** (電話リソースの設定) を使用してください。このコマンドを発行する前に、これらの装置を構成して使用可能にしておく必要があります。

SIOCSTELRSC **ioctl()** 呼び出しを行う前に、アプリケーションは装置名を解決する必要があります。装置名をシステム・ポインターに解決し、このポインターを SIOCSTELRSC コマンドの入力として使用する必要があります。

ソケットがクローズされるまで、装置はそのソケットに関連付けられたままです。最後に、複数の装置を 1 つのソケットに関連付けることもできます。複数の装置を 1 つのソケットに関連付ければ、アプリケーションはそれら複数の装置で行われる呼び出しを 1 つのソケットから listen および応答できます。

以下の図は、AF_TELEPHONY アドレス・ファミリーで使用されるソケット呼び出しの関係を示します。AF_UNIX アドレス・ファミリーを使用する環境の設定に関する詳細は、『ソケット・プログラミングの前提条件』を参照してください。



ソケットのイベントのフロー: AF_TELEPHONY アドレス・ファミリーを使用するクライアント
『例: AF_TELEPHONY 接続の作成』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。
2. ソケットを装置に関連付ける必要があります。そのために、装置名をシステム・ポインターに解決し、要求の構造に書き込みを行います。 **ioctl()** 関数を発行し、装置とシステム・ポインターを関連付けます。
3. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。

4. **send()** 関数が、送信バッファーの内容をクライアントに送信します。
5. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。
6. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

ソケットのイベントのフロー: AF_TELEPHONY アドレス・ファミリーを使用するサーバー・アプリケーション

『例: AF_TELEPHONY 接続の受け入れ』は、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。
2. ソケットを装置に関連付ける必要があります。そのために、装置名をシステム・ポインターに解決し、要求の構造に書き込みを行います。 **ioctl()** 関数を発行し、装置とシステム・ポインターを関連付けます。
3. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。
4. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。
5. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。
6. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。
7. **send()** 関数が、送信バッファーの内容をクライアントに送信します。
8. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

例: AF_TELEPHONY 接続の作成: プログラムは、電話ドメイン・ソケットを介して互いに通信できます。以下のコードを使用して、ソケットがクライアントとの接続を確立できるようにします。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This sample program provides code to make AF_TELEPHONY connections.    */
/*****************************************************************************/

#include <stdio.h>                  /* String Functions           */
#include <string.h>                 /* String Functions           */
#include <miptrnam.h>               /* Pointer types              */

#include <sys/socket.h>             /* Sockets                    */
#include <nettel/tel.h>             /* Telephony address family   */
#include <errno.h>                  /* Error codes                */
#include <sys/ioctl.h>              /* Error codes                */

int main() {
  /*******************************************************************/
  /* Miscellaneous declares                                          */
  /*******************************************************************/
  int xSock, xRC, xLength;

  /*******************************************************************/
  /* Resolve device name to system pointer                           */
  /*******************************************************************/
  _SYSPTR pDev;                     /* System pointer to device    */
  _RSLV_Template_T xTemp;           /* Template for resolve       */
  char pName[]="FRED                        "; /* Device name     */
  struct TelResource xResource;     /* SIOCSTELRSC structure       */

  /*******************************************************************/
  /* Socket address structure                                        */
  /*******************************************************************/
  struct sockaddr_tel xAddr;

  /*******************************************************************/
  /* Buffers                                                         */
  /*******************************************************************/
```

```
char pSendBuffer[1024];
char pRecvBuffer[1024];

/*********************************************************************/
/* Open a socket                                                     */
/*********************************************************************/
xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
if (xSock<0) {
 perror("socket() failed");
 return(-1);
}

/*********************************************************************/
/* Associate the socket with a device                                */
/* ...resolve the device name to a system pointer                    */
/* ...fill in the structure for this request                         */
/* ...issue the ioctl to perform the association                     */
/*********************************************************************/
memset(&xTemp,0x00,sizeof(xTemp));
memcpy(xTemp.Obj.Name, pName, 30);
xTemp.Obj.Type_Subtype = WLI_DEVD;
xTemp.Auth = _AUTH_NONE;
_RSLVSP2(&pDev,&xTemp);

memset(&xResource,0x00,sizeof(xResource));
xResource.trCount=1;
xResource.trResourceList=&pDev;


xRC=ioctl(xSock,SIOCSTELRSC,&xResource);
if (xRC<0) {
  perror("ioctl() failed");
  close(xSock);
  return(-1);
}

/*********************************************************************/
/* Connect to a remote resource (dial a call)                        */
/*********************************************************************/
memset(&xAddr,0x00,sizeof(xAddr));
xAddr.stel_family=AF_TELEPHONY;
xAddr.stel_addr.t_len=11;
memcpy(xAddr.stel_addr.t_addr,"18005551212",11);
xRC=connect(xSock,(struct sockaddr*)&xAddr,sizeof(xAddr));
if (xRC<0) {
  perror("connect() failed");
  close(xSock);
  return(-1);
}

/*********************************************************************/
/* Send the contents of the send buffer                              */
/*********************************************************************/
xRC=send(xSock,pSendBuffer,1024,0);
if (xRC<0) {
  perror("send() failed");
  close(xSock);
  return(-1);
}

/*********************************************************************/
/* Receive a reply                                                   */
/*********************************************************************/
xRC=recv(xSock,pRecvBuffer,1024,0);
if (xRC<0) {
  perror("recv() failed");
  close(xSock);
```

```
    return(-1);
   }

  /*****************************************************************/
  /* All done, close and return                                    */
  /*****************************************************************/
  close(xSock);
  return(0);
}
```

例: AF_TELEPHONY 接続の受け入れ: AF_TELEPHONY アドレス・ファミリーは、電話番号を使用してソケットを識別するアプリケーションで使用されます。このアドレス・ファミリーは、主にファクシミリ・アプリケーションで使用されます。プログラムは、電話ドメイン・ソケットを介して互いに通信できます。以下のコードを使用して、ソケットがサーバーからの接続を受け入れられるようにします。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* This sample program provides code to accept AF_TELEPHONY connections.  */
/*********************************************************************/
#include  <stdio.h>                  /* String Functions */
#include  <string.h>                 /* String Functions */
#include  <miptrnam.h>               /* Pointer types */

#include <sys/socket.h>              /* Sockets */
#include <nettel/tel.h>              /* Telephony address family */
#include <errno.h>                   /* Error codes */
#include <sys/ioctl.h>               /* Error codes */

int main() {

  /*****************************************************************/
  /* Micellaneous declares                                         */
  /*****************************************************************/
  int xSock,xNewSock,xRC,xLength;

  /*****************************************************************/
  /* Resolve device name to system pointer data areas              */
  /*****************************************************************/
  _SYSPTR pDev;                      /* System pointer to device   */
  _RSLV_Template_T xTemp;            /* Template for resolve       */
  char pName[]="GEORGE                        "; /* Device name    */
  struct TelResource xResource;      /* SIOCSTELRSC structure      */

  /*****************************************************************/
  /* Socket address structure                                      */
  /*****************************************************************/
  struct sockaddr_tel xAddr;

  /*****************************************************************/
  /* Buffers                                                       */
  /*****************************************************************/
  char pSendBuffer[1024];
  char pRecvBuffer[1024];

  /*****************************************************************/
  /* Open a socket                                                  */
  /*****************************************************************/
  xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
  if (xSock<0) {
    perror("socket() failed");
    return(-1);
  }

  /*****************************************************************/
  /* ...first, resolve the device name to a system pointer         */
  /*  ...next, fill in the structure for this request              */
```

```
/* ...finally, issue the ioctl to perform the association       */
/*****************************************************************/
memset(&xTemp,0x00,sizeof(xTemp));
memcpy(xTemp.Obj.Name, pName, 30);
xTemp.Obj.Type_Subtype = WLI_DEVD;
xTemp.Auth = _AUTH_NONE;
_RSLVSP2(&pDev,&xTemp);

memset(&xResource,0x00,sizeof(xResource));
xResource.trCount=1;
xResource.trResourceList=&pDev;

xRC=ioctl(xSock,SIOCSTELRSC,&xResource);
if (xRC<0) {
  perror("ioctl() failed");
  close(xSock);
  return(-1);
}

/*****************************************************************/
/* Bind to a local number (using TELADDR_ANY means to accept     */
/* calls for any number in the inbound connection list's entries)*/
/*****************************************************************/
memset(&xAddr,0x00,sizeof(xAddr));
xAddr.stel_family=AF_TELEPHONY;
xAddr.stel_addr.t_len=TELADDR_LEN;
memcpy(xAddr.stel_addr.t_addr,TELADDR_ANY,TELADDR_LEN);
xRC=bind(xSock,(struct sockaddr*)&xAddr,sizeof(xAddr));
if (xRC<0) {
  perror("bind() failed");
  close(xSock);
  return(-1);
}

/*****************************************************************/
/* Listen for incoming calls                                     */
/*****************************************************************/
xRC=listen(xSock,5);
if (xRC<0) {
  perror("listen() failed");
  close(xSock);
  return(-1);
}
/*****************************************************************/
/* Accept an incoming call                                       */
/*****************************************************************/
memset(&xAddr,0x00,sizeof(xAddr));
xLength = sizeof(xAddr);
xNewSock=accept(xSock,(struct sockaddr*)&xAddr,&xLength);
if (xNewSock<0) {
  perror("accept() failed");
  close(xSock);
  return(-1);
}

/*****************************************************************/
/* Receive some data                                             */
/*****************************************************************/
xRC=recv(xNewSock,pRecvBuffer,1024,0);
if (xRC<0) {
  perror("recv() failed");
  close(xSock);
  close(xNewSock);
  return(-1);
}

/*****************************************************************/
```

```
  /* Send a reply                                                    */
  /*****************************************************************/
  xRC=send(xNewSock,pSendBuffer,1024,0);
  if (xRC<0) {
    perror("send() failed");
    close(xSock);
    close(xNewSock);
    return(-1);
  }

  /*****************************************************************/
  /* All done, close both sockets and return                       */
  /*****************************************************************/
  close(xSock);
  close(xNewSock);
  return(0);
}
```

ソケットの概念

以下のトピックでは、ソケットとその機能方法の一般的な説明よりもさらに詳細なソケットの拡張概念について説明します。これらのトピックでは、より大規模で複雑なネットワーク用のソケット・アプリケーションを設計する方法を説明します。以下の各概念は、対応するサンプル・プログラムにリンクしています。

- 非同期入出力
- セキュア・ソケット
- クライアント SOCKS サポート
- スレッド・セーフティー
- 非ブロッキング入出力
- 信号
- IP マルチキャスティング
- ファイル・データ転送 - send_file() および accept_and_recv()
- アウト・オブ・バンド・データ
- 入出力多重化 - select()
- ソケット・ネットワーク関数
- ドメイン・ネーム・システム (DNS) サポート
- BSD との互換性
- プロセス間での記述子の受け渡し - sendmsg() および recvmsg()

非同期入出力

非同期入出力 API は、スレッド化されたクライアント・サーバー・モデルに、高度な同時入出力およびメモリー効率のよい入出力を実行するための方法を提供します。以前のスレッド化されたクライアント/サーバー・モデルでは、一般に 2 つの入出力モデルが用いられていました。1 つ目のモデルでは、1 つのクライアント接続につき 1 つのスレッドを専用に割り当てていました。この 1 つ目のモデルは消費するスレッドの数が多すぎ、スリープおよびウェイクアップのコストがかなりかかることがありました。2 つ目のモデルでは、多くのクライアント接続のセットに対して **select()** API を発行することによって、また作動可能クライアント接続または要求をスレッドに委任することにより、スレッドの数が最小限にされます。この 2 つ目のモデルでは、以降の各選択を選択またはマークする必要があり、冗長作業の量がかなり多くなることがあります。

非同期入出力とオーバーラップ入出力では、ユーザー・アプリケーションに制御が戻った後でユーザー・バッファーとの間でデータをやり取りすることにより、これら 2 つのジレンマが解決されます。非同期入出力は、データの読み取りが可能になると、または接続でデータ転送の準備が可能な状態になると、これらのワーカー・スレッドに通知します。

非同期入出力の利点

- システム・リソースをより効率的に使用する。
  ユーザー・バッファーとやり取りするデータ・コピーは、要求を開始するアプリケーションと非同期で行われます。この並行処理によって、複数のプロセッサーが効率的に使用できるようになり、データ到着時にシステム・バッファーが再使用のために解放されるので、多くの場合ページング率が改善されます。
- プロセス/スレッドの待ち時間が最短になる。
- クライアント要求にサービスを即時に提供する。
- スリープおよびウェイクアップのコストを平均して少なくする。
- 「バースト性アプリケーション」を効率的に処理する。
- より優れたスケーラビリティーを提供する。
- 大規模なデータ転送の最も効率的な処理方法を提供する。
  QsoStartRecv() API の fillBuffer フラグは、非同期入出力を完了する前に、大量のデータを獲得するようにオペレーティング・システムに通知します。大量のデータを一度の非同期操作で送信することもできます。
- 必要なスレッドの数を最小限にする。
- オプションでタイマーを使い、この操作が非同期で完了するまでの最大時間を指定できる。設定された時間に渡ってクライアント接続のアイドル状態が続くと、サーバーはこの接続をクローズします。非同期タイマーを使うことにより、サーバーがこの時間制限を課すことができるようになります。
- **gsk_secure_soc_startInit()** API を使用して、セキュア・セッションを非同期に開始する。

表 13. 非同期入出力 API

関数	説明
gsk_secure_soc_startInit()	SSL 環境およびセキュア・セッションに設定された属性を使用して、セキュア・セッションのネゴシエーションを非同期に開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。
gsk_secure_soc_startRecv()	セキュア・セッションで非同期受信操作を開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。
gsk_secure_soc_startSend()	セキュア・セッションで非同期送信操作を開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。

表 13. 非同期入出力 API (続き)

QsoCreateIOCompletionPort()	完了非同期オーバーラップ入出力操作用の共通待機点を作成します。 **QsoCreateIOCompletionPort()** 関数は、待機点を表すポート・ハンドルを戻します。このハンドルは、非同期オーバーラップ入出力操作を開始するために、**QsoStartRecv()**、**QsoStartSend()**、**QsoStartAccept()**、**gsk_secure_soc_startRecv()**、または **gsk_secure_soc_startSend()** 関数で指定します。このハンドルは、関連する入出力完了ポートでイベントを通知するために、**QsoPostIOCompletion()** と共に使用することもできます。
QsoDestroyIOCompletionPort()	入出力完了ポートを破棄します。
QsoWaitForIOCompletionPort()	完了オーバーラップ入出力操作を待ちます。入出力完了ポートはこの待機点を表します。
QsoStartAccept()	非同期受け入れ操作を開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。
QsoStartRecv()	非同期受信操作を開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。
QsoStartSend()	非同期送信操作を開始します。 **注:** この API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でソケット・タイプが SOCK_STREAM のソケットのみです。
QsoPostIOCompletion()	アプリケーションが、何らかの関数または活動が発生したことを完了ポートを通知できるようにします。

非同期入出力の仕組み

アプリケーションは、**QsoCreateIOCompletionPort()** API を使用して入出力完了ポートを作成します。この API は、非同期入出力要求の完了をスケジュールして待機するために使用できるハンドルを戻します。アプリケーションは、入出力完了ポート・ハンドルを指定して、入力関数または出力関数を開始します。入出力の完了時に、状況情報とアプリケーション定義のハンドルが、指定した入出力完了ポートに通知されます。入出力完了ポートへの通知によって、おそらく多数ある待機中のスレッドのうちの 1 つだけがウェイクアップされます。アプリケーションは、以下を受信します。

- 元の要求で提供されたバッファー
- そのバッファーとやり取りして処理されたデータの長さ
- 完了した入出力操作のタイプの表示
- 初期入出力要求で渡されたアプリケーション定義のハンドル

このアプリケーション・ハンドルは、単にクライアント接続を識別するソケット記述子である場合もあれば、クライアント接続の状態についての広範な情報が入っているストレージを指すポインターである場合もあります。操作が完了してアプリケーション・ハンドルが渡されたので、ワーカー・スレッドはクライアント接続を完了するための次のステップを決定します。これらの完了非同期操作を処理するワーカー・スレッドは、1 つのクライアント要求だけに拘束されるのではなく、さまざまなクライアント要求を処理します。

ユーザー・バッファーとやり取りするコピーは、サーバー・プロセスと非同期で発生するので、クライアント要求の待機時間は減少します。これは、複数のプロセッサーがあるシステムでは利点があります。

非同期入出力を使用する単純なサーバー・モデルの例については、『例: 非同期入出力 API の使用』を参照してください。

セキュア・ソケット

現在、OS/400 は iSeries でセキュア・ソケット・アプリケーションを作成するための 2 つの方法をサポートしています。SSL_API および グローバル・セキュア・ツールキット (GSKit) API は、オープンな通信ネットワーク (たいていの場合はインターネット) での通信プライバシーを提供します。これらの API は、クライアント/サーバー・アプリケーションが盗聴、悪用、メッセージの偽造などを心配せずに通信できるようにすることを目的として設計された手段です。どちらもサーバー/クライアント認証をサポートしており、どちらのアプリケーションも Secure Sockets Layer (SSL) プロトコルを使用できます。ただし、GSKit API がすべての IBM @server プラットフォームでサポートされているのに対して、SSL_ API は OS/400 オペレーティング・システム固有のものです。プラットフォームをまたいで確実に相互運用できるよう、セキュア・ソケット接続用のアプリケーションを開発するときには、GSKit API を使用することをお勧めします。

これらの API については、それぞれ以下のトピックを参照してください。

- グローバル・セキュア・ツールキット (GSKit) API
- SSL_API

『セキュア・ソケット API のエラー・コード・メッセージ』トピックには、セキュア・ソケット API で生じる可能性のある一般的なエラー・コード・メッセージが記載されています。

セキュア・ソケットの概説

Secure Sockets Layer (SSL) プロトコルは元来 Netscape によって開発されたものであり、伝送制御プロトコル (TCP) のような信頼性の高いトランスポートの最上部で使用し、アプリケーションにセキュア通信を提供することを目的とした階層化プロトコルです。セキュア通信を必要とする多くのアプリケーションには、HTTP、FTP、SMTP、TELNET などがあります。

一般に、SSL 対応のアプリケーションは、SSL 非対応のアプリケーションとは別のポートを使用する必要があります。たとえば、SSL 対応のブラウザーは、″HTTP″ ではなく ″HTTPs″ で始まる URL を使用して、SSL 対応の HTTP サーバーにアクセスします。ほとんどの場合、″HTTPs″ という URL は、標準の HTTP サーバーが使用するポート 80 ではなく、サーバー・システムのポート 443 への接続をオープンしようとします。

複数のバージョンの SSL プロトコルが定義されています。最新バージョンの Transport Layer Security (TLS) バージョン 1.0 は、SSL バージョン 3.0 を大幅にアップグレードしたものです。iSeries 固有の SSL_API でも GSKit API でも、TLS バージョン 1.0、SSL バージョン 3.0 との互換性がある TLS バージョン 1.0、SSL バージョン 3.0、SSL バージョン 2.0、およびバージョン 2.0 と互換性のある SSL バージョン 3.0 がサポートされています。TLS バージョン 1.0 の詳細については、Internet Engineering Task Force (IETF) RFC 2246 の『Transport Layer Security』 を参照してください。

グローバル・セキュア・ツールキット (GSKit) API

グローバル・セキュア・ツールキット (GSKit) は、アプリケーションを SSL 化できるようにするプログラマブル・インターフェースのセットです。SSL_API と同様に、GSKit API を使用すれば、ソケット・アプリケーション・プログラムから SSL および TLS 関数にアクセスできます。しかし、GSKit API は、複数

の IBM @server プラットフォームでサポートされており、前述の SSL_ API よりプログラミングが容易です。加えて、セキュア・ソケット・セッションの非同期インスタンスを作成するための新しい GSKit API が追加されています。この API は、着信要求が多数で複数のジョブが必要になる場合に、複数のクライアントを処理するためのセキュア接続を提供します。ただし、この API は OS/400 固有のものであり、他の @server プラットフォームに移植できません。

注: これらの API がサポートするのは、アドレス・ファミリーが AF_INET または AF_INET6 でタイプが SOCK_STREAM のソケットのみです。

以下の表で、GSKit API を説明します。

表 14. グローバル・セキュア・ツールキット API

関数	説明
gsk_attribute_get_buffer()	セキュア・セッションまたは SSL 環境についての特定の文字ストリング情報 (証明書ストア・ファイル、証明書ストア・パスワード、アプリケーション ID、暗号など) を入手します。
gsk_attribute_get_cert_info()	セキュア・セッションまたは SSL 環境用のサーバー証明書またはクライアント証明書についての特定の情報を入手します。
gsk_attribute_get_enum_value()	セキュア・セッションまたは SSL 環境用の特定の列挙型データの値を入手します。
gsk_attribute_get_numeric_value()	セキュア・セッションまたは SSL 環境についての特定の数値情報を入手します。
gsk_attribute_set_buffer()	指定したバッファー属性を、指定したセキュア・セッションまたは SSL 環境内の値に設定します。
gsk_attribute_set_enum()	指定した列挙型タイプ属性を、セキュア・セッションまたは SSL 環境内の列挙型値に設定します。
gsk_attribute_set_numeric_value()	セキュア・セッションまたは SSL 環境用の特定の数値情報を設定します。
gsk_environment_close()	SSL 環境をクローズし、その環境に関連するすべてのストレージを解放します。
gsk_environment_init()	必須属性の設定後に SSL 環境を初期設定します。
gsk_environment_open()	保管して後続の gsk 呼び出しで使用する必要のある SSL 環境ハンドルを戻します。
gsk_secure_soc_close()	セキュア・セッションをクローズし、そのセキュア・セッションのすべての関連リソースを解放します。
gsk_secure_soc_init()	SSL 環境およびセキュア・セッションに設定された属性を使用して、セキュア・セッションをネゴシエーションします。
gsk_secure_soc_misc()	セキュア・セッション用の各種関数を実行します。
gsk_secure_soc_open()	セキュア・セッション用のストレージを入手し、属性のデフォルト値を設定し、保管してセキュア・セッションに関連した関数呼び出しで使用する必要のあるハンドルを戻します。
gsk_secure_soc_read()	セキュア・セッションからデータを受信します。

表 14. グローバル・セキュア・ツールキット API (続き)

gsk_secure_soc_startInit()	SSL 環境およびセキュア・セッションに設定された属性を使用して、セキュア・セッションのネゴシエーションを非同期に開始します。
gsk_secure_soc_write()	セキュア・セッションでデータを書き込みます。
gsk_secure_soc_startRecv()	セキュア・セッションで非同期受信操作を開始します。
gsk_secure_soc_startSend()	セキュア・セッションで非同期送信操作を開始します。
gsk_strerror()	エラー・メッセージおよび GSK API の呼び出しで戻された戻り値を記述する、関連したテキスト・ストリングを取り出します。

ソケットおよび GSKit API を使用するアプリケーションには、以下の要素が含まれています。

1. ソケット記述子を入手するための **socket()** への呼び出し。
2. SSL 環境へのハンドルを入手するための **gsk_environment_open()** への呼び出し。
3. SSL 環境の属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、GSK_OS400_APPLICATION_ID 値または GSK_KEYRING_FILE 値を設定するための、**gsk_attribute_set_buffer()** への呼び出し。どちらか一方の値のみを設定します。GSK_OS400_APPLICATION_ID 値を使用することを推奨します。 **gsk_attribute_set_enum()** を使用して、アプリケーション (クライアントまたはサーバー) のタイプ (GSK_SESSION_TYPE) も必ず設定してください。
4. **gsk_environment_init()** への呼び出し。この呼び出しは、SSL を処理するためのこの環境を初期設定し、この環境を使用して実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。
5. 接続を活動化させるためのソケット呼び出し。この呼び出しは、**connect()** を呼び出してクライアント・プログラムのために接続を活動化させたり、**bind()**、**listen()**、および **accept()** を呼び出して着信接続要求を受け入れるようサーバーを使用可能にします。
6. セキュア・セッションへのハンドルを入手するための **gsk_secure_soc_open()** への呼び出し。
7. セキュア・セッションの属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、特定のソケットをこのセキュア・セッションに関連付けるための **gsk_attribute_set_numeric_value()** への呼び出し。
8. 暗号パラメーターの SSL ハンドシェーク・ネゴシエーションを開始するための **gsk_secure_soc_init()** への呼び出し。

   注: 通常は、サーバー・プログラムが SSL ハンドシェークに必要な証明書を提示しないと、通信は成功しません。またサーバーは、サーバー証明書に関連した秘密鍵と、証明書が保管されているキー・データベース・ファイルへアクセスできなければなりません。場合によっては、SSL ハンドシェーク処理中にクライアントも証明書を提示しなければならないこともあります。そうなるのは、クライアントが接続しているサーバーで、クライアント認証が使用可能にされている場合です。 **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) または **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API 呼び出しは、ハンドシェーク中に使用される証明書および秘密鍵の入手先のキー・データベース・ファイルを (それぞれ異なる方法で) 識別します。
9. データを送受信するための **gsk_secure_soc_read()** および **gsk_secure_soc_write()** への呼び出し。
10. セキュア・セッションを終了するための **gsk_secure_soc_close()** への呼び出し。
11. SSL 環境をクローズするための **gsk_environment_close()** への呼び出し。

12. 接続ソケットを破棄するための **close()** への呼び出し。

GSKit API を使用する、以下のサンプル・プログラムを参照してください。

- 例: 非同期データ受信を使用する GSKit セキュア・サーバー
- 例: 非同期ハンドシェークを使用する GSKit セキュア・サーバー
- 例: グローバル・セキュア・ツールキット (GSKit) API によってセキュア・クライアントを確立する

SSL_API

SSL_ API を使用すると、プログラマーは iSeries 上でセキュア・ソケット・アプリケーションを作成できます。GSKit API とは異なり、SSL_ API は OS/400 システム固有のものに過ぎません。次の表で、OS/400 実装でサポートされている 9 つの SSL_API を説明します。Information Center の API 情報にリストされている個々の API の詳細を知るには、それぞれのリンクを使用してください。

表 15. SSL_API

関数	説明
SSL_Create()	指定されたソケット記述子が SSL サポートを得られるようにします。
SSL_Destroy()	指定された SSL セッションおよびソケットで SSL サポートを終了させます。
SSL_Handshake()	SSL ハンドシェーク・プロトコルを開始します。
SSL_Init()	SSL の現行ジョブを初期設定し、現行ジョブの SSL セキュリティー情報を設定します。 **注:** SSL を使用するには、まず **SSL_Init()** または **SSL_Init_Application()** API をプロセスで実行する必要があります。
SSL_Init_Application()	SSL の現行ジョブを初期設定し、現行ジョブの SSL セキュリティー情報を設定します。 **注:** SSL を使用するには、まず **SSL_Init()** または **SSL_Init_Application()** API をプロセスで実行する必要があります。
SSL_Read()	SSL 対応ソケット記述子からデータを受信します。
SSL_Write()	SSL 対応ソケット記述子にデータを書き込みます。
SSL_Strerror()	SSL 実行時エラー・メッセージを取り出します。
SSL_Perror()	SSL エラー・メッセージを印刷します。

ソケットおよび SSL_API を使用するアプリケーションには、以下の要素が含まれています。

- ソケット記述子を入手するための **socket()** への呼び出し。
- 呼び出し **SSL_Init()** または **SSL_Init_Application()**。この呼び出しは、SSL 処理用のジョブ環境を初期設定し、現行ジョブで実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。どちらか一方の API のみを使用します。 **SSL_Init_Application()** API を使用することを推奨します。
- 接続を活動化させるためのソケット呼び出し。この呼び出しは、**connect()** を呼び出してクライアント・プログラムのために接続を活動化させたり、**bind()**、**listen()**、および **accept()** を呼び出して着信接続要求を受け入れるようサーバーを使用可能にします。
- 接続ソケットが SSL サポートを得られるようにするための **SSL_Create()** への呼び出し。

- 暗号パラメーターの SSL ハンドシェーク・ネゴシエーションを開始するための **SSL_Handshake()** への呼び出し。

  注: 通常は、サーバー・プログラムが SSL ハンドシェークに必要な証明書を提示しないと、通信は成功しません。またサーバーは、サーバー証明書に関連した秘密鍵と、証明書が保管されているキー・データベース・ファイルへアクセスできなければなりません。場合によっては、SSL ハンドシェーク処理中にクライアントも証明書を提示しなければならないこともあります。そうなるのは、クライアントが接続しているサーバーで、クライアント認証が使用可能にされている場合です。 **SSL_Init()** または **SSL_Init_Application()** API は、ハンドシェーク中に使用される証明書および秘密鍵の入手先のキー・データベース・ファイルを (それぞれ異なる方法で) 識別します。
- データを送受信するための **SSL_Read()** および **SSL_Write()** への呼び出し。
- ソケットに対する SSL サポートを使用不可にするための **SSL_Destroy()** への呼び出し。
- 接続ソケットを破棄するための **close()** への呼び出し。

これらの SSL_ API を使用するサンプル・プログラムについては、以下を参照してください。

- 例: SSL_API によってセキュア・サーバーを確立する
- 例: SSL_API によってセキュア・クライアントを確立する

セキュア・ソケット API のエラー・コード・メッセージ

以下のセキュア・ソケットのエラー・コード・メッセージの情報を見るには、以下の事柄を実行してください。

1. コマンド行で、次のように入力します。

   ```
   DSPMSGD RANGE(XXXXXXX)
   ```

   ここで、XXXXXXX は戻りコードのメッセージ ID です。たとえば、戻りコードが 3 だった場合、次のように入力します。

   ```
   DSPMSGD RANGE(CPDBCB9)
   ```

2. **1** を選択して、メッセージ・テキストを表示します。

表 16. セキュア・ソケット API のエラー・コード・メッセージ

戻りコード	メッセージ ID	定数名
0	CPCBC80	GSK_OK
4	CPCBC80	GSK_INSUFFICIENT_STORAGE
502	CPE3406	GSK_WOULD_BLOCK
1	CPDBCA1	GSK_INVALID_HANDLE
2	CPDBCB3	GSK_API_NOT_AVAILABLE
3	CPDBCB9	GSK_INTERNAL_ERROR
5	CPDBC95	GSK_INVALID_STATE
107	CPDBC98	GSK_KEYFILE_CERT_EXPIRED
201	CPDBCA4	GSK_NO_KEYFILE_PASSWORD
202	CPDBCB5	GSK_KEYRING_OPEN_ERROR
301	CPDBCA5	GSK_CLOSE_FAILED
402	CPDBC81	GSK_ERROR_NO_CIPHERS
403	CPDBC82	GSK_ERROR_NO_CERTIFICATE
404	CPDBC84	GSK_ERROR_BAD_CERTIFICATE

表 16. セキュア・ソケット API のエラー・コード・メッセージ (続き)

405	CPDBC86	GSK_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
406	CPDBC8A	GSK_ERROR_IO
407	CPDBCA3	GSK_ERROR_BAD_KEYFILE_LABEL
408	CPDBCA7	GSK_ERROR_BAD_KEYFILE_PASSWORD
409	CPDBC9A	GSK_ERROR_BAD_KEY_LEN_FOR_EXPORT
410	CPDBC8B	GSK_ERROR_BAD_MESSAGE
411	CPDBC8C	GSK_ERROR_BAD_MAC
412	CPDBC8D	GSK_ERROR_UNSUPPORTED
414	CPDBC84	GSK_ERROR_BAD_CERT
415	CPDBC8B	GSK_ERROR_BAD_PEER
417	CPDBC92	GSK_ERROR_SELF_SIGNED
420	CPDBC96	GSK_ERROR_SOCKET_CLOSED
421	CPDBCB7	GSK_ERROR_BAD_V2_CIPHER
422	CPDBCB7	GSK_ERROR_BAD_V3_CIPHER
428	CPDBC82	GSK_ERROR_NO_PRIVATE_KEY
501	CPDBCA8	GSK_INVALID_BUFFER_SIZE
601	CPDBCAC	GSK_ERROR_NOT_SSLV3
602	CPDBCA9	GSK_MISC_INVALID_ID
701	CPDBCA9	GSK_ATTRIBUTE_INVALID_ID
702	CPDBCA6	GSK_ATTRIBUTE_INVALID_LENGTH
703	CPDBCAA	GSK_ATTRIBUTE_INVALID_ENUMERATION
705	CPDBCAB	GSK_ATTRIBUTE_INVALID_NUMERIC
6000	CPDBC97	GSK_OS400_ERROR_NOT_TRUSTED_ROOT
6001	CPDBCB1	GSK_OS400_ERROR_PASSWORD_EXPIRED
6002	CPDBCC9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPDBCAD	GSK_OS400_ERROR_NO_ACCESS
6004	CPDBCB8	GSK_OS400_ERROR_CLOSED
6005	CPDBCCB	GSK_OS400_ERROR_NO_CERTIFICATE_AUTHORITIES
6007	CPDBCB4	GSK_OS400_ERROR_NO_INITIALIZE
6008	CPDBCAE	GSK_OS400_ERROR_ALREADY_SECURE
6009	CPDBCAF	GSK_OS400_ERROR_NOT_TCP
6010	CPDBC9C	GSK_OS400_ERROR_INVALID_POINTER
6011	CPDBC9B	GSK_OS400_ERROR_TIMED_OUT
6012	CPCBCBA	GSK_OS400_ASYNCHRONOUS_RECV
6013	CPCBCBB	GSK_OS400_ASYNCHRONOUS_SEND
6014	CPDBCBC	GSK_OS400_ERROR_INVALID_OVERLAPPEDIO_T
6015	CPDBCBD	GSK_OS400_ERROR_INVALID_IOCOMPLETIONPORT
6016	CPDBCBE	GSK_OS400_ERROR_BAD_SOCKET_DESCRIPTOR
6017	CPDBCBF	GSK_OS400_ERROR_CERTIFICATE_REVOKED
6018	CPDBC87	GSK_OS400_ERROR_CRL_INVALID
6019	CPCBC88	GSK_OS400_ASYNCHRONOUS_SOC_INIT

表 16. セキュア・ソケット API のエラー・コード・メッセージ (続き)

0	CPCBC80	正常な戻りコード
-1	CPDBC81	SSL_ERROR_NO_CIPHERS
-2	CPDBC82	SSL_ERROR_NO_CERTIFICATE
-4	CPDBC84	SSL_ERROR_BAD_CERTIFICATE
-6	CPDBC86	SSL_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
-10	CPDBC8A	SSL_ERROR_IO
-11	CPDBC8B	SSL_ERROR_BAD_MESSAGE
-12	CPDBC8C	SSL_ERROR_BAD_MAC
-13	CPDBC8D	SSL_ERROR_UNSUPPORTED
-15	CPDBC84	SSL_ERROR_BAD_CERT (-4 にマップ)
-16	CPDBC8B	SSL_ERROR_BAD_PEER (-11 にマップ)
-18	CPDBC92	SSL_ERROR_SELF_SIGNED
-21	CPDBC95	SSL_ERROR_BAD_STATE
-22	CPDBC96	SSL_ERROR_SOCKET_CLOSED
-23	CPDBC97	SSL_ERROR_NOT_TRUSTED_ROOT
-24	CPDBC98	SSL_ERROR_CERT_EXPIRED
-26	CPDBC9A	SSL_ERROR_BAD_KEY_LEN_FOR_EXPORT
-91	CPDBCB1	SSL_ERROR_KEYPASSWORD_EXPIRED
-92	CPDBCB2	SSL_ERROR_CERTIFICATE_REJECTED
-93	CPDBCB3	SSL_ERROR_SSL_NOT_AVAILABLE
-94	CPDBCB4	SSL_ERROR_NO_INIT
-95	CPDBCB5	SSL_ERROR_NO_KEYRING
-97	CPDBCB7	SSL_ERROR_BAD_CIPHER_SUITE
-98	CPDBCB8	SSL_ERROR_CLOSED
-99	CPDBCB9	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPDBCD8	SSL_ERROR_NO_REUSE

クライアント SOCKS サポート

iSeries は、SOCKS バージョン 4 を使用することにより、SOCK_STREAM ソケット・タイプを指定した AF_INET アドレス・ファミリーを使用するプログラムが、ファイアウォールの外側のシステム上で実行されているサーバー・プログラムと通信できるようにします。ファイアウォールとは、ネットワーク管理者がセキュア内部ネットワークと非セキュア外部ネットワークとの間に置く高度なセキュア・ホストのことです。一般に、そのようなネットワーク構成では、セキュア・ホストから非セキュア・ネットワークへと、またはその逆へと経路指定される通信は許可されません。ファイアウォール上に置かれる proxy サーバーは、セキュア・ホストと非セキュア・ネットワークとの間で必要となる管理を援助します。

セキュア内部ネットワーク内のホストで実行されるアプリケーションは、自らの要求をファイアウォールの proxy サーバーに送信することによって、ファイアウォールまでナビゲートしなければなりません。それを受けて、proxy サーバーはそれらの要求を非セキュア・ネットワーク上の実サーバーに転送します。また、

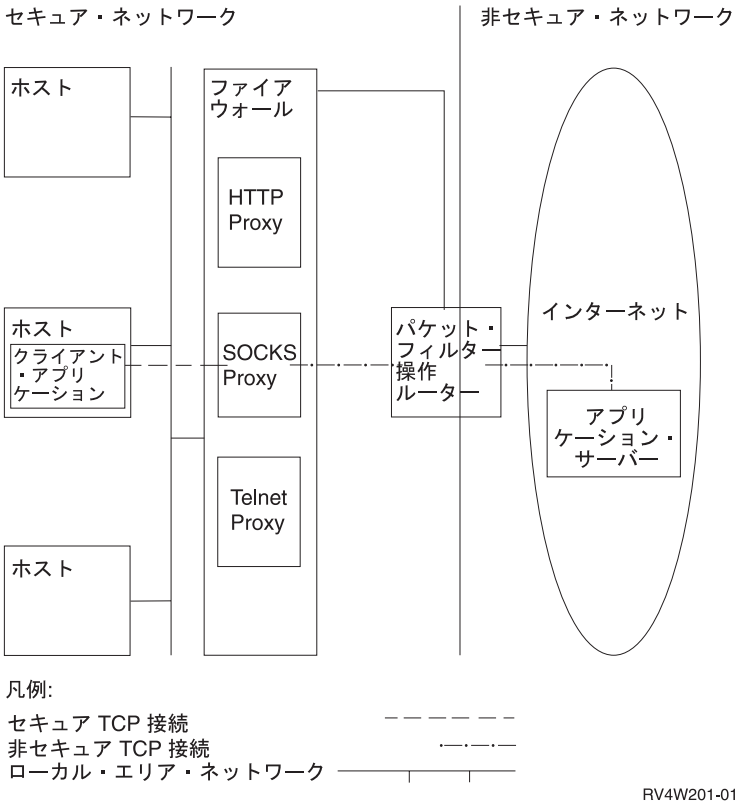
送信元ホストのアプリケーションに応答を戻すこともできます。 proxy サーバーの一般的な例は、HTTP proxy サーバーです。 proxy サーバーは、HTTP クライアントのために、以下のような数多くのタスクを実行します。

- proxy サーバーは、外部システムから内部ネットワークを隠します。
- proxy サーバーは、外部システムによる直接アクセスからホストを保護します。
- proxy サーバーは、適切に設計および構成されていれば、外部からのデータをフィルターに掛けることができます。

HTTP proxy サーバーは、HTTP クライアントのみを扱います。

1 つのファイアウォールで複数の proxy サーバーを実行する別の一般的な方法は、SOCKS サーバーとして知られる、より堅固な proxy サーバーを実行することです。 SOCKS サーバーは、ソケット API を使用して確立された TCP クライアント接続に対して、proxy としての役割を果たすことができます。 iSeries クライアント SOCKS サポートの大きな利点は、クライアント・コードを全く変更しなくても、クライアント・アプリケーションが SOCKS サーバーに透過的にアクセスできることです。

以下の図は、一般的なファイアウォールの配置を示しています。ファイアウォールには、HTTP proxy、Telnet proxy、SOCKS proxy が置かれています。インターネット上のサーバーにアクセスするセキュア・クライアントのために、2 つの TCP 接続が別々に使用されていることに注意してください。 1 つはセキュア・ホストから SOCKS サーバーへと、もう 1 つは非セキュア・ネットワークから SOCKS サーバーへと接続されています。



SOCKS サーバーを使用するためには、セキュア・クライアント・ホストで以下の 2 つのアクションを実行しなければなりません。

1. SOCKS サーバーの構成。IBM は 2000 年 2 月 15 日に、SOCKS サーバーのサポートを提供するファイアウォール iSeries 用 (5769-FW1) は、現在の V4R4 を最後に以降は機能強化を行わないことを発表しました。
2. セキュア・クライアント・システムにおいて、クライアント・システムの SOCKS サーバーに送信されるすべてのアウトバウンド・クライアント TCP 接続を定義します。セキュア・クライアントの SOCKS 構成項目は、iSeries Access 95 または Windows NT® の iSeries オペレーション・ナビゲーター機能の「SOCKS」タブを使用して構成できます。「SOCKS」タブには、セキュア・クライアント・システムをクライアント SOCKS サポート用に構成する場合に役立つヘルプがあります。

   クライアント SOCKS サポートを構成するには、以下を実行します。

   a. iSeries ナビゲーターで、「**iSeries/400 サーバー**」->「**ネットワーク**」->「**TCP/IP 構成**」と展開します。
   b. 「**TCP/IP 構成**」を右クリックします。
   c. 「**プロパティー**」をクリックします。
   d. 「**SOCKS**」タブをクリックします。
   e. 接続情報を「SOCKS」ページに入力します。

   注: セキュア・クライアント SOCKS 構成データは、セキュア・クライアント・ホスト・システムにあるライブラリー QUSRSYS の ファイル QASOSCFG に保管されます。

構成が完了すると、システムは SOCKS ページで指定された SOCKS サーバーへ、特定のアウトバウンド接続を自動的に送信します。システムが自動的に行うので、クライアント・アプリケーションに変更を加える必要はありません。これが要求を受信すると、SOCKS サーバーは非セキュア・ネットワーク内のサーバーに対して別の外部の TCP/IP 接続を確立します。それから SOCKS サーバーは、内部 TCP/IP 接続と外部 TCP/IP 接続の間でデータを中継します。

注: 非セキュア・ネットワーク上のリモート・ホストは直接 SOCKS サーバーに接続しますが、セキュア・クライアントに直接アクセスすることはできません。

ここまでは、セキュア・クライアントから送信される「アウトバウンド」TCP 接続のみを説明しました。クライアント SOCKS サポートは、SOCKS サーバーにファイアウォールを経由するインバウンド接続要求を許可させることもできます。セキュア・クライアントから **Rbind()** 呼び出しを出せば、この通信が可能になります。 **Rbind()** が操作を行えるようにするため、セキュア・クライアントはすでに **connect()** 呼び出しを発行済みで、その呼び出しの結果、SOCKS サーバーを介したアウトバウンド接続が行われている必要があります。 **Rbind()** インバウンド接続は、**connect()** が確立したアウトバウンド接続が宛先にしていたのと同じ IP アドレスからのものでなければなりません。

以下は、アプリケーションに対して透過的な SOCKS サーバーと、ソケット関数がどのように対話するかを分かりやすく示したものです。この例では、FTP クライアントが **bind()** 関数の代わりに、**Rbind()** 関数[1]を呼び出しています。この呼び出しを行うためには、 __Rbind プリプロセッサー #define (**bind()** を **Rbind()** として定義する) を使用して FTP クライアント・コードを再コンパイルします。別の方法として、アプリケーションが関連するソース・コードで **Rbind()** を明示的にコーディングすることもできます。アプリケーションが SOCKS サーバーからのインバウンド接続を必要としない場合は、**Rbind()** を使用しないでください。

注:

1. FTP クライアントは **Rbind()** を使用します。これは、FTP クライアントからのファイルまたはデータの送信要求に応えるため、FTP プロトコルは FTP サーバーがデータ接続を確立できるようにするからです。

2. SOCKS サーバーは FTP クライアントとのデータ接続を確立し、FTP クライアントと FTP サーバーの間でデータを中継します。ほとんどの SOCKS サーバーは、サーバーが一定の時間だけセキュア・クライアントに接続するのを許可します。サーバーがその時間内に接続しない場合は、**accept()** でエラー ECONNABORTED が戻されます。
3. FTP クライアントは、非セキュア・ネットワークへのアウトバウンド TCP 接続を SOCK サーバー経由で開始します。 FTP クライアントが connect() で指定する宛先アドレスは、非セキュア・ネットワークに置かれている FTP サーバーの IP アドレスおよびポートです。セキュア・ホスト・システムは SOCKS ページから構成されています。これにより、この接続を SOCKS サーバーから送信できます。構成が完了すると、システムは SOCKS ページで指定された SOCKS サーバーへ接続を自動的に送信します。
4. ソケットがオープンされ、インバウンド TCP 接続を確立するための **Rbind()** が呼び出されます。確立が完了すると、このインバウンド接続は上記で指定したのと同じ宛先アウトバウンド IP アドレスから接続します。同じスレッドについては、SOCKS サーバーを介するアウトバウンド接続とインバウンド接続は対になっていなければなりません。言い換えれば、すべての **Rbind()** インバウンド接続は、SOCKS サーバーを使用したアウトバウンド接続の直後に実行されなければならず、このスレッドに関係する非 SOCKS 介入接続は、**Rbind()** が実行される前に行うことはできません。
5. **getsockname()** は SOCKS サーバーのアドレスを戻します。ソケットは、SOCKS サーバーから選択したポートと対になっている SOCKS サーバー IP アドレスに論理的にバインドされます。この例では、アドレスは制御接続ソケット CTLed によって、非セキュア・ネットワークに置かれている FTP サーバーへ送信されます。これは FTP サーバーが接続されているアドレスです。 FTP サーバーは SOCKS サーバーに接続されますが、直接セキュア・ホストに接続されることはありません。
6. SOCKS サーバーは FTP クライアントとのデータ接続を確立し、FTP クライアントと FTP サーバーの間でデータを中継します。ほとんどの SOCKS サーバーは、サーバーが一定の時間だけセキュア・クライアントに接続するのを許可します。サーバーがその時間内に接続しない場合は、**accept()** でエラー ECONNABORTED が戻されます。

スレッド・セーフティー

同一プロセス内の複数のスレッドで同時に開始できれば、その関数はスレッド・セーフであると見なせます。関数がスレッド・セーフであるのは、その関数が呼び出すすべての関数もスレッド・セーフである場合のみです。ソケット API は、システム関数とネットワーク関数 (両方ともスレッド・セーフ) で構成されています。

名前の末尾に _r が付いているすべてのネットワーク関数も類似したセマンティクスを持ち、スレッド・セーフです。スレッド・セーフのソケット API を使用するサンプル・プログラムについては、『例: gethostbyaddr_r() を使用したスレッド・セーフ・ネットワーク・ルーチン』を参照してください。

その他の resolver ルーチンは互いにスレッド・セーフですが、それらは _res データ構造を使用します。このデータ構造は、1 つのプロセスのすべてのスレッド間で共用され、リゾルバー呼び出しの間にアプリケーションによって変更することができます。 resolver ルーチンを使用するサンプル・プログラムについては、『例: DNS の更新および照会』を参照してください。

非ブロッキング入出力

アプリケーションがソケット入力関数の 1 つを発行し、読み取るデータがない場合、関数がブロック化し、読み取るデータができるまで戻りません。同様に、データを即時に送信できない場合、アプリケーションはソケット出力関数をブロックできます。最終的に **connect()** および **accept()** は、パートナーのプログラムとの接続の確立を待機している間にブロックできます。

ソケットは、アプリケーション・プログラムが、ブロック化する関数を発行して遅延なく関数が戻るようにする方法を提供します。これは、**O_NONBLOCK** フラグをオンにするために **fcntl()** を呼び出すか、**FIONBIO** フラグをオンにするために **ioctl()** を呼び出すことによって行われます。非ブロッキング・モードを実行すると、関数がブロック化せずに完了できない場合は、関数は即時に戻ります。 **connect()** が **[EINPROGRESS]** と一緒に戻ることがあります。これは、接続が開始済みであることを示します。 **select()** を使用して接続完了時を判別することができます。 非ブロッキング・モードでの実行に影響を受ける他のすべての関数については、**[EWOULDBLOCK]** というエラー・コードが、呼び出しが成功しなかったことを示します。

以下のソケット関数で非ブロッキングを使用できます。

- **accept()**
- **connect()**
- **gsk_secure_soc_read()**
- **gsk_secure_soc_write()**
- **read()**
- **readv()**
- **recv()**
- **recvfrom()**
- **recvmsg()**
- **send()**
- **send_file()**
- **send_file64()**
- **sendmsg()**
- **sendto()**
- **SSL_Read()**
- **SSL_Write()**
- **write()**
- **writev()**

非ブロッキング入出力を使用するサンプル・プログラムについては、『例: 非ブロッキング入出力および select()』を参照してください。

信号

アプリケーション・プログラムは、アプリケーションがかかわる条件が発生するときに、非同期に通知することを要求 (システムが**信号**を送信するように要求) できます。ソケットがアプリケーションに送信する非同期信号は、2 つあります。

1. **SIGURG** は、アウト・オブ・バンド (OOB) データの概念をサポートするソケットで OOB データが受信されたときに送信される信号です。たとえば、AF_INET アドレス・ファミリーで SOCK_STREAM タイプのソケットは、SIGURG 信号を送信するように条件付けることができます。
2. **SIGIO** は、あらゆるタイプのソケットで通常のデータ、OOB データ、エラー条件、その他ほとんどすべてのことが発生したときに送信される信号です。

アプリケーションは、信号の送信をシステムに要求する前に、信号の受信を処理できることを確認する必要があります。これは、**信号ハンドラー**を設定することによって実行します。信号ハンドラーを設定する方法の 1 つに、**sigaction()** 呼び出しを発行する方法があります。

アプリケーションは、次の方法の 1 つを使用して、システムに **SIGURG** 信号を送信するように要求します。

- **fcntl()** 呼び出しを発行し、F_SETOWN コマンドを使用して、プロセス ID またはプロセス・グループ ID を指定する。
- **ioctl()** 呼び出しを発行し、FIOSETOWN または SIOCSPGRP コマンド (要求) 値を指定する。

アプリケーションは、**SIGIO** 信号を 2 段階で送信するように要求します。最初に、**SIGURG** 信号について上記で説明したように、プロセス ID あるいはプロセス・グループ ID を設定しなければなりません。これは、アプリケーションがどこへ信号を転送したいかをシステムに通知するためです。次に、アプリケーションは次のうちのどちらかを実行しなければなりません。

- **fcntl()** 呼び出しを発行し、FASYNC フラグを付けて F_SETFL コマンドを指定する。
- **ioctl()** 呼び出しを発行し、FIOASYNC コマンドを指定する。

このステップは、SIGIO 信号を生成するようシステムに要求します。これらのステップは任意の順序で実行できます。また、listen 中のソケットでアプリケーションがこれらの要求を出す場合には、要求で設定した値は、**accept()** 関数からアプリケーションに戻されるすべてのソケットに継承されることにも注意してください。すなわち、新たに受け付けたソケットも、SIGIO 信号の送信に関する情報と同様に、同じプロセス ID あるいは同じプロセス・グループ ID を持つことになります。

ソケットは、エラー条件に関する同期信号を生成することもできます。アプリケーションがソケット関数の **errno** として [EPIPE] を受け取るときは常に、 SIGPIPE 信号も **errno** 値を受け取る命令を出したプロセスに転送されます。BSD 実装のデフォルトでは、 SIGPIPE 信号により、**errno** 値を受け取ったプロセスは終了します。OS/400 の前のリリースとの互換性を維持するために、OS/400 では、 SIGPIPE 信号の無視というデフォルトの動作を使用します。これによって、シグナル関数を追加しても、既存のアプリケーションが悪影響を受けることはありません。

ソケット関数でブロックされているプロセスに信号が転送されると、この関数は **[EINTR] errno** 値と共に待ち状態から戻り、アプリケーションのシグナル・ハンドラーが実行できるようになります。これが発生する関数は以下のとおりです。

- **accept()**
- **connect()**
- **read()**
- **readv()**
- **recv()**
- **recvfrom()**
- **recvmsg()**
- **select()**
- **send()**
- **sendto()**
- **sendmsg()**
- **write()**
- **writev()**

信号は、信号によって示される条件が実際にどこに存在するかを示すソケット記述子を、アプリケーション・プログラムには提供しないという点に注意することは重要です。したがって、このように、アプリケーション・プログラムが複数のソケット記述子を使用している場合、記述子をポーリングするかあるいは、**select()** 呼び出しを使用してなぜ信号が受信されたかを判断する必要があります。

信号を使用するサンプル・プログラムについては、『例: ブロック化ソケット API での信号の使用』を参照してください。

IP マルチキャスティング

IP マルチキャスティングは、ネットワークにあるホストのグループが受信できる、アプリケーションによる単一の IP データグラムの送信を可能にします。グループにあるホストは、単一のサブネットに常駐する場合も、マルチキャスト機能のあるルーターが接続する異なるサブネットに位置する場合もあります。ホストはいつでもグループに結合したり分離したりできます。ホスト・グループでのメンバーの位置や数については、制限はありません。 224.0.0.1 から 239.255.255.255 の範囲のクラス D の IP アドレスは、ホスト・グループを識別します。

現在は、AF_INET アドレス・ファミリーでしか IP マルチキャスティングを使用できません。

アプリケーション・プログラムは、ソケット API およびコネクションレス型 SOCK_DGRAM タイプ・ソケットを使用することによりマルチキャスト・データグラムを送受信できます。マルチキャストは、1 対多の伝送方式です。マルチキャストには、タイプ SOCK_STREAM のコネクション型ソケットを使用することはできません。タイプ SOCK_DGRAM のソケットが作成されると、アプリケーションは **setsockopt()** 関数を使用して、このソケットに関連するマルチキャスト特性を制御することができます。**setsockopt()** 関数は、以下の IPPROTO_IP レベル・フラグを受け取ります。

- IP_ADD_MEMBERSHIP: 指定されたマルチキャスト・グループを結合させます。
- IP_DROP_MEMBERSHIP: 指定されたマルチキャスト・グループを出ます。
- IP_MULTICAST_IF: 発信マルチキャスト・データグラムが送信されるインターフェースを設定します。
- IP_MULTICAST_TTL: 発信マルチキャスト・データグラムについて IP ヘッダーの存続時間 (TTL) を設定します。
- IP_MULTICAST_LOOP: 発信マルチキャスト・データグラムのコピーがマルチキャスト・グループのメンバーであるかぎり送信しているホストに送達されるようにするかどうかを指定します。

IP マルチキャスティングの例については、以下の例『例: マルチキャスティングの使用』を参照してください。

ファイル・データ転送 - send_file() および accept_and_recv()

OS/400 のソケットは **send_file()** および **accept_and_recv()** API を備えていますが、これらを使用すれば、接続ソケットにファイルを高速かつ簡単に転送できます。これら 2 つの API は、Hypertext Transfer Protocol (HTTP) サーバーなどのファイル処理アプリケーションで特に便利です。

send_file() を使用すれば、たった 1 回の API 呼び出しで、ファイル・データを直接ファイル・システムから接続ソケットへ送信できます。

accept_and_recv() は、次の 3 つのソケット関数の組み合わせです。すなわち、**accept()**、 **getsockname()**、および **recv()** の 3 つです。

send_file() および **accept_and_recv()** API のサンプル・プログラムについては、『例: send_file() および accept_and_recv() API を使用したファイル・データの転送』を参照してください。

アウト・オブ・バンド・データ

アウト・オブ・バンド (OOB) データは、コネクション型 (ストリーム) ソケットにのみ意味のあるユーザー固有のデータです。ストリーム・データは、一般に送信された順序で受信されます。OOB データはストリーム内の位置に関係なく (および送信時の順序に関係なく) 受信されます。これが可能なのは、データがプログラム A からプログラム B に送信される場合、プログラム B にデータの到着を通知するようにマーク付けられているためです。

OOB データは、AF_INET (SOCK_STREAM) と AF_INET6 (SOCK_STREAM) でしかサポートされていません。

OOB データは **send()**、**sendto()**、および **sendmsg()** 関数で MSG_OOB フラグを指定することにより送信されます。

OOB データの伝送は通常のデータの伝送と同じです。バッファーに入れたデータの後に送信されます。つまり、OOB データがバッファーに入れられるデータよりも優先されることはなく、データは送信順に伝送されます。

受信側の環境はやや複雑になっています。

- ソケット API が、OOB マーカーを使用してシステムで受信される OOB データのトラックを保持します。OOB マーカーは、送信された OOB データの最終バイトを指します。

  注: OOB マーカーが指しているバイトを表す値は、システムの基本に設定されています (すべてのアプリケーションがこの値を使用します)。値は TCP 接続のローカル端末とリモート端末間で一致していなければなりません。この値を使用するソケット・アプリケーションは、クライアント・アプリケーションとサーバー・アプリケーション間で一貫してその値を使用しなければなりません。OOB マーカーが指すバイトの変更方法については、Information Center の『Change TCP Attributes (CHGTCPA) command』を参照してください。

  SIOCATMARK ioctl() 要求は、読み取りポインターが最終 OOB バイトを指しているかどうかを決定します。

  注: OOB データの複数オカレンスが送信されると、OOB マーカーは最終オカレンスの最終 OOB バイトを指します。
- OOB データがインラインで受信されるかどうかにかかわらず、OOB データが送信されると、入力操作でデータが OOB マーカーまで処理されます。
- (MSG_OOB フラグが設定されている) **recv()**、**recvmsg()**、または **recvfrom()** 関数を使用して OOB データを受信できます。受信関数の 1 つが完了して以下のいずれかが起きた場合、[EINVAL] エラーが戻されます。
  - ソケット・オプションの SO_OOBINLINE が設定されておらず、受信される OOB データがない。
  - ソケット・オプションの SO_OOBINLINE が設定されている。

  ソケット・オプションの SO_OOBINLINE が設定されておらず、送信プログラムが 1 バイトを超えるサイズの OOB データを送信した場合は、最終バイト以外のすべてのバイトは通常データであると見なされます。 (通常データとは、受信プログラムが MSG_OOB フラグの指定なしで受信できるデータのことです。) 送信された OOB データの最終バイトは、通常データ・ストリームには保管されません。このバイトを検索する唯一の方法は、 MSG_OOB フラグが設定されている **recv()**、**recvmsg()**、または **recvfrom()** 関数を発行するという方法です。 MSG_OOB フラグが設定されずに受信関数が発行され、通

常データが受信された場合、OOB バイトは削除されます。また、OOB データの複数オカレンスが送信された場合、先のオカレンスの OOB データは失われ、最終 OOB データ・オカレンスの OOB データの位置が記憶されます。

ソケット・オプションの SO_OOBINLINE を設定すると、送信されたすべての OOB データが通常データ・ストリームに保管されます。データを検索するには、上記の 3 つの関数のうち 1 つを MSG_OOB フラグを設定せずに発行します (このフラグを指定すると、エラー [EINVAL] が戻されます)。OOB データの複数オカレンスが送信される場合は、OOB データは失われません。

- SO_OOBINLINE を設定しておらず、OOB データがすでに受信されていて、その後ユーザーが SO_OOBINLINE をオンに設定する場合は、OOB データは破棄されません。最初の OOB バイトは、通常データと見なされます。
- SO_OOBINLINE を設定しないで OOB データが送信され、受信プログラムが入力関数を発行して OOB データを受信した場合、OOB マーカーは有効のままです。OOB バイトが受信されても、受信プログラムは読み取りポインターが OOB マーカーにあるかどうかをチェックできます。

入出力多重化 - select()

非同期入出力によって、アプリケーション・リソースをさらに効率的な方法で最大限に活用できるので、**select()** API ではなく、非同期入出力 API を使用することをお勧めします。ただし、特定のアプリケーション設計によっては **select()** を使用できます。非同期入出力と同様に **select()** は、同時に複数の条件で待機する共通点を作成します。ただし、**select()** によって、アプリケーションは次のことを行うために一連の記述子を指定することができます。

- 読み取るデータがあるかどうかを確かめる。
- データを書き込めるかどうかを確かめる。
- 例外条件があるかどうかを確かめる。

それぞれのセットに指定できる記述子は、ソケット記述子、ファイル記述子、または記述子で表される他のオブジェクトになることができます。

データが利用可能になるのを待とうとする場合、**select()** 関数によってアプリケーションが指定できます。アプリケーションはどれくらい待つべきか指定することが可能です。サンプル・プログラムについては、『例: 非ブロッキング入出力および select()』を参照してください。

ソケット・ネットワーク関数

ソケット・ネットワーク関数を使用して、アプリケーション・プログラムはホスト、プロトコル、サービス、およびネットワーク・ファイルから情報を獲得することができます。情報には、名前、アドレス、またはファイルの順次アクセスによってアクセスできます。これらのネットワーク関数 (またはルーチン) は、複数のネットワーク内で実行されるプログラム間の通信をセットアップする場合は必須であり、AF_UNIX ソケットによって使用されることはありません。これらの個々のネットワーク関数ルーチンの要約については、Information Center の「API Reference」トピックの『Sockets Network Functions (Routines)』を参照してください。

ルーチンは以下のことを行います。

- ホスト名をネットワーク・アドレスにマップする。
- ネットワーク名をネットワーク番号にマップする。
- プロトコル名をプロトコル番号にマップする。
- サービス名をポート番号にマップする。

- インターネット・ネットワーク・アドレスのバイト・オーダーを変換する。
- IP アドレスおよびドット 10 進表記を変換する。

resolver ルーチンと呼ばれる一連のルーチン・グループはネットワーク・ルーチンに含まれています。これらのルーチンは、インターネット・ドメインでネーム・サーバー用のパケットを作成、送信、および解釈し、名前を解決するためにも使用されます。resolver ルーチンは、通常、**gethostbyname()**、**gethostbyaddr()**、**getnameinfo()**、および **getaddrinfo()** によって呼び出されますが、直接呼び出すことも可能です。これらの resolver ルーチンを使用する例については、『例: gethostbyaddr_r() を使用したスレッド・セーフ・ネットワーク・ルーチン』を参照してください。resolver ルーチンは主に、ソケット・アプリケーションを介してドメイン・ネーム・システム (DNS) にアクセスするために使用されます。DNS でソケットを使用できる方法の詳細については、『ドメイン・ネーム・システム (DNS) サポート』を参照してください。

ドメイン・ネーム・システム (DNS) サポート

iSeries は、リゾルバー関数を介してドメイン・ネーム・システム (DNS) にアクセスするアプリケーションを提供します。DNS には、以下の 3 つの主な構成要素があります。

- **ドメイン・ネーム・スペースおよびリソース・レコード**
  ツリー構造名スペースおよび名前に関連したデータを指定している。
- **ネーム・サーバー**
  ドメイン・ツリー構造についての情報を保持し、情報を設定するサーバー・プログラム。ネーム・サーバーの詳細については、Information Center の『DNS』トピックを参照してください。
- **リゾルバー**
  クライアント要求に対する応答においてネーム・サーバーからの情報を取り出すプログラム。

OS/400 実装で提供のリゾルバーは、ネーム・サーバーとの接続を提供するソケット関数です。これらのルーチンを使用すると、パケットの作成、送信、更新、解釈、およびパフォーマンスのための名前キャッシングの実行を行えます。これらのルーチンは、ASCII から EBCDIC への変換および EBCDIC から ASCII への変換を行うための関数も提供します。オプションで、リゾルバーは DNS とのセキュア通信を行うためにトランザクション署名 (TSIG) を使用します。個々の resolver ルーチンの要約については、Information Center の「API Reference」トピックの『Sockets Network Functions (Routines)』を参照してください。このリンクには、_res 構造についての情報もあります。_res 構造には、これらのルーチンで使用されるグローバル情報が入っています。

ドメイン・ネームの詳細については、以下の RFC を参照してください。これらは RFC 検索ページにあります。

- RFC 1034, Domain names - concepts and facilities
- RFC 1035, Domain names - implementation and specification
- RFC 1886, DNS Extensions to support IP version 6
- RFC 2136, Dynamic Updates in the Domain Name System (DNS UPDATE)
- RFC 2181, Clarifications to the DNS Specification
- RFC 2845, Secret Key Transaction Authentication for DNS (TSIG)
- RFC 3152, DNS Delegation of IP6.ARPA

ソケット・アプリケーションで DNS を使用するための別の方法については、以下のトピックを参照してください。

- 環境変数
  このトピックは、ネーム・レゾリューションに使用できる環境変数について説明します。
- データ・キャッシュ
  このトピックは、ソケットを使用して、ネットワーク上のトラフィックの量を減らすために、DNS 照会に対する応答をキャッシュすることについて詳細に説明します。『例: DNS の更新および照会』には、ソケット API によって DNS レコードを照会し更新する方法を示すサンプル・プログラムが記載されています。

環境変数

環境変数を使用して、リゾルバー関数のデフォルトの初期化を指定変更できます。環境変数の検査は、**res_init()** または **res_ninit()** の呼び出しが成功した後に初めて行われます。よって、構造が手動で初期化された場合、環境変数は無視されます。また構造が初期化されるのは一度だけなので、後から環境変数に変更を加えても無視されるということにも注意してください。

注: 環境変数の名前は英大文字でなければなりません。ストリング値は英大文字小文字混合でもかまいません。CCSID 290 を使用する日本語システムの場合は、環境変数名と値の両方について、上段シフト文字と番号のみを使用してください。以下に、**res_init()** API と **res_ninit()** API で使用可能な環境変数を説明します。

LOCALDOMAIN
この環境変数には、6 つの検索ドメインのリストを設定します。各ドメインはスペースで区切り (スペースを含めて)、合計 256 文字の長さにします。これによって、構成された検索リスト (struct state.defdname および struct state.dnsrch) が指定変更されます。検索リストが指定した場合、デフォルトのローカル・ドメインは照会に使用されません。

RES_OPTIONS
特定の内部リゾルバー変数を変更できるようにします。この環境変数には、以下のオプションを 1 つ以上設定できます。それぞれのオプションはスペースで区切ります。

- **NDOTS:n**
  ドットの数のしきい値を設定します。 **res_query()** に指定された名前に含まれるドットの数がこのしきい値に達すると、最初の絶対照会が行われます。n のデフォルトは ″1″ です。これは、名前にドットが 1 つでもあれば、検索リストの要素を追加する前に、その名前を絶対名としてまず試してみるということです。
- **TIMEOUT:n**
  リゾルバーがリモート・ネーム・サーバーからの応答を待機する時間を設定します (秒単位)。この時間が経過すると、リゾルバーは待つのをやめて、照会を再試行します。
- **ATTEMPTS:n**
  リゾルバーが、指定されたネーム・サーバーへ送信する照会の数を設定します。送信した照会がこの数に達すると、リゾルバーはこのネーム・サーバーへの照会をやめ、次にリストされているネーム・サーバーに送信を行います。
- **ROTATE**
  _res.options の RES_ROTATE を設定します。これは、リストされているネーム・サーバーを順番に選択するためのものです。これにより、すべてのクライアントが毎回リストの最初のサーバーに照会を試みる代わりに、リストされているサーバーすべての間で照会の負荷の均衡を図ることができます。

- **NO-CHECK-NAMES**
  _res.options の RES_NOCHECKNAME を設定します。これは、着信するホスト名やメールの名前に無効な文字 (下線 (_)、非 ASCII、または制御文字など) が含まれていないか調べるという、現代的な BIND 検査を使用不可にします。

QIBM_BIND_RESOLVER_FLAGS
この環境変数には、リゾルバーのオプション・フラグのリストを設定します。各フラグはスペースで区切ります。この環境変数によって、RES_DEFAULT オプション (struct state.options) と、システム構成値 (TCP/IP ドメインの変更 - CHGTCPDMN) が指定変更されます。state.options 構造は、RES_DEFAULT、OPTIONS 環境値および CHGTCPDMN 構成値を使用して、普通に初期化されます。それから、これらのデフォルトを指定変更するために、この環境変数が使用されます。この環境変数に指定されたフラグの前に '+'、'-'、または 'NOT_' を付けることにより、値を設定 ('+' の場合)、またはリセット ('-' と 'NOT_' の場合) できます。

たとえば、RES_NOCHECKNAME をオンにして RES_ROTATE をオフにするには、文字ベースのインターフェースから以下のコマンドを使用します。

```
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('RES_NOCHECKNAME NOT_RES_ROTATE')
```

または

```
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('+RES_NOCHECKNAME -RES_ROTATE')
```

QIBM_BIND_RESOLVER_SORTLIST
この環境変数には IP アドレス/マスクの対を設定して、ソート・リスト (struct state.sort_list) を作成します。IP アドレス/マスクの対は、ドット 10 進形式 (9.5.9.0/255.255.255.0) で、最大 10 個まで指定できます (スペース区切り)。

データ・キャッシュ

DNS 照会に対する応答のキャッシングは、ネットワーク・トラフィックの量を少なくするために、OS/400 ソケットによって行われます。キャッシュは必要に応じて追加および更新されます。

_res オプションに RES_AAONLY (権限のある回答のみ) を設定した場合、照会は常にネットワークに送信されます。この場合、回答についてキャッシュを検査することはありません。 RES_AAONLY を設定しない場合、ネットワークへの送信が実行される前に、キャッシュは照会に対する回答の検査をします。回答が検出され、存続時間が満了していない場合は、回答は照会への回答としてユーザーに戻されます。存続時間が満了した場合は、項目が除去され、ネットワークに照会が送信されます。キャッシュに回答が見つからない場合も、照会はネットワークに送信されます。

応答に権限があれば、ネットワークからの回答はキャッシュされます。権限のない回答はキャッシュされません。逆照会の結果として受信される応答もキャッシュされません。CHGTCPDMN、CFGTCP オプション 12、iSeries ナビゲーターのいずれかを使用して DNS 構成を更新すれば、このキャッシュをクリアできます。

データ・キャッシングを使用するプログラム例については、『例: DNS の更新および照会』を参照してください。

バークレー・ソフトウェア・ディストリビューション (BSD) との互換性

ソケットはバークレー・ソフトウェア・ディストリビューション (BSD) のインターフェースです。アプリケーションが受け取る戻りコードやサポートされている関数で使用可能な引き数などのセマンティクスは、

BSD のセマンティクスです。ただし、いくつかの BSD のセマンティクスは OS/400 の実装では使用できないので、システムで実行するためには、典型的な BSD ソケット・アプリケーションを変更する必要がある場合もあります。

以下のリストは、OS/400 と BSD の違いを要約したものです。

/etc/hosts、/etc/services、/etc/networks、および /etc/protocols

これらのファイルの場合、OS/400 実装は、それぞれ同じ関数を実行する次のデータベース・ファイルを提供します。

QUSRSYS ファイル	**内容**
QATOCHOST	ホスト名とそれに対応する IP アドレスのリスト。
QATOCPN	ネットワークとそれに対応する IP アドレスのリスト。
QATOCPP	インターネットで使用されるプロトコルのリスト。
QATOCPS	サービスおよびサービスで使用する特定のポートとプロトコルのリスト。

/etc/resolv.conf

OS/400 実装では、この情報を iSeries ナビゲーターの TCP/IP プロパティー・ページを使用して構成する必要があります。TCP/IP プロパティー・ページにアクセスするには、以下のステップを実行します。

1. iSeries ナビゲーターで、「**iSeries/400 サーバー**」->「**ネットワーク**」->「**TCP/IP 構成**」と展開します。
2. 「**TCP/IP 構成**」を右クリックします。
3. 「**プロパティー**」をクリックします。

bind() BSD システムでは、クライアントは socket() を使用して AF_UNIX ソケットを作成し、 connect() を使用してサーバーに接続し、次いで bind() を使用して名前をソケットにバインドできます。OS/400 では、このシナリオをサポートしていません (bind() は失敗します)。

close()

OS/400 では、 close() に対してリンガー・タイマーをサポートしています (SNA 上の AF_INET ソケットを除く)。一部の BSD では、 close() に対してリンガー・タイマーをサポートしていません。

connect()

BSD システムでは、connect() が以前にアドレスに接続されていたソケットに対して発行され、コネクションレス型トランスポート・サービスを使用しており、無効なアドレスまたは無効なアドレス長が使用されている場合は、ソケットは切断されます。OS/400 では、このシナリオをサポートしていません (connect() は失敗し、ソケットは接続されたままです)。

connect() が発行されたコネクションレス型トランスポート・ソケットは、address_length パラメーターをゼロに設定して、別の connect() を発行することによって切断できます。

accept()、getsockname()、getpeername()、recvfrom()、および recvmsg()

AF_UNIX または AF_UNIX_CCSID アドレス・ファミリーを使用しており、ソケットがバインドされていない場合は、デフォルトの OS/400 実装ではゼロのアドレス長および指定されていないアドレス構造が戻されることがあります。OS/400 BSD 4.4/UNIX 98 および他の実装では、アドレス・ファミリーだけが指定されている小さいアドレス構造が戻されることがあります。

ioctl()

- BSD システムでは、タイプが SOCK_DGRAM のソケットで、FIONREAD 要求はデータ長とアドレス長を足したものを戻します。OS/400 実装では、FIONREAD はデータ長を戻すのみです。
- ほとんどの BSD で **ioctl()** を実行する場合に使用できるすべての要求が、OS/400 実装で **ioctl()** を実行する場合に使用できるわけではありません。

listen() BSD システムでは、バックログ・パラメーターをゼロよりも小さい値に設定して **listen()** を出すと、エラーになりません。さらに、BSD 実装では、バックログ・パラメーターが使用されなかったり、バックログ値の最終結果を見出すためにアルゴリズムが使用される場合があります。OS/400 実装では、バックログ値がゼロより小さい場合はエラーが戻されます。バックログを有効な値に設定すると、その値がバックログとして使用されます。ただし、バックログを {SOMAXCONN} より大きい値に設定すると、バックログはデフォルトで {SOMAXCONN} に設定した値になります。

アウト・オブ・バンド (OOB) データ
OS/400 実装では、SO_OOBINLINE を設定しておらず、OOB データがすでに受信されていて、その後ユーザーが SO_OOBINLINE をオン設定する場合は、OOB データは破棄されません。最初の OOB バイトは、通常データと見なされます。

socket() のプロトコル・パラメーター
セキュリティーを追加する手段として、 IPPROTO_TCP または IPPROTO_UDP のプロトコルを指定する SOCK_RAW ソケットを作成することはできません。

res_xlate() および res_close()
これらの関数は、OS/400 の resolver ルーチンに組み込まれています。 **res_xlate()** は、DNS パケットを EBCDIC から ASCII に、また ASCII から EBCDIC に変換します。 **res_close()** を使用すれば、RES_STAYOPEN オプションが設定された **res_send()** が使用していたソケットをクローズできます。また、_res 構造のリセットもします。

sendmsg() および recvmsg()

OS/400 で **sendmsg()** と **recvmsg()** を実行する場合には、{MSG_MAXIOVLEN} 入出力ベクトルまでが許可されます。BSD では、{MSG_MAXIOVLEN - 1} 入出力ベクトルが許可されます。

shutdown()
OS/400 実装では、ソケット記述子で現在ブロックされている出力関数は、**shutdown()** の後もブロックしたままです。BSD では、ブロック出力関数は、[EPIPE] errno 値を出して終了します。同様に BSD 実装では、入力操作がブロックされており、別のプロセスまたはスレッドから **shutdown()** が出されると、出力値ゼロで、入力操作のブロックを終了します。OS/400 実装では、出力値ゼロで後続のすべての入力関数を失敗させますが、ブロック入力関数は、データが受信されるか、待ち状態からデータを解放するために他の何らかの処置が取られるまで、ブロックを続けます。

信号 信号サポートに関連したいくつかの相違点を以下に示します。

- BSD は、出力命令で送信されたデータについて、肯定応答を受信するたびに、SIGIO 信号を出します。OS/400 ソケットは、アウトバウンド・データに関連した信号の生成を行いません。
- BSD での SIGPIPE 信号のデフォルトの動作は、処理を終了させることです。OS/400 の前のリリースとの下位互換性を維持するために、OS/400 では、SIGPIPE 信号の無視というデフォルトのアクションを使用します。

SO_REUSEADDR オプション

BSD システムの場合、AF_INET ファミリーおよびタイプ SOCK_DGRAM のソケットで **connect()** を実行すると、システムはソケットがバインドされるアドレスを別のアドレスに変更します。すなわち、**connect()** に指定されたアドレスに到達するために使用するインターフェースのアドレスに変更します。たとえば、タイプ SOCK_DGRAM のソケットをアドレス INADDR_ANY にバイン

ドし、それをアドレス a.b.c.d に接続した場合、システムは、現在バインドされているソケットを別のアドレス、すなわち、パケットをアドレス a.b.c.d に経路指定するために選択されたインターフェースの IP アドレスに変更します。さらに、ソケットがバインドされているこの IP アドレスが a.b.c.e だとすれば、アドレス a.b.c.e は INADDR_ANY の代わりに **getsockname()** に現れるので、他のソケットをアドレスが a.b.c.e である同一のポート番号にバインドするには、SO_REUSEADDR オプションを使用しなければなりません。

反対に、この例では、OS/400 はローカル・アドレスを INADDR_ANY から a.b.c.e に変更しません。 **getsockname()** は connect が実行された後も INADDR_ANY を戻し続けます。

SO_SNDBUF および SO_RCVBUF オプション

BSD システム上で SO_SNDBUF および SO_RCVBUF に設定された値は、OS/400 上で設定された値よりも高い制御レベルを提供します。OS/400 実装では、これらの値は通知値と見なされます。

UNIX 98 互換性

UNIX 98 は、開発者とベンダーの協会である Open Group によって作成されました。 UNIX オペレーティング・システムの名声を高めたインターネット関連の多くの機能を取り込みつつ、UNIX の不具合を改良しています。 V5R2 から、OS/400 ソケットを使って、UNIX 98 操作環境と互換性のあるソケット・アプリケーションを作成できるようになりました。現在、IBM は、大部分のソケット API について、2 つのバージョンをサポートしています。基本 OS/400 API は、BSD 4.3 の構造と構文を使用します。もう一方は、BSD 4.4 および UNIX 98 プログラミング・インターフェース仕様と互換性のある構文および構造を使用します。 _XOPEN_SOURCE マクロの値を 520 以上に定義すると、UNIX 98 互換のインターフェースを選択できます。

UNIX 98 互換アプリケーションのアドレス構造の相違点

_XOPEN_OPEN マクロを指定すれば、デフォルトの OS/400 実装で使用するのと同じアドレス・ファミリーを使って、UNIX 98 互換アプリケーションを作成できます。ただし、**sockaddr** アドレス構造に相違があります。以下の表に、BSD 4.3 の **sockaddr** アドレス構造と、UNIX 98 互換のアドレス構造の比較を示します。

表 17. BSD 4.3 と UNIX 98/BSD 4.4 のソケット・アドレス構造の比較

BSD 4.3 構造	**BSD 4.4/UNIX 98 互換の構造**
sockaddr アドレス構造	

`struct sockaddr {`
` u_short sa_family;`
` char sa_data[14];`
`};`
	`struct sockaddr {`
	` uint8_t sa_len;`
	` sa_family_t sa_family`
	` char sa_data[14];`
	`};`

sockaddr_in アドレス構造	

`struct sockaddr_in {`
`  short sin_family;`
`  u_short sin_port;`
`  struct in_addr sin_addr;`
`  char sin_zero[8];`
`};`
	`struct sockaddr_in {`
	`  uint8_t sin_len;`
	`  sa_family_t sin_family;`
	`  u_short sin_port;`
	`  struct in_addr sin_addr;`
	`  char sin_zero[8];`
	`};`

sockaddr_in6 アドレス構造	

表 17. BSD 4.3 と UNIX 98/BSD 4.4 のソケット・アドレス構造の比較 (続き)

struct sockaddr_in6 {
  sa_family_t sin6_family;
  in_port_t sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;
  uint32_t sin6_scope_id;
};
	struct sockaddr_in6 {
	  uint8_t sin6_len;
	  sa_family_t sin6_family;
	  in_port_t sin6_port;
	  uint32_t sin6_flowinfo;
	  struct in6_addr sin6_addr;
	  uint32_t sin6_scope_id;
	};

sockaddr_un アドレス構造	

struct sockaddr_un {
  short sun_family;
  char sun_path[126];
};
	struct sockaddr_un {
	  uint8_t sun_len;
	  sa_family_t sun_family;
	  char sun_path[126]
	};

API の相違点

アプリケーションを ILE ベースの言語で開発し、_XOPEN_SOURCE マクロを使ってコンパイルすると、一部のソケット API が内部名にマップされます。これらの内部名が提供する機能は、元の API と同じです。以下の表に、影響を受ける API をリストします。 C ベースの他の言語でソケット・アプリケーションを作成する場合、直接にこれらの API の内部名を使用できます。これらの API の両方のバージョンの使用上の注意と詳細について調べるには、元の API のリンクをたどってください。

表 18. API とそれに相当する UNIX 98 での名前

API 名	内部名
accept()	qso_accept98()
accept_and_recv()	qso_accept_and_recv98()
bind()	qso_bind98()
connect()	qso_connect98()
endhostent()	qso_endhostent98()
endnetent()	qso_endnetent98()
endprotoent()	qso_endprotoent98()
endservent()	qso_endservent98()
getaddrinfo()	qso_getaddrinfo98()
gethostbyaddr()	qso_gethostbyaddr98()
gethostbyaddr_r()	qso_gethostbyaddr_r98()
gethostname()	qso_gethostname98()
gethostname_r()	qso_gethostname_r98()
gethostbyname()	qso_gethostbyname98()
gethostent()	qso_gethostent98()
getnameinfo()	qso_getnameinfo98()
getnetbyaddr()	qso_getnetbyaddr98()
getnetbyname()	qso_getnetbyname98()
getnetent()	qso_getnetent98()
getpeername()	qso_getpeername98()
getprotobyname()	qso_getprotobyname98()

表 18. API とそれに相当する UNIX 98 での名前 (続き)

getprotobynumber()	qso_getprotobynumber98()
getprotoent()	qso_getprotoent98()
getsockname()	qso_getsockname98()
getsockopt()	qso_getsockopt98()
getservbyname()	qso_getservbyname98()
getservbyport()	qso_getservbyport98()
getservent()	qso_getservent98()
inet_addr()	qso_inet_addr98()
inet_lnaof()	qso_inet_lnaof98()
inet_makeaddr()	qso_inet_makeaddr98()
inet_netof()	qso_inet_netof98()
inet_network()	qso_inet_network98()
listen()	qso_listen98()
Rbind()	qso_Rbind98()
recv()	qso_recv98()
recvfrom98()	qso_recvfrom98()
recvmsg()	qso_recvmsg98()
send()	qso_send98()
sendmsg()	qso_sendmsg98()
sendto()	qso_sendto98()
sethostent()	qso_sethostent98()
setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setprotoent98()
setsockopt()	qso_setsockopt98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

プロセス間での記述子の受け渡し - sendmsg() および recvmsg()

ジョブ間でオープン記述子の受け渡しを行うことができれば、新しい方法でクライアント・アプリケーションとサーバー・アプリケーションを設計できるようになります。ジョブ間でオープン記述子の受け渡しを行うことによって、あるプロセス (通常はサーバー) が記述子を入手するために必要なすべてのこと (ファイルをオープンし、接続を確立し、**accept()** API が完了するのを待機する) を行えるようになり、別のプロセス (通常はワーカー) が記述子のオープン後にすべてのデータ転送操作を処理できるようになります。この設計は、サーバー・ジョブとワーカー・ジョブの論理を簡単にします。この設計によって、異なるタイプのワーカー・ジョブも簡単にサポートできるようになります。サーバーは、どのタイプのワーカーが記述子を受信するのかを簡単に判別できるようになります。

ソケットには、サーバー・ジョブ間で記述子を受け渡すことのできる以下の 3 つの API が用意されています。

- **spawn()**

注: **spawn()** はソケット API ではありません。これは OS/400 プロセスに関連した API の一部として提供されています。

- **givedescriptor()** および **takedescriptor()**
- **sendmsg()** および **recvmsg()**

spawn() API は、新しいサーバー・ジョブ (「子ジョブ」という) を開始し、特定の記述子をその子ジョブに割り当てます。その子ジョブがすでに活動状態である場合は、**givedescriptor()** および **takedescriptor()** API、または **sendmsg()** および **recvmsg()** API を使用する必要があります。

しかし、**sendmsg()** および **recvmsg()** API は、**spawn()** および **givedescriptor()** および **takedescriptor()** よりも以下の点で優れています。

可搬性

givedescriptor() および **takedescriptor()** API は、非標準で、iSeries だけで使用される API です。iSeries と UNIX の間でのアプリケーションの移植性が問題となっている場合は、**sendmsg()** および **recvmsg()** API を使用することができます。

制御情報の通信

ほとんどの場合、ワーカー・ジョブは記述子を受け取るときに、以下のような追加情報を必要とします。

- 記述子のタイプ
- ワーカー・ジョブがすべきこと

sendmsg() および **recvmsg()** API を使用すれば、記述子だけでなく、制御情報などのデータも転送できます。 **givedescriptor()** および **takedescriptor()** では、それが行えません。

パフォーマンス

sendmsg() および **recvmsg()** API を使用するアプリケーションの方が、**givedescriptor()** および **takedescriptor()** API を使用するアプリケーションよりも、以下の 3 つの分野で優れた結果を残す傾向にあります。

- 所要時間
- CPU の使用率
- スケーラビリティー

アプリケーションのパフォーマンスがどの程度向上するかは、アプリケーションが記述子を受け渡しする範囲によって異なります。

ワーカー・ジョブのプール

ワーカー・ジョブのプールを設定しておけば、サーバーはそこに記述子を渡すことができ、プール内の 1 つのジョブだけがその記述子を受け取ることができます。このことを行うためには、**sendmsg()** および **recvmsg()** API を使用して、すべてのワーカー・ジョブが共用記述子を使うようにします。サーバーが **sendmsg()** を呼び出すと、それらのワーカー・ジョブのうち 1 つだけが記述子を受け取ります。

不明ワーカー・ジョブ ID

givedescriptor() API では、サーバー・ジョブがワーカー・ジョブのジョブ ID を知っている必要があります。一般に、ワーカー・ジョブはジョブ ID を入手して、それをデータ待ち行列によってサーバー・ジョブに転送します。 **sendmsg()** および **recvmsg()** では、このデータ待ち行列を作成および管理するための余分のオーバーヘッドが不要です。

最適なサーバー設計

givedescriptor() および **takedescriptor()** を使用してサーバーを設計した場合、通常はデータ待ち行

列を使用してワーカー・ジョブからサーバーへとジョブ ID を転送します。転送後、サーバーは **socket()**、**bind()**、**listen()**、および **accept()** を実行します。**accept()** API が完了すると、サーバーは次に使用可能なジョブ ID をデータ待ち行列からプルオフします。それから、インバウンド接続をそのワーカー・ジョブに渡します。問題が発生するのは、一度に多くの着信接続要求が出され、使用可能なワーカー・ジョブが不足する場合です。ワーカー・ジョブ ID を含んでいるデータ待ち行列が空になると、サーバーはワーカー・ジョブが使用可能になるのを待機することを止めるか、あるいは追加のワーカー・ジョブを作成します。ほとんどの環境では、追加の着信要求が listen バックログを満たす可能性があるため、これら 2 つのどちらも行うべきではありません。

sendmsg() および **recvmsg()** API を使用して記述子を渡すサーバーは、活動が活発に行われている間はブロックされない状態のままです。サーバーは、どのワーカー・ジョブがそれぞれの着信接続を扱うのかを知る必要がないためです。サーバーが **sendmsg()** を呼び出すと、着信接続の記述子と制御データが AF_UNIX ソケット用の内部待ち行列に書き込まれます。ワーカー・ジョブは、使用可能になると **recvmsg()** を呼び出して、待ち行列に最初に書き込まれた記述子と制御データを受け取ります。

非活動ワーカー・ジョブ

givedescriptor() API の場合、ワーカー・ジョブが活動状態でなければなりません。 **sendmsg()** API の場合は、活動状態でなくても構いません。 **sendmsg()** を呼び出すジョブは、ワーカー・ジョブについての情報を必要としません。 **sendmsg()** API を使用するために必要なことは、AF_UNIX ソケット接続を設定することのみです。

sendmsg() API を使用して、存在しないジョブに記述子をどのように渡すことができるかを示す例を、以下に紹介します。

サーバーは、**socketpair()** API を使用して AF_UNIX ソケットの組を作成し、**sendmsg()** API を使用して **socketpair()** で作成した AF_UNIX ソケットの組の一方を介して記述子を送信し、次いで **spawn()** を呼び出してソケットの組のもう一方を継承する子ジョブを作成することができます。子ジョブは **recvmsg()** を呼び出して、サーバーが渡した記述子を受信します。サーバーが **sendmsg()** を呼び出したとき、子ジョブは活動状態ではありませんでした。

一度に複数の記述子を渡す

givedescriptor() および **takedescriptor()** API は、一度に 1 つの記述子しか渡すことができません。 **sendmsg()** および **recvmsg()** API を使用すれば、記述子の配列を渡すことができます。

sendmsg() および **recvmsg()** API を使用するサンプル・プログラムについては、『例: プロセス間での記述子の受け渡し』を参照してください。

ソケットのシナリオ: IPv4 クライアントと IPv6 クライアントを受け入れるアプリケーションの作成

状況

iSeries 用ソケット・アプリケーション専門のアプリケーション開発企業で働く、ソケット・プログラマーであると想像してみてください。競争相手の先を行くため、AF_INET6 アドレス・ファミリーを使った、IPv4 と IPv6 の両方の接続を受け入れるアプリケーションのスイートを開発することにします。作成したいのは、IPv4 ノードと IPv6 ノードの両方からの要求を処理するアプリケーションです。 OS/400 が、AF_INET アドレス・ファミリー・ソケットとの相互運用性を備えた、AF_INET6 アドレス・ファミリーをサポートすることは分かっています。そのために IPv4 マップ IPv6 アドレス形式を使用するということも分かっています。 IPv6 アプリケーションと IPv4 アプリケーションの間の相互運用性の処理の詳細については、『IPv6 アプリケーションと IPv4 アプリケーションの互換性』を参照してください。

シナリオの目的

このシナリオの目的は次のとおりです。

1. IPv6 クライアントと IPv4 クライアントからの要求を受け入れて処理する、サーバー・アプリケーションを作成する。
2. IPv4 サーバー・アプリケーションまたは IPv6 サーバー・アプリケーションにデータを要求する、クライアント・アプリケーションを作成する。

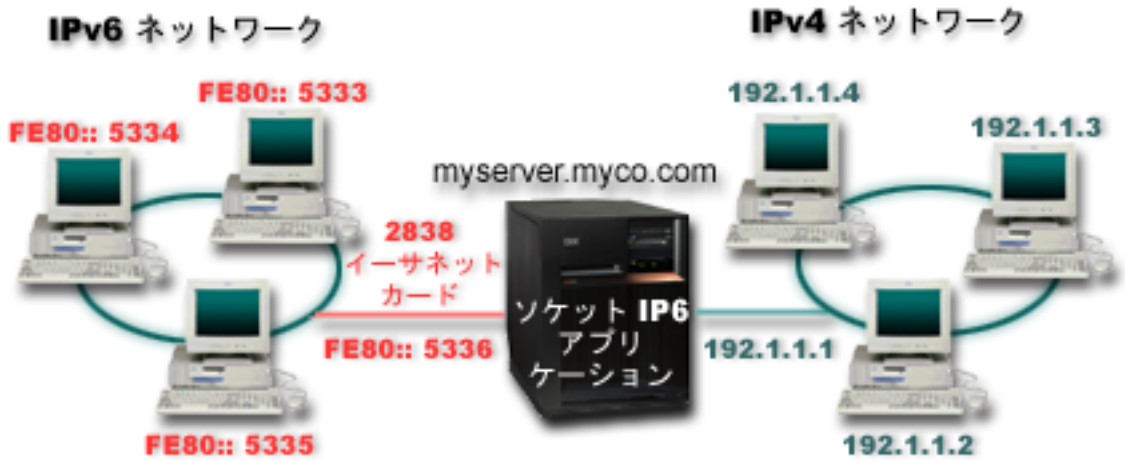
前提条件のステップ

上記の目的を満たすアプリケーションを開発するには、あらかじめ以下のタスクを実行しておく必要があります。

1. QSYSINC ライブラリーをインストールする。このライブラリーは、ソケット・アプリケーションのコンパイル時に必要なヘッダー・ファイルを提供します。
2. C Compiler ライセンス・プログラム (5722-CX2) をインストールする。
3. 2838 イーサネット・カードをインストールして、構成する。イーサネット・オプションについては、Information Center の『イーサネット』のトピックを参照してください。
4. TCP/IP および IPv6 ネットワークをセットアップする。

シナリオの詳細

以下の図は、アプリケーションを作成する対象となる IPv6 ネットワークを示したものです。作成するアプリケーションは、IPv6 クライアントと IPv4 クライアントからの要求を処理します。プログラムは iSeries 上で稼働し、これらのクライアントからの要求を listen し、処理します。このネットワークは 2 つの別個のドメインで構成されます。一方には IPv4 クライアントのみが含まれ、もう一方のリモート・ネットワークには IPv6 クライアントのみが含まれます。 iSeries のドメイン・ネームは myserver.myco.com です。サーバー・アプリケーションは、AF_INET6 アドレス・ファミリーを使用してこれらの着信要求を処理します。 **bind()** 関数呼び出しには、in6addr_any を指定します。



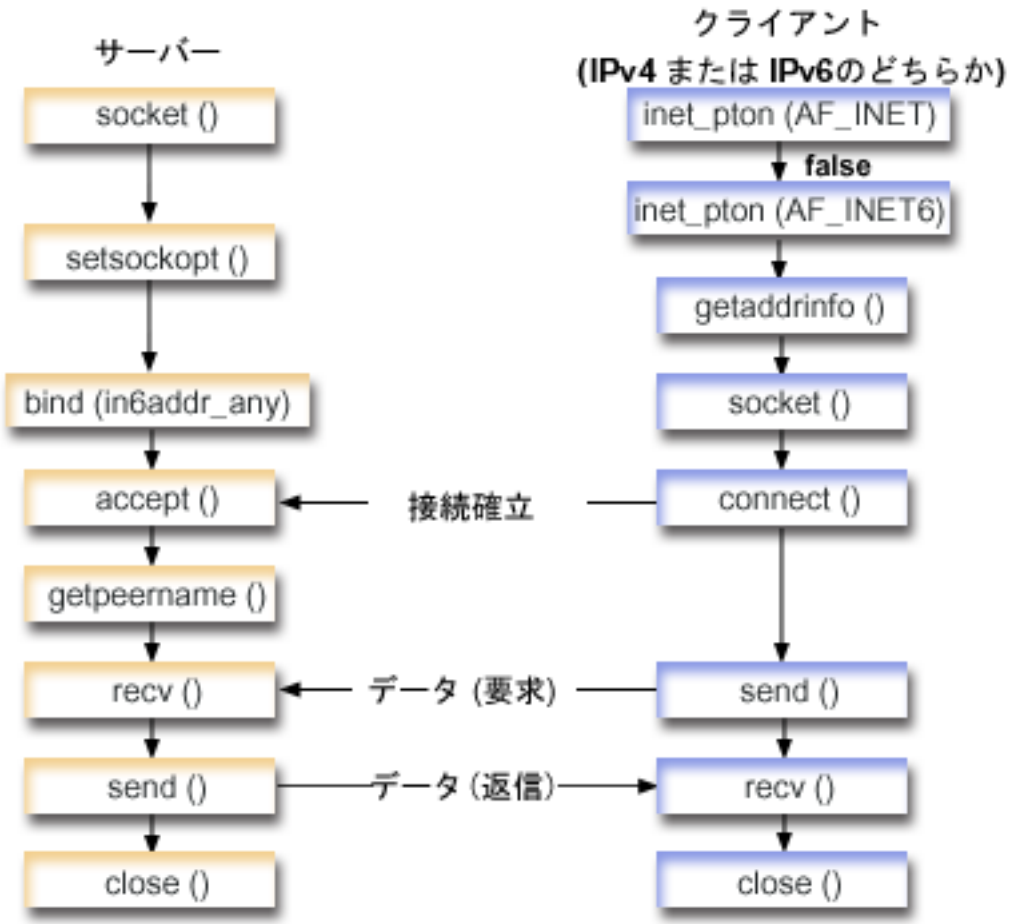
このシナリオで使用される以下のプログラム例を参照してください。

- 例: IPv6 クライアントと IPv4 クライアントの両方から接続を受け入れる
- 例: IPv4 または IPv6 クライアント

例: IPv6 クライアントと IPv4 クライアントの両方から接続を受け入れる

このサンプル・プログラムを使用して、IPv4 (AF_INET アドレス・ファミリーを使用するソケット・アプリケーション) および IPv6 (AF_INET6 アドレス・ファミリーを使用するアプリケーション) の両方からの要求を受け入れるサーバー/クライアント・モデルを作成します。現在のところ、ソケット・アプリケーションは、TCP および UDP プロトコルを考慮に入れた AF_INET アドレス・ファミリーのみを使用しているかもしれません。しかし、これは、IPv6 アドレスの使用の増加に伴って、変わっていく可能性があります。このサンプル・プログラムを使用して、両方のアドレス・ファミリーに対応するアプリケーションを作成することができます。

以下の図は、このプログラム例がどのように機能するかを示しています。



ソケットのイベントのフロー: IPv4 クライアントと IPv6 クライアントの両方からの要求を受け入れるサーバー・アプリケーション

このフローは、IPv4 クライアントと IPv6 クライアントの両方の要求を受け入れるソケット・アプリケーションに、どのような関数呼び出しが含まれており、それぞれが何を行うかを説明したものです。

1. **socket()** API が、端点を作成するソケット記述子を指定します。さらに、IPv6 をサポートする AF_INET6 アドレス・ファミリーも指定します。このソケットには、TCP トランスポート (SOCK_STREAM) が使用されます。

2. **setsockopt()** 関数により、必要な待ち時間が満了する前にサーバーが再始動した場合に、アプリケーションはローカル・アドレスを再利用できるようになります。
3. **bind()** 関数が、ソケットの固有名を指定します。この例では、プログラマーはアドレスを in6addr_any に設定します。これにより、ポート 3005 を指定するあらゆる IPv4 クライアントまたは IPv6 クライアントが接続を確立できるようになります。 (つまり、IPv4 ポート空間と IPv6 ポート空間の両方にバインドされます)

   注: サーバーが IPv6 クライアントのみを処理できればよい場合は、IPV6_ONLY ソケット・オプションを使用できます。
4. **listen()** 関数により、サーバーが着信クライアント接続を受け入れられるようになります。この例では、プログラマーはバックログを 10 に設定します。これは、待ち行列に入れられた接続要求が 10 個に達すると、システムが着信要求を拒否するようになるということです。
5. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、IPv4 クライアントまたは IPv6 クライアントからの着信接続を待機して、無期限にブロックします。
6. **getpeername()** 関数が、クライアントのアドレスをアプリケーションに戻します。 IPv4 クライアントの場合、アドレスは IPv4 マップ IPv6 アドレスとして表示されます。
7. **recv()** 関数が、クライアント・アプリケーションから 250 バイトのデータを受信します。この例の場合、プログラマーは、クライアントが 250 バイトのデータを送信してくることが分かっているものとします。そのため、プログラマーは SO_RCVLOWAT ソケット・オプションを使用し、250 バイトのデータがすべて到着するまで **recv()** がウェイクアップしないように指定します。
8. **send()** 関数が、クライアントにデータを送り返します。
9. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

ソケットのイベントのフロー: IPv4 クライアントまたは IPv6 クライアントからの要求

注: このクライアントの例は、IPv4 ノードまたは IPv6 ノードの要求を受け入れる、他のサーバー・アプリケーション設計と一緒に使用できます。『例: コネクション型設計』で説明されているような他のサーバー設計を、このクライアントの例と組み合わせて使用することができます。

1. **inet_pton()** 呼び出しが、テキスト形式のアドレスをバイナリー形式に変換します。この例では、以下の 2 つの呼び出しを発行します。最初の呼び出しは、サーバーが有効な AF_INET アドレスかどうかを判別します。 2 番目の **inet_pton()** 呼び出しは、サーバーが AF_INET6 アドレスを持っているかを判別します。アドレスが数値の場合、**getaddrinfo()** がネーム・レゾリューションをしないようにします。さもないと、**getaddrinfo()** 呼び出しが発行されるときに、提供されたホスト名を解決することが必要になります。
2. **getaddrinfo()** 呼び出しが、後続の **socket()** 呼び出しと **connect()** 呼び出しに必要なアドレス情報を検索します。
3. **socket()** 関数が、端点を表すソケット記述子を戻します。さらにステートメントは、**getaddrinfo()** から戻される情報を使用して、アドレス・ファミリー、ソケット・タイプ、およびプロトコルも識別します。
4. サーバーが IPv4 か IPv6 であるかにかかわらず、**connect()** 関数がサーバーへの接続を確立します。
5. **send()** 関数が、サーバーにデータ要求を送信します。
6. **recv()** 関数が、サーバー・アプリケーションからデータを受信します。
7. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

以下のサンプル・コードは、このシナリオのサーバー・アプリケーションです。クライアント・アプリケーションについては、例: IPv4 または IPv6 クライアントを参照してください。このコードの使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /*********************************************************************/
   /* Variable and structure definitions.                               */
   /*********************************************************************/
   int sd=-1, sdconn=-1;
   int rc, on=1, rcdsize=BUFFER_LENGTH;
   char buffer[BUFFER_LENGTH];
   struct sockaddr_in6 serveraddr, clientaddr;
   int addrlen=sizeof(clientaddr);
   char str[INET6_ADDRSTRLEN];

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The */
   /* close() of each of the socket descriptors is only done once at the */
   /* very end of the program.                                          */
   /*********************************************************************/
   do
   {

      /*********************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  Get a socket for address family AF_INET6 to        */
      /* prepare to accept incoming connections on.                        */
      /*********************************************************************/
      if ((sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
      {
         perror("socket() failed");
         break;
      }

      /*********************************************************************/
      /* The setsockopt() function is used to allow the local address to  */
      /* be reused when the server is restarted before the required wait  */
      /* time expires.                                                     */
      /*********************************************************************/
      if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                     (char *)&on,sizeof(on)) < 0)
      {
         perror("setsockopt(SO_REUSEADDR) failed");
         break;
      }

      /*********************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.  In this example, the user sets the  */
      /* address to in6addr_any, which (by default) allows connections to */
```

```
/* be established from any IPv4 or IPv6 client that specifies port  */
/* 3005. (i.e. the bind is done to both the IPv4 and IPv6 TCP/IP    */
/* stacks).  This behavior can be modified using the IPPROTO_IPV6   */
/* level socket option IPV6_V6ONLY if desired.                      */
/********************************************************************/
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin6_family = AF_INET6;
serveraddr.sin6_port   = htons(SERVER_PORT);
/********************************************************************/
/* Note: applications use in6addr_any similarly to the way they use */
/* INADDR_ANY in IPv4.  A symbolic constant IN6ADDR_ANY_INIT also   */
/* exists but can only be used to initialize an in6_addr structure  */
/* at declaration time (not during an assignment).                  */
/********************************************************************/
serveraddr.sin6_addr   = in6addr_any;
/********************************************************************/
/* Note: the remaining fields in the sockaddr_in6 are currently not */
/* supported and should be set to 0 to ensure upward compatibility. */
/********************************************************************/

if (bind(sd,
         (struct sockaddr *)&serveraddr,
         sizeof(serveraddr)) < 0)
{
   perror("bind() failed");
   break;
}

/********************************************************************/
/* The listen() function allows the server to accept incoming       */
/* client connections.  In this example, the backlog is set to 10.  */
/* This means that the system will queue 10 incoming connection     */
/* requests before the system starts rejecting the incoming         */
/* requests.                                                        */
/********************************************************************/
if (listen(sd, 10) < 0)
{
   perror("listen() failed");
   break;
}

printf("Ready for client connect().\n");

/********************************************************************/
/* The server uses the accept() function to accept an incoming      */
/* connection request.  The accept() call will block indefinitely   */
/* waiting for the incoming connection to arrive from an IPv4 or    */
/* IPv6 client.                                                     */
/********************************************************************/
if ((sdconn = accept(sd, NULL, NULL)) < 0)
{
   perror("accept() failed");
   break;
}
else
{
   /*****************************************************************/
   /* Display the client address.  Note that if the client is       */
   /* an IPv4 client, the address will be shown as an IPv4 Mapped   */
   /* IPv6 address.                                                 */
   /*****************************************************************/
   getpeername(sdconn, (struct sockaddr *)&clientaddr, &addrlen);
   if(inet_ntop(AF_INET6, &clientaddr.sin6_addr, str, sizeof(str))) {
      printf("Client address is %s\n", str);
      printf("Client port is %d\n", ntohs(clientaddr.sin6_port));
   }
}
```

```
      /*******************************************************************/
      /* In this example we know that the client will send 250 bytes of  */
      /* data over.  Knowing this, we can use the SO_RCVLOWAT socket     */
      /* option and specify that we don't want our recv() to wake up     */
      /* until all 250 bytes of data have arrived.                       */
      /*******************************************************************/
      if (setsockopt(sdconn, SOL_SOCKET, SO_RCVLOWAT,
                     (char *)&rcdsize,sizeof(rcdsize)) < 0)
      {
         perror("setsockopt(SO_RCVLOWAT) failed");
         break;
      }

      /*******************************************************************/
      /* Receive that 250 bytes of data from the client                  */
      /*******************************************************************/
      rc = recv(sdconn, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("recv() failed");
         break;
      }

      printf("%d bytes of data were received\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("The client closed the connection before all of the\n");
         printf("data was sent\n");
         break;
      }

      /*******************************************************************/
      /* Echo the data back to the client                                */
      /*******************************************************************/
      rc = send(sdconn, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("send() failed");
         break;
      }

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*******************************************************************/
   /* Close down any open socket descriptors                          */
   /*******************************************************************/
   if (sd != -1)
      close(sd);
   if (sdconn != -1)
      close(sdconn);
}
```

例: IPv4 または IPv6 クライアント

このサンプル・プログラムは、IPv4 クライアントまたは IPv6 クライアントのどちらかからの要求を受け入れるサーバー・アプリケーションと一緒に使用できます。

```
/*********************************************************************/
/* This is an IPv4 or IPv6 client.                                   */
/*********************************************************************/
```

```
/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
   /*********************************************************************/
   /* Variable and structure definitions.                               */
   /*********************************************************************/
   int    sd=-1, rc, bytesReceived=0;
   char   buffer[BUFFER_LENGTH];
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   char   servport[] = "3005";
   struct in6_addr serveraddr;
   struct addrinfo hints, *res=NULL;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program along with the free of the list of addresses.        */
 /*********************************************************************/
   do
   {
      /*********************************************************************/
      /* If an argument was passed in, use this as the server, otherwise  */
      /* use the #define that is located at the top of this program.      */
      /*********************************************************************/
      if (argc > 1)
         strcpy(server, argv[1]);
      else
         strcpy(server, SERVER_NAME);

      memset(&hints, 0x00, sizeof(hints));
      hints.ai_flags    = AI_NUMERICSERV;
      hints.ai_family   = AF_UNSPEC;
      hints.ai_socktype = SOCK_STREAM;
      /*********************************************************************/
      /* Check if we were provided the address of the server using        */
      /* inet_pton() to convert the text form of the address to binary    */
      /* form. If it is numeric then we want to prevent getaddrinfo()     */
      /* from doing any name resolution.                                   */
      /*********************************************************************/
      rc = inet_pton(AF_INET, server, &serveraddr);
      if (rc == 1)    /* valid IPv4 text address? */
      {
         hints.ai_family = AF_INET;
         hints.ai_flags |= AI_NUMERICHOST;
      }
      else
```

```
{
   rc = inet_pton(AF_INET6, server, &serveraddr);
   if (rc == 1) /* valid IPv6 text address? */
   {

      hints.ai_family = AF_INET6;
      hints.ai_flags |= AI_NUMERICHOST;
   }
}
/*********************************************************************/
/* Get the address information for the server using getaddrinfo().  */
/*********************************************************************/
rc = getaddrinfo(server, servport, &hints, &res);
if (rc != 0)
{
   printf("Host not found --> %s\n", gai_strerror(rc));
   if (rc == EAI_SYSTEM)
      perror("getaddrinfo() failed");
   break;
}

/*********************************************************************/
/* The socket() function returns a socket descriptor representing   */
/* an endpoint.  The statement also identifies the address family,  */
/* socket type, and protocol using the information returned from    */
/* getaddrinfo().                                                   */
/*********************************************************************/
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sd < 0)
{
   perror("socket() failed");
   break;
}
/*********************************************************************/
/* Use the connect() function to establish a connection to the      */
/* server.                                                          */
/*********************************************************************/
rc = connect(sd, res->ai_addr, res->ai_addrlen);
if (rc < 0)
{
   /*****************************************************************/
   /* Note: the res is a linked list of addresses found for server. */
   /* If the connect() fails to the first one, subsequent addresses */
   /* (if any) in the list could be tried if desired.               */
   /*****************************************************************/
   perror("connect() failed");
   break;
}

/*********************************************************************/
/* Send 250 bytes of a's to the server                              */
/*********************************************************************/
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("send() failed");
   break;
}

/*********************************************************************/
/* In this example we know that the server is going to respond with */
/* the same 250 bytes that we just sent.  Since we know that 250    */
/* bytes are going to be sent back to us, we could use the          */
/* SO_RCVLOWAT socket option and then issue a single recv() and     */
/* retrieve all of the data.                                        */
/*                                                                  */
```

```
      /* The use of SO_RCVLOWAT is already illustrated in the server      */
      /* side of this example, so we will do something different here.    */
      /* The 250 bytes of the data may arrive in separate packets,        */
      /* therefore we will issue recv() over and over again until all     */
      /* 250 bytes have arrived.                                          */
      /********************************************************************/
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("recv() failed");
            break;
         }
         else if (rc == 0)
         {
            printf("The server closed the connection\n");
            break;
         }

         /*****************************************************************/
         /* Increment the number of bytes that have been received so far  */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
   /*********************************************************************/
   /* Free any results returned from getaddrinfo                        */
   /*********************************************************************/
   if (res != NULL)
      freeaddrinfo(res);
}
```

ソケット・アプリケーション設計の推奨事項

ソケット・アプリケーションを処理する前に、機能要件、目標、およびソケット・アプリケーションの必要を査定してください。アプリケーションのパフォーマンス要件およびシステム・リソースの影響についても考慮してください。以下の推奨事項のリストは、ソケット・アプリケーションのこれらの問題のいくつかに取り組み、ソケットのより良い使用方法およびソケット・アプリケーションのより良い設計方法に注目するために役立ちます。

表 19. ソケット・アプリケーション設計

推奨事項	理由	最適な使用方法
非同期入出力を使用する。	スレッド化されたサーバー・モデルで使用される非同期入出力を、より一般的な select() モデルよりも推奨します。非同期入出力を使用する利点の詳細については、『非同期入出力』を参照してください。非同期入出力 API を使用するサンプル・プログラムについては、『例: 非同期入出力 API の使用』を参照してください。	多くの並行クライアントを処理するソケット・サーバー・アプリケーション。

表 19. ソケット・アプリケーション設計 (続き)

非同期入出力を使用するときは、プロセス内のスレッドの数を、処理するクライアントの数に最適の数になるように調整する。	定義するスレッドの数が少なすぎると、一部のクライアントは処理される前にタイムアウトになることもあります。定義するスレッドの数が多すぎると、一部のシステム・リソースは効率的に使用されないこともあります。 **注:** スレッドの数は、少なすぎるよりは多すぎる方が利点があります。	非同期入出力を使用するソケット・アプリケーション。
非同期入出力のすべての開始操作で postflag を使用しないようにソケット・アプリケーションを設計する。	操作が同期してすでに完了している場合は、完了ポートへ移行するというパフォーマンス上のオーバーヘッドを回避できます。	非同期入出力を使用するソケット・アプリケーション。
send() および **recv()** を、**read()** および **write()** に優先して使用する。	**send()** および **recv()** API のパフォーマンスと保守容易性は、**read()** および **write()** よりもいくらか改善されています。	ファイル記述子ではなくソケット記述子を使用することを認識しているすべてのソケット・プログラム。
すべてのデータが到着するまで受信オペレーションがループするのを避けるために、受信最低水準 (SO_RCVLOWAT) ソケット・オプションを使用する。	アプリケーションが、ブロック受信操作を完了する前に、ソケットで最少量のデータが受信されるまで待機するようになります。	データを受信するすべてのソケット・アプリケーション。
すべてのデータが到着するまで受信オペレーションがループするのを避けるために、MSG_WAITALL フラグを使用する。	アプリケーションが、受信操作のために提供されるバッファー全体が受信されるのを待ってから、ブロック受信操作を完了するようになります。	データを受信し、着信するデータの量が事前に分かっているすべてのソケット・アプリケーション。
sendmsg() および **recvmsg()** を、**givedescriptor()** および **takedescriptor()** に優先して使用する。	利点については、『プロセス間での記述子の受け渡し - sendmsg() および recvmsg()』を参照してください。 **sendmsg()** および **recvmsg()** を使用するサンプル・プログラムについては、『例: プロセス間での記述子の受け渡し』を参照してください。	プロセス間でソケットまたはファイル記述子を受け渡しするすべてのソケット・アプリケーション。
select() を使用するときは、読み取り、書き込み、または例外セットに多くの記述子を入れないようにする。 **注: select()** 処理で多くの記述子が使用されている場合は、上記の非同期入出力の推奨事項を参照してください。	読み取り、書き込み、または例外セットに多くの記述子がある場合は、**select()** が呼び出されるたびにかなりの冗長作業が発生します。 **select()** が完了しても、実際のソケット関数はまだ完了していないはずです。つまり、読み取りまたは書き込みまたは受け入れが、まだ実行中であるはずです。非同期入出力 API は、ソケットで何かが起きたという通知を、実際の入出力操作と結び付けます。	**select()** 用に多くの (> 50) 記述子が活動状態になっているアプリケーション。

表 19. ソケット・アプリケーション設計 (続き)

select() を再発行するたびに読み取り、書き込み、および例外セットを再作成することを避けるために、**select()** を使用してそれらのセットのコピーを保管する。	これによって、**select()** を発行しようとするたびに、読み取り、書き込み、または例外セットを再作成するオーバーヘッドを減らすことができます。**select()** を使用するサンプル・プログラムについては、『例: 非ブロッキング入出力および select()』を参照してください。	読み取り、書き込み、または例外処理のために使用可能になっているソケット記述子が多くある状態で **select()** を使用しているすべてのアプリケーション。
select() をタイマーとして使用しない。代わりに **sleep()** を使用する。**注:** **sleep()** タイマーの細分度が十分でない場合は、**select()** をタイマーとして使用しなければならない場合があります。この場合、最大記述子を 0、読み取り、書き込みおよび例外セットを NULL に設定してください。	タイマー応答が向上し、システム・オーバーヘッドが減ります。	**select()** をタイマーとしてだけ使用しているすべてのソケット・アプリケーション。
ソケット・アプリケーションで、DosSetRelMaxFH() を使用してプロセスごとに許可されているファイルおよびソケット記述子の最大数を増やし、この同じアプリケーションで **select()** を使用している場合は、新しい最大値が、**select()** 処理に使用される読み取り、書き込みおよび例外セットのサイズに与える影響に注意する。	読み取り、書き込み、または例外セットの範囲 (FD_SETSIZE によって指定される) の外側の記述子を割り振る場合は、ストレージを上書きしたり破壊したりする可能性があります。設定するサイズは、プロセスに設定される記述子の最大数および **select()** API で指定される記述子の最大値がどのようなものであっても、少なくともそれを処理できる大きさのものにしてください。	DosSetRelMaxFH() および **select()** を使用するすべてのアプリケーションまたはプロセス。
読み取りまたは書き込みセットのすべてのソケット記述子を非ブロッキングに設定する。読み取りまたは書き込みのために記述子が使用可能になったら、EWOULDBLOCK が戻されるまで、すべてのデータをループおよび消費または送信する。**select()** を使用するサンプル・プログラムについては、『例: 非ブロッキング入出力および select()』を参照してください。	これによって、記述子でデータがまだ処理可能または読み取り可能になっているときに、**select()** 呼び出しの数を最小限にできます。	**select()** を使用しているすべてのソケット・アプリケーション。
select() 処理で使用する必要のあるセットのみを指定してください。	ほとんどのアプリケーションでは、例外セットまたは書き込みセットを指定する必要はありません。	**select()** を使用しているすべてのソケット・アプリケーション。

表 19. ソケット・アプリケーション設計 (続き)

SSL API ではなく GSKit API を使用する。	グローバル・セキュア・ツールキット (GSKit) と OS/400 SSL_ API のどちらを使っても、AF_INET または AF_INET6 による、セキュアな SOCK_STREAM ソケット・アプリケーションを開発できます。しかし、GSKit API は複数の IBM@server プラットフォームでサポートされているため、セキュア・アプリケーションにはこちらの API のセットを使うほうが好ましいと言えます。 SSL_ API は、OS/400 システム固有のものに過ぎません。	SSL/TLS 処理のために使用可能にする必要のあるすべてのソケット・アプリケーション。
信号を使用しないようにする。	信号のパフォーマンス上のオーバーヘッドは (iSeries のみでなくすべてのプラットフォームで)、影響が大きくなります。非同期入出力または **select()** API を使用するようにソケット・アプリケーションを設計することを推奨します。	ソケット・アプリケーションで信号を使用することを考慮しているすべてのプログラマー。
可能な場合には、**inet_ntop()**、**inet_pton()**、**getaddrinfo()**、**getnameinfo()** のような、プロトコルから独立したルーチンを使用する。	IPv6 をサポートする準備がまだできていなくても、これらの API を (**inet_ntoa()**、**inet_addr()**、**gethostbyname()**、および **gethostbyaddr()** の代わりに) 使用する場合は、将来のマイグレーションが容易になるよう準備していることになります。	ネットワーク・ルーチンを使用する、あらゆる AF_INET アプリケーションまたは AF_INET6 アプリケーション。
sockaddr_storage を使用して、あらゆるアドレス・ファミリーのアドレスのためのストレージを宣言する。	複数のアドレス・ファミリーとプラットフォームをまたいで移植可能なコードを作成するのが容易になります。最大のアドレス・ファミリーでも保持できるのみの大きさのストレージが宣言され、境界合わせも正しく行われます。	アドレスを保管するすべてのソケット・アプリケーション。

例: ソケット・アプリケーション設計

以下の例では、ソケットのさらに高度な概念を例示する多くのサンプル・プログラムを示します。これらのサンプル・プログラムを使用して、同様のタスクを実行する独自のアプリケーションを作成できます。こうした例に付随して、個々のアプリケーションのイベントのフローを示す、図と呼び出しの一覧を示します。Xsocket ツールを対話式で使用して、これらのプログラムの中の API の一部を使用したり、自分の環境に合わせて変更を加えたりすることができます。

- 例: コネクション型設計
- 例: セキュア接続の確立
- 例: gethostbyaddr_r() を使用したスレッド・セーフ・ネットワーク・ルーチン
- 例: 非ブロッキング入出力および select()

- 例: ブロック化ソケット API での信号の使用
- 例: AF_INET アドレス・ファミリーによってマルチキャスティングを使用する
- 例: DNS の更新および照会
- 例: send_file() および accept_and_recv() API を使用したファイル・データの転送

例: コネクション型設計

iSeries でコネクション型ソケット・サーバーを設計する方法はいくつかあります。以下のプログラム例を使用して、独自のコネクション型設計を作成することができます。別のソケット・サーバー設計も可能ですが、以下の例に示す設計が最も一般的です。

反復サーバー

反復サーバーの例では、クライアント・ジョブとの間で生じるすべての着信接続とデータ・フローを、単一のサーバー・ジョブが処理します。 **accept()** API が完了すると、サーバーがトランザクション全体を処理します。このサーバーは開発は最も容易ですが、問題がいくつかあります。サーバーは指定されたクライアントからの要求を処理しますが、別のクライアントもサーバーに接続しようとする可能性があります。こうした要求で **listen()** のバックログがいっぱいになって、結局、一部の要求が拒否されることになります。

他のすべての例は並行サーバー設計です。これらの設計では、システムは複数のジョブとスレッドを使用して着信接続要求を処理します。並行サーバーの場合、このサーバーに同時に接続する、複数のクライアントが存在するのが普通です。

ネットワークに複数の並行クライアントがある場合は、非同期入出力ソケット API を使用することをお勧めします。これらの API は、複数の並行クライアントのあるネットワークで最高のパフォーマンスを発揮します。『非同期入出力』では、これらの API が何をどのように実行するかを説明します。これらの API を使用するプログラム例については、『例: 非同期入出力 API の使用』を参照してください。

spawn() サーバーおよび spawn() ワーカー

spawn() サーバーと spawn() ワーカーの例では、spawn() API を使用して、個々の着信要求のために新しいジョブを 1 つずつ作成します。 spawn() が完了したら、サーバーは **accept()** API の上で、次の着信接続を受信するまで待機します。

このサーバー設計の唯一の問題点は、接続を受信するたびに新しいジョブを作成することに伴う、パフォーマンス上のオーバーヘッドです。事前開始ジョブを使用する場合は、spawn() サーバーの例のパフォーマンス上のオーバーヘッドを回避できます。つまり、接続を受信してから新しいジョブを作成するのではなく、事前にアクティブになっているジョブに着信接続を渡します。このトピックの他のすべての例では、事前開始ジョブを使用しています。

sendmsg() サーバーおよび recvmsg() ワーカー

sendmsg() サーバーと recvmsg() ワーカーの例では、着信接続をワーカー (クライアント) ジョブに渡します。最初のサーバー・ジョブが開始するとき、サーバーはすべてのワーカー・ジョブを事前に開始しておきます。

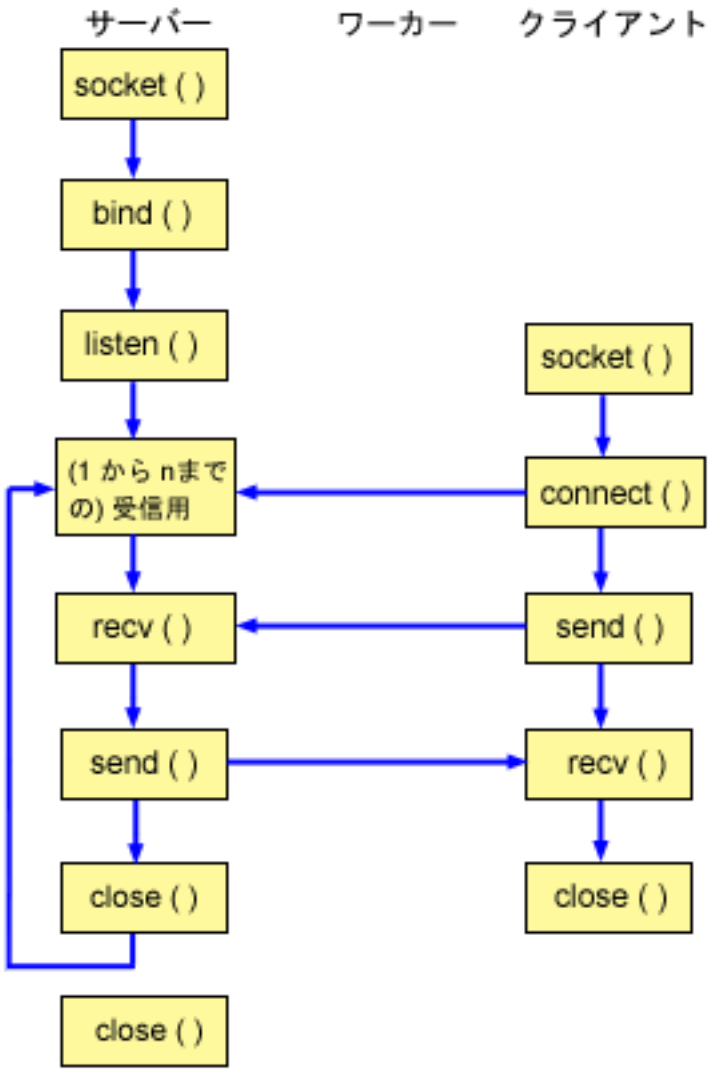
複数の accept() サーバーと複数の accept() ワーカー

直前の例では、サーバーが着信接続要求を受信するまで、ワーカー・ジョブは関係がありませんでした。システムの複数の accept() サーバーと複数の accept() ワーカーの例では、各ワーカー・ジョブを反復サーバーに変えます。サーバー・ジョブは、以前と同様、**socket()**、**bind()**、および **listen()** API を呼び出します。 **listen()** 呼び出しが完了すると、サーバーはそれぞれのワーカー・ジョブを作成し、listen ソケットをそれぞれに与えます。それから、すべてのワーカー・ジョブが **accept()** API を呼び出します。クライアントがサーバーに接続しようとすると、1 つの **accept()** 呼び出しのみが完了し、そのワーカーが接続を処理します。

これらすべての例では、クライアント接続の基本設計が使用されています。詳細については、『例: 汎用クライアント』を参照してください。

例: 反復サーバー・プログラムの作成

この例は、すべての着信接続を処理する単一のサーバー・ジョブを作成する場合に使用します。 **accept()** API が完了すると、サーバーがトランザクション全体を処理します。以下の図は、システムが反復サーバー設計を使用した場合に、サーバーとクライアント・ジョブが対話する方法を示しています。この例で使用できる共通クライアント・ジョブのコードを含む例については、『例: 汎用クライアント』を参照してください。



ソケットのイベントのフロー: 反復サーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバー・アプリケーションとワーカー・アプリケーションの関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。このフローのクライアント部分については、『例: 汎用クライアント』を参照してください。以下のシーケンスは、反復サーバー・アプリケーション用のこのサンプル・プログラムの関数呼び出しを示しています。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。
3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。
4. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続の成功を待機して、無期限にブロックします。
5. **recv()** 関数が、クライアント・アプリケーションからデータを受信します。
6. **send()** 関数が、クライアントにデータを送り返します。
7. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

```
/*****************************************************************************/
/* Application creates an iterative server design                            */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT  12345

main (int argc, char *argv[])
{
   int    i, len, num, rc, on = 1;
   int    listen_sd, accept_sd;
   char   buffer[80];
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Create an AF_INET stream socket to receive    */
   /* incoming connections on                       */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("socket() failed");
      exit(-1);
   }

   /*************************************************/
   /* Allow socket descriptor to be reuseable       */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("setsockopt() failed");
      close(listen_sd);
      exit(-1);
   }

   /*************************************************/
```

```
/* Bind the socket                                */
/**************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port        = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/**************************************************/
/* Set the listen back log                        */
/**************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/**************************************************/
/* Inform the user that the server is ready       */
/**************************************************/
printf("The server is ready\n");

/**************************************************/
/* Go through the loop once for each connection  */
/**************************************************/
for (i=0; i < num; i++)
{
   /*********************************************/
   /* Wait for an incoming connection           */
   /*********************************************/
   printf("Interation: %d\n", i+1);
   printf("  waiting on accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("accept() failed");
      close(listen_sd);
      exit(-1);
   }
   printf("  accept completed successfully\n");

   /*********************************************/
   /* Receive a message from the client         */
   /*********************************************/
   printf("  wait for client to send us a message\n");
   rc = recv(accept_sd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("recv() failed");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("  <%s>\n", buffer);

   /*********************************************/
   /* Echo the data back to the client          */
   /*********************************************/
```

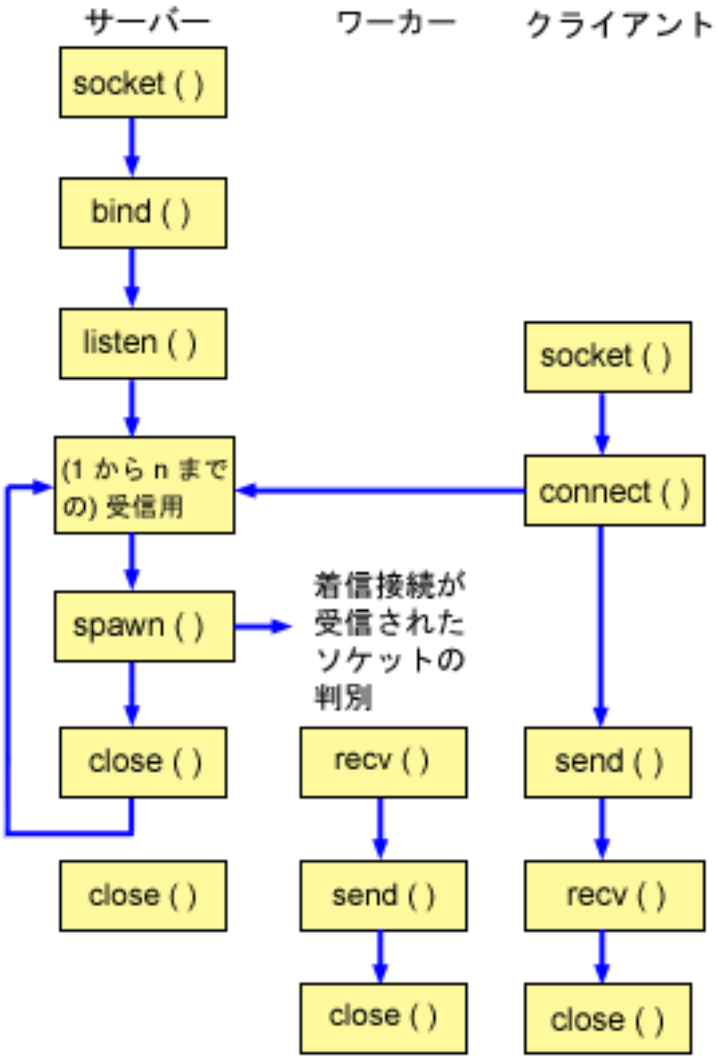
```
      printf("  echo it back\n");
      len = rc;
      rc = send(accept_sd, buffer, len, 0);
      if (rc <= 0)
      {
         perror("send() failed");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }

      /*********************************************/
      /* Close down the incoming connection        */
      /*********************************************/
      close(accept_sd);
   }

   /*********************************************/
   /* Close down the listen socket              */
   /*********************************************/
   close(listen_sd);
}
```

例: spawn() API を使用した子プロセスの作成

この例では、サーバー・プログラムが **spawn()** API を使用して、親からソケット記述子を継承する子プロセスを作成する方法を示しています。サーバー・ジョブは着信接続を待機し、それから **spawn()** を呼び出して着信接続をハンドルするための子ジョブを作成します。子プロセスは、**spawn()** 関数によって以下の属性を継承します。

- ソケットおよびファイル記述子
- 信号マスク
- 信号アクション・ベクトル
- 環境変数

以下の図は、**spawn()** サーバー設計が使用される場合に、サーバー、ワーカー、およびクライアント・ジョブが対話する方法を示しています。



ソケットのイベントのフロー: spawn() を使用して要求を受け入れ処理するサーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバーとワーカーの例の関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: spawn() を使用するサーバーを作成する』では、以下のソケット呼び出しを使用して、**spawn()** 関数呼び出しによって子プロセスを作成します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。
3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。
4. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続の成功を待機して、無期限にブロックします。
5. **spawn()** 関数が、着信要求を処理するワーカー・ジョブのパラメーターを初期設定します。この例では、新規接続のソケット記述子が子プログラムの記述子 0 にマップされます。

6. この例では、最初の **close()** 関数が listen ソケット記述子をクローズします。 2 番目の **close ()** 呼び出しは、受け入れたソケットを終了します。

ソケットのイベントのフロー: spawn() によって作成されるワーカー・ジョブ

『例: ワーカー・ジョブがデータ・バッファーを受信できるようにする』では、以下の関数呼び出しのシーケンスを使用します。

1. サーバーで **spawn()** 関数が呼び出されると、**recv()** 関数が着信接続からデータを受信します。
2. **send()** 関数が、クライアントにデータを送り返します。
3. **close()** 関数が、spawn されたワーカー・ジョブを終了します。

例: spawn() を使用するサーバーを作成する: 以下の例では、**spawn()** API を使用して、親プロセスからソケット記述子を継承する子プロセスを作成する方法を示しています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* Application creates an child process using spawn().               */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT  12345

main (int argc, char *argv[])
{
   int    i, num, pid, rc, on = 1;
   int    listen_sd, accept_sd;
   int    spawn_fdmap[1];
   char  *spawn_argv[1];
   char  *spawn_envp[1];
   struct inheritance   inherit;
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Create an AF_INET stream socket to receive    */
   /* incoming connections on                       */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("socket() failed");
      exit(-1);
   }

   /*************************************************/
   /* Allow socket descriptor to be reuseable       */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
```

```
{
   perror("setsockopt() failed");
   close(listen_sd);
   exit(-1);
}

/*********************************************/
/* Bind the socket                           */
/*********************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_port        = htons(SERVER_PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/*********************************************/
/* Set the listen back log                   */
/*********************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/*********************************************/
/* Inform the user that the server is ready  */
/*********************************************/
printf("The server is ready\n");

/*********************************************/
/* Go through the loop once for each connection  */
/*********************************************/
for (i=0; i < num; i++)
{
   /*********************************************/
   /* Wait for an incoming connection           */
   /*********************************************/
   printf("Interation: %d\n", i+1);
   printf("  waiting on accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("accept() failed");
      close(listen_sd);
      exit(-1);
   }
   printf("  accept completed successfully\n");

   /*********************************************/
   /* Initialize the spawn parameters           */
   /*                                           */

   /* The socket descriptor for the new         */
   /* connection is mapped over to descriptor 0  */
   /* in the child program.                     */
   /*********************************************/
   memset(&inherit, 0, sizeof(inherit));
   spawn_argv[0]  = NULL;
```

```
      spawn_envp[0]  = NULL;
      spawn_fdmap[0] = accept_sd;

      /*********************************************/
      /* Create the worker job                     */
      /*********************************************/
      printf("  creating worker job\n");
      pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR1.PGM",
                  1, spawn_fdmap, &inherit,
                  spawn_argv, spawn_envp);
      if (pid < 0)
      {
         perror("spawn() failed");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }
      printf("  spawn completed successfully\n");

      /*********************************************/
      /* Close down the incoming connection since  */
      /* it has been given to a worker to handle   */
      /*********************************************/
      close(accept_sd);
   }

   /*************************************************/
   /* Close down the listen socket                  */
   /*************************************************/
   close(listen_sd);
}
```

ソケット記述子を使用してプロセスを実行するサンプル・プログラムについては、『例: ワーカー・ジョブがデータ・バッファーを受信できるようにする』を参照してください。

例: ワーカー・ジョブがデータ・バッファーを受信できるようにする: この例には、ワーカー・ジョブがクライアント・ジョブからデータ・バッファーを受け取るようにし、これをそのまま戻すためのコードが含まれています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* Worker job that receives and echoes back a data buffer to a client        */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    sockfd;
   char   buffer[80];

   /*************************************************/
   /* The descriptor for the incoming connection is */
   /* passed to this worker job as a descriptor 0.  */
   /*************************************************/
   sockfd = 0;

   /*************************************************/
   /* Receive a message from the client             */
   /*************************************************/
   printf("Wait for client to send us a message\n");
   rc = recv(sockfd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
```

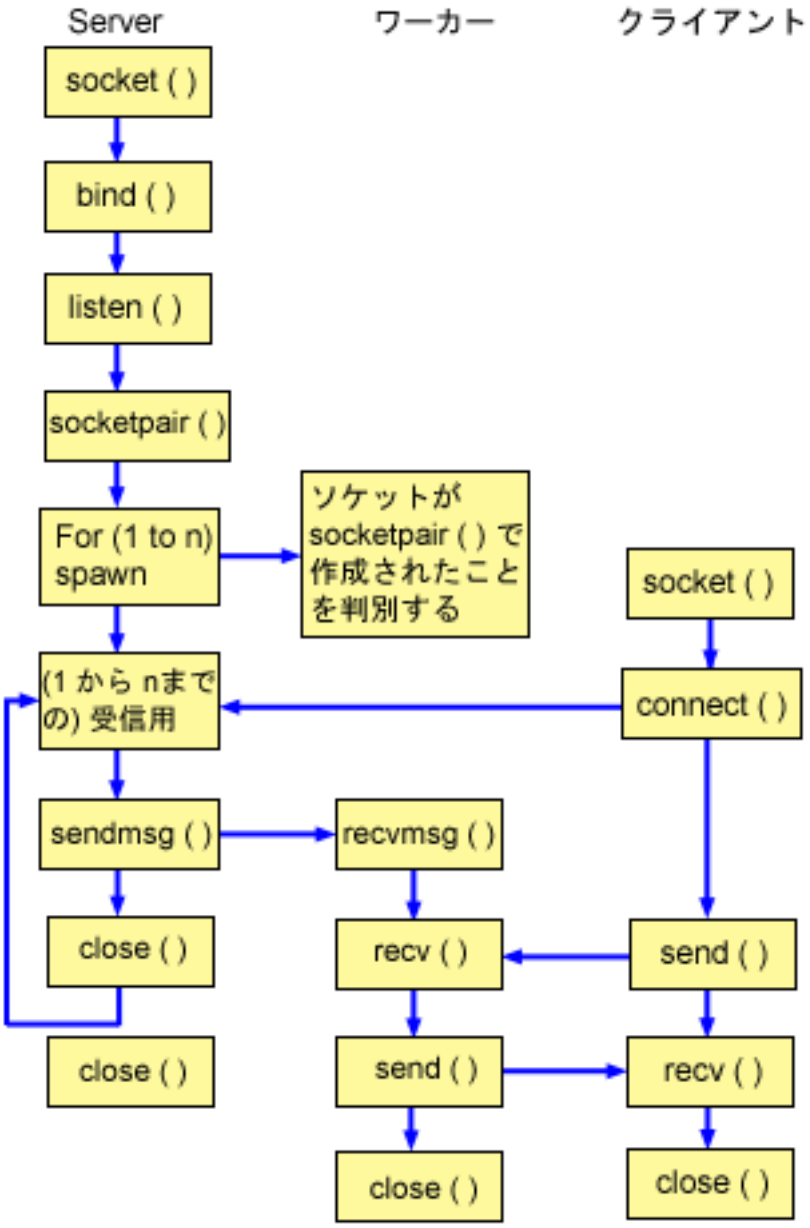
```
      perror("recv() failed");
      close(sockfd);
      exit(-1);
   }
   printf("<%s>\n", buffer);

   /*********************************************/
   /* Echo the data back to the client          */
   /*********************************************/
   printf("Echo it back\n");
   len = rc;
   rc = send(sockfd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("send() failed");
      close(sockfd);
      exit(-1);
   }

   /*********************************************/
   /* Close down the incoming connection        */
   /*********************************************/
   close(sockfd);

}
```

例: プロセス間での記述子の受け渡し

sendmsg() および **recvmsg()** の例では、これらの API を使用して着信接続をハンドルするようにサーバー・プログラムを設計する方法を示しています。サーバーが開始されると、ワーカー・ジョブのプールを作成します。これらの事前割り当てされた (spawn された) ワーカー・ジョブは、必要が生じるまで待機します。クライアント・ジョブがサーバーに接続すると、サーバーはワーカー・ジョブの 1 つに着信接続を確立します。

以下の図は、システムが **sendmsg()** および **recvmsg()** サーバー設計を使用する際にサーバー、ワーカー、およびクライアント・ジョブが対話する方法を示しています。

注: この図のクライアント部分については、『例: 汎用クライアント』を参照してください。



ソケットのイベントのフロー: sendmsg() および recvmsg() 関数を使用するサーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバーとワーカーの例の関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: sendmsg() および recvmsg() で使用するサーバー・プログラム』では、以下のソケット呼び出しを使用して、**sendmsg()** および **recvmsg()** 関数呼び出しによって子プロセスを作成します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。
3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。

4. **socketpair()** 関数が、一対の UNIX データグラム・ソケットを作成します。サーバーは **socketpair()** API を使用して、一対の AF_UNIX ソケットを作成することができます。
5. **spawn()** 関数が、着信要求を処理するワーカー・ジョブのパラメーターを初期設定します。この例の場合、作成された子ジョブは、**socketpair()** によって作成されたソケット記述子を継承します。
6. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続を待機して、無期限にブロックします。
7. **sendmsg()** 関数が、着信接続をワーカー・ジョブの 1 つに送信します。子プロセスは、**recvmsg()** 関数によって接続を受け入れます。サーバーが **sendmsg()** を呼び出したとき、子ジョブは活動状態ではありません。
8. この例では、最初の **close()** 関数が、受け入れたソケットをクローズします。 2 番目の **close** () 呼び出しは、listen ソケットを終了します。

ソケットのイベントのフロー: recvmsg() を使用するワーカー・ジョブ

『例: sendmsg () および recvmsg () で使用するワーカー・プログラム』では、以下の関数呼び出しのシーケンスを使用します。

1. サーバーが接続を受け入れ、そのソケット記述子をワーカー・ジョブに渡すと、**recvmsg()** 関数が記述子を受信します。この例の場合、**recvmsg()** 関数は、サーバーが記述子を送信するまで待機します。
2. **recv()** 関数が、クライアントからデータを受信します。
3. **send()** 関数が、クライアントにデータを送り返します。
4. **close()** 関数が、ワーカー・ジョブを終了します。

例: sendmsg() および recvmsg() で使用するサーバー・プログラム: 以下の例では、**sendmsg()** API を使用してワーカー・ジョブのプールを作成する方法を示しています。この例で使用できる共通クライアント・ジョブのコードを含む例については、『例: 汎用クライアント』を参照してください。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* Server example that uses sendmsg() to create worker jobs                  */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT  12345

main (int argc, char *argv[])
{
   int    i, num, pid, rc, on = 1;
   int    listen_sd, accept_sd;
   int    server_sd, worker_sd, pair_sd[2];
   int    spawn_fdmap[1];
   char  *spawn_argv[1];
   char  *spawn_envp[1];
   struct inheritance   inherit;
   struct msghdr        msg;
   struct sockaddr_in   addr;

   /************************************************/
   /* If an argument was specified, use it to      */
   /* control the number of incoming connections   */
   /************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
```

```
   num = 1;

/*************************************************/
/* Create an AF_INET stream socket to receive    */
/* incoming connections on                       */
/*************************************************/
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if (listen_sd < 0)
{
   perror("socket() failed");
   exit(-1);
}

/*************************************************/
/* Allow socket descriptor to be reuseable       */
/*************************************************/
rc = setsockopt(listen_sd,
                SOL_SOCKET,  SO_REUSEADDR,
                (char *)&on, sizeof(on));
if (rc < 0)
{
   perror("setsockopt() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Bind the socket                               */
/*************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port        = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Set the listen back log                       */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Create a pair of UNIX datagram sockets        */
/*************************************************/
rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, pair_sd);
if (rc != 0)
{
   perror("socketpair() failed");
   close(listen_sd);
   exit(-1);
}
server_sd = pair_sd[0];
worker_sd = pair_sd[1];

/*************************************************/
```

```
/* Initialize parms prior to entering for loop   */
/*                                               */
/* The worker socket descriptor is mapped to     */
/* descriptor 0 in the child program.            */
/*************************************************/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0]  = NULL;
spawn_envp[0]  = NULL;
spawn_fdmap[0] = worker_sd;

/*************************************************/
/* Create each of the worker jobs                */
/*************************************************/
printf("Creating worker jobs...\n");
for (i=0; i < num; i++)
{
   pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR2.PGM",
               1, spawn_fdmap, &inherit,
               spawn_argv, spawn_envp);
   if (pid < 0)
   {
      perror("spawn() failed");
      close(listen_sd);
      close(server_sd);
      close(worker_sd);
      exit(-1);
   }
   printf("  Worker = %d\n", pid);
}

/*************************************************/
/* Close down the worker side of the socketpair  */
/*************************************************/
close(worker_sd);

/*************************************************/
/* Inform the user that the server is ready      */
/*************************************************/
printf("The server is ready\n");

/*************************************************/
/* Go through the loop once for each connection  */
/*************************************************/
for (i=0; i < num; i++)
{
   /**********************************************/
   /* Wait for an incoming connection            */
   /**********************************************/
   printf("Interation: %d\n", i+1);
   printf("  waiting on accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("accept() failed");
      close(listen_sd);
      close(server_sd);
      exit(-1);
   }
   printf("  accept completed successfully\n");

   /**********************************************/
   /* Initialize message header structure        */
   /**********************************************/
   memset(&msg, 0, sizeof(msg));

   /**********************************************/
   /* We are not sending any data so we do not   */
```

```
      /* need to set either of the msg_iov fields.  */
      /* The memset of the message header structure */
      /* will set the msg_iov pointer to NULL and   */
      /* it will set the msg_iovcnt field to 0.     */
      /*********************************************/

      /*********************************************/
      /* The only fields in the message header      */
      /* structure that need to be filled in are    */
      /* the msg_accrights fields.                  */
      /*********************************************/
      msg.msg_accrights  = (char *)&accept_sd;
      msg.msg_accrightslen = sizeof(accept_sd);

      /*********************************************/
      /* Give the incoming connection to one of the */
      /* worker jobs.                               */
      /*                                            */
      /* NOTE: We do not know which worker job will */
      /*       get this inbound connection.         */
      /*********************************************/
      rc = sendmsg(server_sd, &msg, 0);
      if (rc < 0)
      {
         perror("sendmsg() failed");
         close(listen_sd);
         close(accept_sd);
         close(server_sd);
         exit(-1);
      }
      printf("  sendmsg completed successfully\n");

      /*********************************************/
      /* Close down the incoming connection since   */
      /* it has been given to a worker to handle    */
      /*********************************************/
      close(accept_sd);
   }

   /*********************************************/
   /* Close down the server and listen sockets      */
   /*********************************************/
   close(server_sd);
   close(listen_sd);
}
```

例: sendmsg() および recvmsg() で使用するワーカー・プログラム: 以下の例では、**recvmsg()** API クライアント・ジョブを使用してワーカー・ジョブを受信する方法を示しています。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* Worker job that uses the recvmsg to process client requests            */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    worker_sd, pass_sd;
   char   buffer[80];
   struct iovec   iov[1];
   struct msghdr  msg;

   /*********************************************/
```

```
/* One of the socket descriptors that was        */
/* returned by socketpair(), is passed to this   */
/* worker job as descriptor 0.                   */
/*************************************************/
worker_sd = 0;

/*************************************************/
/* Initialize message header structure           */
/*************************************************/
memset(&msg,   0, sizeof(msg));
memset(iov,    0, sizeof(iov));

/*************************************************/
/* The recvmsg() call will NOT block unless a    */
/* non-zero length data buffer is specified      */
/*************************************************/
iov[0].iov_base = buffer;
iov[0].iov_len  = sizeof(buffer);
msg.msg_iov     = iov;
msg.msg_iovlen  = 1;

/*************************************************/
/* Fill in the msg_accrights fields so that we   */
/* can receive the descriptor                    */
/*************************************************/
msg.msg_accrights    = (char *)&pass_sd;
msg.msg_accrightslen = sizeof(pass_sd);

/*************************************************/
/* Wait for the descriptor to arrive             */
/*************************************************/
printf("Waiting on recvmsg\n");
rc = recvmsg(worker_sd, &msg, 0);
if (rc < 0)
{
   perror("recvmsg() failed");
   close(worker_sd);
   exit(-1);
}
else if (msg.msg_accrightslen <= 0)
{
   printf("Descriptor was not received\n");
   close(worker_sd);
   exit(-1);
}
else
{
   printf("Received descriptor = %d\n", pass_sd);
}

/*************************************************/
/* Receive a message from the client             */
/*************************************************/
printf("Wait for client to send us a message\n");
rc = recv(pass_sd, buffer, sizeof(buffer), 0);
if (rc <= 0)
{
   perror("recv() failed");
   close(worker_sd);
   close(pass_sd);
   exit(-1);
}
printf("<%s>\n", buffer);

/*************************************************/
/* Echo the data back to the client              */
/*************************************************/
```

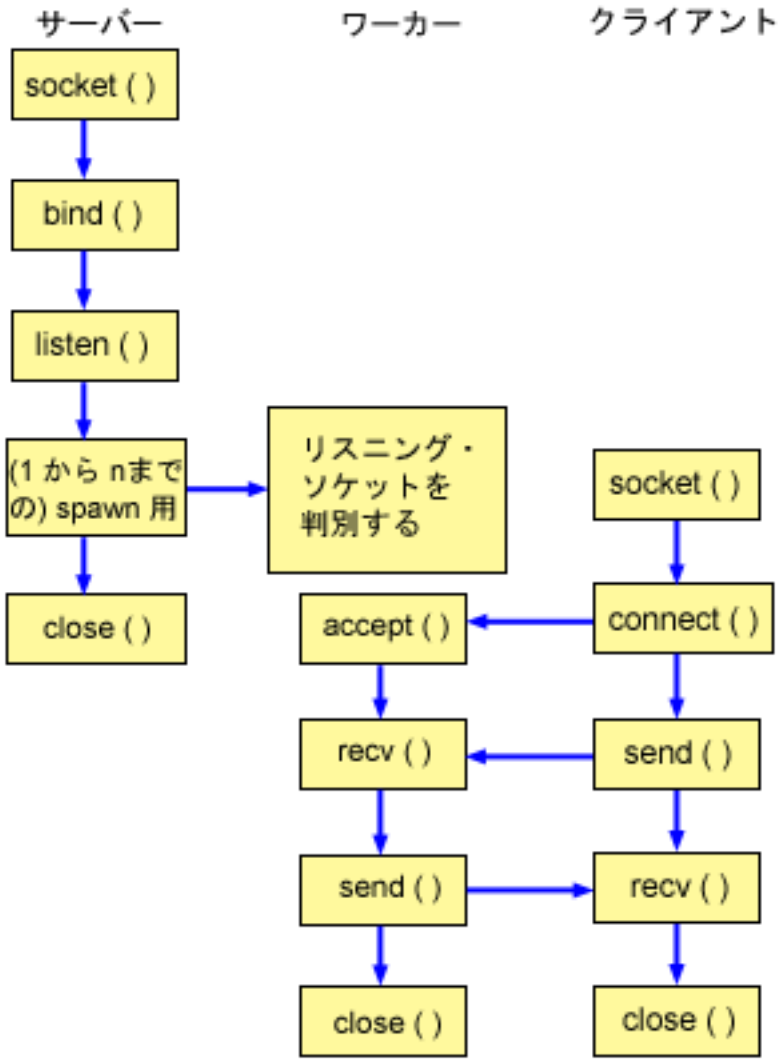
```
   printf("Echo it back\n");
   len = rc;
   rc = send(pass_sd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("send() failed");
      close(worker_sd);
      close(pass_sd);
      exit(-1);
   }

   /*********************************************/
   /* Close down the descriptors                */
   /*********************************************/
   close(worker_sd);
   close(pass_sd);
}
```

例: 複数の accept() API を使用した着信要求のハンドル

以下の例では、複数の **accept()** モデルを使用するサーバー・プログラムを設計して、着信接続要求をハンドリングする方法を示しています。複数の **accept()** サーバーが始動すると、通常どおり **socket()**、**bind()**、および **listen()** を実行します。それからワーカー・ジョブのプールを作成し、それぞれのワーカー・ジョブに listen ソケットを与えます。それぞれの複数の **accept()** ワーカーは、**accept()** を呼び出します。

以下の図は、システムが複数の **accept()** サーバー設計を使用する際に、サーバー、ワーカー、およびクライアント・ジョブがどのように対話するかを示しています。

注: この図のクライアント部分については、『例: 汎用クライアント』を参照してください。



ソケットのイベントのフロー: 複数の accept() ワーカー・ジョブのプールを作成するサーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバーとワーカーの例の関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: 複数の accept() ワーカー・ジョブのプールを作成するためのサーバー・プログラム』では、以下のソケット呼び出しを使用して子プロセスを作成します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。
3. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。
4. **spawn()** 関数が、各ワーカー・ジョブを作成します。
5. この例では、最初の **close()** 関数が listen ソケットをクローズします。

ソケットのイベントのフロー: 複数の accept() を使用するワーカー・ジョブ

『例: 複数の accept() 用のワーカー・ジョブ』では、以下の関数呼び出しのシーケンスを使用します。

1. サーバーがワーカー・ジョブを spawn した後、このワーカー・ジョブに listen ソケット記述子が、コマンド行パラメーターとして渡されます。 **accept()** 関数が、着信クライアント接続を待機します。
2. **recv()** 関数が、クライアントからデータを受信します。
3. **send()** 関数が、クライアントにデータを送り返します。
4. **close()** 関数が、ワーカー・ジョブを終了します。

例: 複数の accept() ワーカー・ジョブのプールを作成するためのサーバー・プログラム: 以下の例では、複数の **accept()** モデルを使用して、ワーカー・ジョブをプールを作成する方法を示しています。この例で使用できる共通クライアント・ジョブのコードを含む例については、『例: 汎用クライアント』を参照してください。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* Server example creates a pool of worker jobs with multiple accept()    */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT  12345

main (int argc, char *argv[])
{
   int    i, num, pid, rc, on = 1;
   int    listen_sd, accept_sd;
   int    spawn_fdmap[1];
   char  *spawn_argv[1];
   char  *spawn_envp[1];
   struct inheritance   inherit;
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Create an AF_INET stream socket to receive    */
   /* incoming connections on                       */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("socket() failed");
      exit(-1);
   }

   /*************************************************/
   /* Allow socket descriptor to be reuseable       */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
```

```
{
   perror("setsockopt() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Bind the socket                               */
/*************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port        = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Set the listen back log                       */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Initialize parms prior to entering for loop   */
/*                                               */
/* The listen socket descriptor is mapped to     */
/* descriptor 0 in the child program.            */
/*************************************************/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0]  = NULL;
spawn_envp[0]  = NULL;
spawn_fdmap[0] = listen_sd;

/*************************************************/
/* Create each of the worker jobs                */
/*************************************************/
printf("Creating worker jobs...\n");
for (i=0; i < num; i++)
{
   pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR4.PGM",
               1, spawn_fdmap, &inherit,
               spawn_argv, spawn_envp);
   if (pid < 0)
   {
      perror("spawn() failed");
      close(listen_sd);
      exit(-1);
   }
   printf("  Worker = %d\n", pid);
}

/*************************************************/
/* Inform the user that the server is ready      */
/*************************************************/
printf("The server is ready\n");
```

```
   /*********************************************/
   /* Close down the listening socket           */
   /*********************************************/
   close(listen_sd);
}
```

例: 複数の accept() 用のワーカー・ジョブ: 以下の例は、複数の **accept()** API がワーカー・ジョブを受信し、**accept()** サーバーを呼び出す方法について示します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* Worker job uses multiple accept() to handle incoming client connections*/
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    listen_sd, accept_sd;
   char   buffer[80];

   /*********************************************/
   /* The listen socket descriptor is passed to     */
   /* this worker job as a command line parameter   */
   /*********************************************/
   listen_sd = 0;

   /*********************************************/
   /* Wait for an incoming connection           */
   /*********************************************/
   printf("Waiting on accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("accept() failed");
      close(listen_sd);
      exit(-1);
   }
   printf("Accept completed successfully\n");

   /*********************************************/
   /* Receive a message from the client         */
   /*********************************************/
   printf("Wait for client to send us a message\n");
   rc = recv(accept_sd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("recv() failed");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("<%s>\n", buffer);

   /*********************************************/
   /* Echo the data back to the client          */
   /*********************************************/
   printf("Echo it back\n");
   len = rc;
   rc = send(accept_sd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("send() failed");
      close(listen_sd);
```

```
      close(accept_sd);
      exit(-1);
   }

   /*************************************************/
   /* Close down the descriptors                    */
   /*************************************************/
   close(listen_sd);
   close(accept_sd);
}
```

例: 汎用クライアント

以下のコード例に、共通クライアント・ジョブのコードが含まれています。クライアント・ジョブは、**socket()**、**connect()**、**send()**、**recv()**、および **close()** を実行します。クライアント・ジョブは、これが送受信するデータ・バッファーがサーバーではなくワーカー・ジョブに入ることを認識しません。サーバーが AF_INET アドレス・ファミリーまたは AF_INET6 アドレス・ファミリーのどちらであっても動作するクライアント・アプリケーションを作成したい場合、『例: IPv4 または IPv6 クライアント』を使用してください。

このクライアント・ジョブは、以下のそれぞれの共通コネクション型サーバー設計を処理します。

- 反復サーバー。サンプル・プログラムについては、『例: 反復サーバー・プログラムの作成』を参照してください。
- spawn サーバーおよびワーカー。サンプル・プログラムについては、『例: spawn() API を使用した子プロセスの作成』を参照してください。
- sendmsg() サーバーおよび rcvmsg() ワーカー。サンプル・プログラムについては、『例: sendmsg() および recvmsg() で使用するサーバー・プログラム』を参照してください。
- 複数の accept() 設計。サンプル・プログラムについては、『例: 複数の accept() ワーカー・ジョブのプールを作成するためのサーバー・プログラム』を参照してください。
- 非ブロッキング入出力および select() 設計。サンプル・プログラムについては、『例: 非ブロッキング入出力および select()』を参照してください。
- IPv4 クライアントまたは IPv6 クライアントからの接続を受け入れるサーバー。サンプル・プログラムについては、『例: IPv6 クライアントと IPv4 クライアントの両方から接続を受け入れる』を参照してください。

ソケットのイベントのフロー: 汎用クライアント

次のサンプル・プログラムは、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. ソケット記述子を受信したら、**connect()** 関数を使用して、サーバーへの接続を確立します。
3. **send()** 関数が、データ・バッファーをワーカー・ジョブに送信します。
4. **recv()** 関数が、データ・バッファーをワーカー・ジョブから受信します。
5. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* Generic client example is used with connection-oriented server designs */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
```

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

#define SERVER_PORT  12345

main (int argc, char *argv[])
{
   int    len, rc;
   int    sockfd;
   char   send_buf[80];
   char   recv_buf[80];
   struct sockaddr_in   addr;

   /************************************************/
   /* Create an AF_INET stream socket              */
   /************************************************/
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
   {
      perror("socket");
      exit(-1);
   }

   /************************************************/
   /* Initialize the socket address structure      */
   /************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family      = AF_INET;
   addr.sin_addr.s_addr = htonl(INADDR_ANY);
   addr.sin_port        = htons(SERVER_PORT);

   /************************************************/
   /* Connect to the server                        */
   /************************************************/
   rc = connect(sockfd,
                (struct sockaddr *)&addr,
                sizeof(struct sockaddr_in));
   if (rc < 0)
   {
      perror("connect");
      close(sockfd);
      exit(-1);
   }
   printf("Connect completed.\n");

   /************************************************/
   /* Enter data buffer that is to be sent         */
   /************************************************/
   printf("Enter message to be sent:\n");
   gets(send_buf);

   /************************************************/
   /* Send data buffer to the worker job           */
   /************************************************/
   len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
   if (len != strlen(send_buf) + 1)
   {
      perror("send");
      close(sockfd);
      exit(-1);
   }
   printf("%d bytes sent\n", len);

   /************************************************/
   /* Receive data buffer from the worker job      */
   /************************************************/
   len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
```

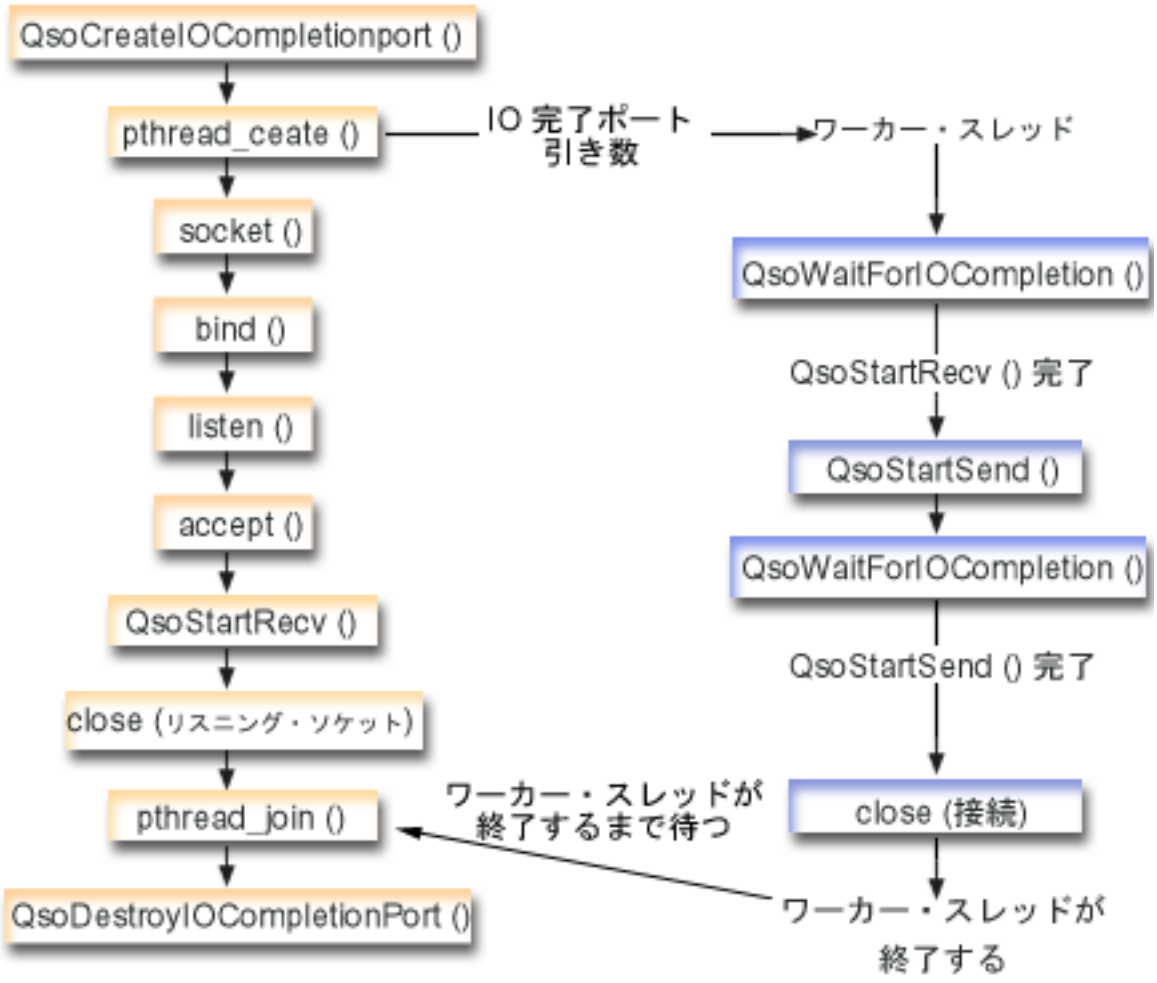
```
   if (len != strlen(send_buf) + 1)
   {
      perror("recv");
      close(sockfd);
      exit(-1);
   }
   printf("%d bytes received\n", len);

   /************************************************/
   /* Close down the socket                        */
   /************************************************/
   close(sockfd);
}
```

例: 非同期入出力 API の使用

アプリケーションは、**QsoCreateIOCompletionPort()** API を使用して入出力完了ポートを作成します。この API は、非同期入出力要求の完了をスケジュールして待機するために使用できるハンドルを戻します。アプリケーションは、入出力完了ポート・ハンドルを指定して、入力関数または出力関数を開始します。入出力の完了時に、状況情報とアプリケーション定義のハンドルが、指定した入出力完了ポートに通知されます。入出力完了ポートへの通知によって、おそらく多数ある待機中のスレッドのうちの 1 つだけがウェイクアップされます。アプリケーションは、以下を受信します。

- 元の要求で提供されたバッファー
- そのバッファーとやり取りして処理されたデータの長さ
- 完了した入出力操作のタイプの表示
- 初期入出力要求で渡されたアプリケーション定義のハンドル

このアプリケーション・ハンドルは、単にクライアント接続を識別するソケット記述子である場合もあれば、クライアント接続の状態についての広範な情報が入っているストレージを指すポインターである場合もあります。操作が完了してアプリケーション・ハンドルが渡されたので、ワーカー・スレッドはクライアント接続を完了するための次のステップを決定します。これらの完了非同期操作を処理するワーカー・スレッドは、1 つのクライアント要求だけに拘束されるのではなく、さまざまなクライアント要求を処理します。ユーザー・バッファーとやり取りするコピーは、サーバー・プロセスと非同期で発生するので、クライアント要求の待機時間は減少します。これは、複数のプロセッサーがあるシステムでは利点があります。



ソケットのイベントのフロー: 非同期入出力サーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバーとワーカーの例の関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。このフローは、以下のサンプル・アプリケーションでのソケット呼び出しを示しています。このサーバー例を、汎用クライアントの例と一緒に使用してください。

1. マスター・スレッドは、**QsoCreateIOCompletionPort()** を呼び出すことによって、入出力完了ポートを作成します。
2. マスター・スレッドは、**pthread_create** 関数によって、入出力完了ポート要求を処理するためにワーカー・スレッドのプールを作成します。
3. ワーカー・スレッドは、クライアント要求が処理を行うことを待機する **QsoWaitForIOCompletionPort()** を呼び出します。
4. マスター・スレッドはクライアント接続を受け入れ、ワーカー・スレッドが待機している入出力完了ポートを指定する **QsoStartRecv()** を発行するようになります。

   注: QsoStartAccept() を使用することによって、受け入れを非同期で使用することもできます。

5. ある時点で、クライアント要求はサーバー・プロセスに対して非同期で到着します。ソケット・オペレーティング・システムは、提供されたユーザー・バッファーをロードし、完了した **QsoStartRecv()** 要求を、指定した入出力完了ポートに送信します。 1 つのワーカー・スレッドがウェイクアップされ、この要求の処理を続行します。
6. ワーカー・スレッドは、アプリケーション定義のハンドルからクライアント・ソケット記述子を取り出し、**QsoStartSend()** 操作を実行することによって、受信したデータをクライアントにエコーするようになります。
7. データが即時に送信可能な場合は、**QsoStartSend()** がその事実の通知を戻し、そうでない場合は、ソケット・オペレーティング・システムがデータを可能な限り早く送信し、その事実を、指定した入出力完了ポートに通知します。ワーカー・スレッドは、データが送信されたという通知を得て、入出力完了ポートで別の要求を待機するか、終了するように命令が与えられる場合は終了します。ワーカー・スレッド終了イベントを通知するために、マスター・スレッドが **QsoPostIOCompletion()** を使用できます。
8. マスター・スレッドは、ワーカー・スレッドが終了するのを待ち、次いで **QsoDestroyIOCompletionPort()** を呼び出すことによって入出力完了ポートを破棄します。

注: このサーバー例は、『例: 汎用クライアント』で説明されている共通クライアント・コードを処理します。

この例では、サーバーが非同期 API を使用できる方法を示します。コード例の使用については、『コードの特記事項情報』を参照してください。

```
      #include <stdio.h>
      #include <stdlib.h>
      #include <string.h>
      #include <sys/time.h>
      #include <sys/types.h>
      #include <sys/socket.h>
      #include <netinet/in.h>
      #include <errno.h>
      #include <unistd.h>
      #define _MULTI_THREADED
      #include "pthread.h"
      #include "qsoasync.h"
      #define BufferLength 80
      #define Failure 0
      #define Success 1
      #define SERVPORT 12345

void *workerThread(void *arg);

/*********************************************************************/
/*                                                                   */
/* Function Name: main                                               */
/*                                                                   */
/* Descriptive Name:  Master thread will establish a client          */
/*      connection and hand processing responsibility                */
/*      to a worker thread.                                          */
/* Note: Due to the thread attribute of this program, spawn() must   */
/*       be used to invoke.                                          */
/*********************************************************************/

int main()
{
       int listen_sd, client_sd, rc;
       int  on = 1, ioCompPort;
       pthread_t thr;
       void *status;
       char buffer[BufferLength];
       struct sockaddr_in serveraddr;
       Qso_OverlappedIO_t ioStruct;
```

```
/*********************************************/
/* Create an I/O completion port for this    */
/* process.                                  */
/*********************************************/
if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
{
  perror("QsoCreateIOCompletionPort() failed");
  exit(-1);
}

/*********************************************/
/* Create a worker thread to                 */
/* to process all client requests. The       */
/* worker thread will wait for client        */
/* requests to arrive on the I/O completion  */
/* port just created.                        */
/*********************************************/
rc = pthread_create(&thr, NULL, workerThread,
                    &ioCompPort);
if (rc < 0)
{
  perror("pthread_create() failed");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Create an AF_INET stream socket to receive*/
/* incoming connections on                   */
/*********************************************/
if ((listen_sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
  perror("socket() failed");
  QsoDestroyIOCompletionPort(ioCompPort);
  exit(-1);
}

/*********************************************/
/* Allow socket descriptor to be reuseable   */
/*********************************************/
if ((rc = setsockopt(listen_sd, SOL_SOCKET,
                     SO_REUSEADDR,
                     (char *)&on,
                     sizeof(on))) < 0)
{
  perror("setsockopt() failed");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* bind the socket                           */
/*********************************************/
memset(&serveraddr, 0x00, sizeof(struct sockaddr_in));
serveraddr.sin_family        = AF_INET;
serveraddr.sin_port          = htons(SERVPORT);
serveraddr.sin_addr.s_addr   = htonl(INADDR_ANY);

if ((rc = bind(listen_sd,
               (struct sockaddr *)&serveraddr,
                sizeof(serveraddr))) < 0)
{
  perror("bind() failed");
```

```
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Set listen backlog                        */
/*********************************************/
if ((rc = listen(listen_sd, 10)) < 0)
{
  perror("listen() failed");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

printf("Waiting for client connection.\n");

/*********************************************/
/* accept an incoming client connection.     */
/*********************************************/
if ((client_sd = accept(listen_sd, (struct sockaddr *)NULL,
                  NULL)) < 0)
{
  perror("accept() failed");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Issue QsoStartRecv() to receive client    */
/* request.                                  */
/* Note:                                     */
/*    postFlag == on denoting request should */
/*                posted to the I/O          */
/*                completion port, even if   */
/*                if request is immediately  */
/*                available.  Worker thread  */
/*                will process client        */
/*                request.                   */
/*********************************************/

/*********************************************/
/* initialize Qso_OverlappedIO_t structure - */
/* reserved fields must be hex 00's.         */
/*********************************************/
memset(&ioStruct, '\0', sizeof(ioStruct));

ioStruct.buffer = buffer;
ioStruct.bufferLength = sizeof(buffer);


/*********************************************/
/* Store the client descriptor in the        */
/* Qso_OverlappedIO_t descriptorHandle field.*/
/* This area is used to house information    */
/* defining the state of the client          */
/* connection. Field descriptorHandle is     */
/* defined as a (void *) to allow the server */
/* to address more extensive client          */
/* connection state if needed.               */
/*********************************************/
*((int*)&ioStruct.descriptorHandle) = client_sd;
ioStruct.postFlag = 1;
ioStruct.fillBuffer = 0;
```

```
        rc = QsoStartRecv(client_sd, ioCompPort, &ioStruct);
        if (rc == -1)
        {
          perror("QsoStartRecv() failed");
          QsoDestroyIOCompletionPort(ioCompPort);
          close(listen_sd);
          close(client_sd);
          exit(-1);
        }
        /*******************************************/
        /* close the server's listening socket.    */
        /*******************************************/
        close(listen_sd);

        /*******************************************/
        /* Wait for worker thread to finish        */
        /* processing client connection.           */
        /*******************************************/
        rc = pthread_join(thr, &status);

        QsoDestroyIOCompletionPort(ioCompPort);
        if ( rc == 0 && (rc = __INT(status)) == Success)
        {
          printf("Success.\n");
          exit(0);
        }
        else
        {
          perror("pthread_join() reported failure");
          exit(-1);
        }
}
/* end workerThread */



/*********************************************************************/
/*                                                                   */
/* Function Name: workerThread                                       */
/*                                                                   */
/* Descriptive Name:  Process client connection.                     */
/*********************************************************************/
void *workerThread(void *arg)
{
      struct timeval waitTime;
      int ioCompPort, clientfd;
      Qso_OverlappedIO_t ioStruct;
      int rc, tID;
      pthread_t thr;
      pthread_id_np_t t_id;
      t_id = pthread_getthreadid_np();
      tID = t_id.intId.lo;

      /*******************************************/
      /* I/O completion port is passed to this   */
      /* routine.                                */
      /*******************************************/
      ioCompPort = *(int *)arg;

      /*******************************************/
      /* Wait on the supplied I/O completion port  */
      /* for a client request.                   */
      /*******************************************/
      waitTime.tv_sec = 500;
      waitTime.tv_usec = 0;
      rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
      if (rc == 1 && ioStruct.returnValue != -1)
```

```
/*******************************************/
/* Client request has been received.       */
/*******************************************/
  ;
else
{
  printf("QsoWaitForIOCompletion() or QsoStartRecv() failed.\n");
  perror("QsoWaitForIOCompletion() or QsoStartRecv() failed");
  return __VOID(Failure);
}

/*******************************************/
/* Obtain the socket descriptor associated */
/* with the client connection.             */
/*******************************************/
clientfd = *((int *) &ioStruct.descriptorHandle);

/*******************************************/
/* Echo the data back to the client.       */
/* Note: postFlag == 0. If write completes */
/* immediate then indication will be       */
/* returned, otherwise once the            */
/* write is performed the I/O Completion   */
/* port will be posted.                    */
/*******************************************/
ioStruct.postFlag = 0;
ioStruct.bufferLength = ioStruct.returnValue;
rc = QsoStartSend(clientfd, ioCompPort, &ioStruct);

if (rc == 0)
/*******************************************/
/* Operation complete  - data has been sent. */
/*******************************************/
  ;
else
{
/*******************************************/
/* Two possibilities                       */
/*    rc == -1                             */
/*      Error on function call             */
/*    rc ==  1                             */
/*      Write could not be immediately     */
/*      performed. Once complete, the I/O  */
/*      completion port will be posted.    */
/*******************************************/

  if (rc == -1)
  {
    printf("QsoStartSend() failed.\n");
    perror("QsoStartSend() failed");
    close(clientfd);
    return __VOID(Failure);
  }
 /*******************************************/
 /* Wait for operation to complete.         */
 /*******************************************/
  rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
  if (rc == 1 && ioStruct.returnValue != -1)
 /*******************************************/
 /* Send sucessful.                         */
 /*******************************************/
    ;
  else
  {
    printf("QsoWaitForIOCompletion() or QsoStartSend() failed.\n");
    perror("QsoWaitForIOCompletion() or QsoStartSend() failed");
    return __VOID(Failure);
```

```
        }
      }
      close(clientfd);
      return __VOID(Success);
} /* end workerThread */
```

例: セキュア接続の確立

グローバル・セキュア・ツールキット (GSKit) API または SSL_ API のどちらかを使用して、セキュア・サーバーとクライアントを作成できます。 GSKit API は複数の IBM @server プラットフォームでセキュア接続を提供するため、こちらをお勧めします。 SSL_API は OS/400 固有のものに過ぎません。それぞれのセキュア・ソケット API のセットには、セキュア・ソケット接続を確立する際のエラーを識別するのに役立つ戻りコードがあります。これらのエラー・メッセージの情報を表示する方法については、『セキュア・ソケット API のエラー・コード・メッセージ』を参照してください。

以下の例では、以下のそれぞれの方法を用いて、セキュア・サーバーとクライアントを確立する方法を説明します。

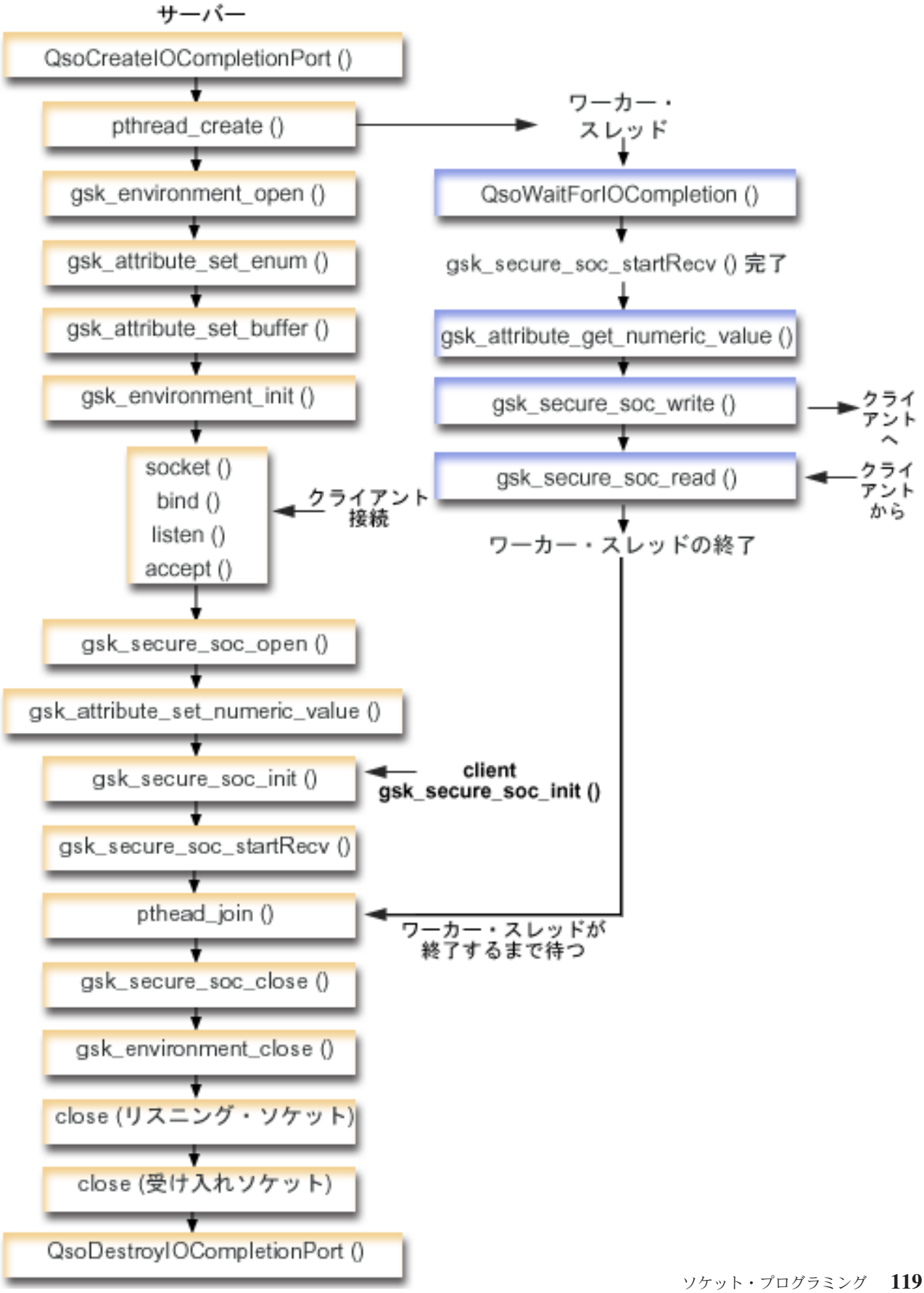
- 例: 非同期データ受信を使用する GSKit セキュア・サーバー
- 例: 非同期ハンドシェークを使用する GSKit セキュア・サーバー
- 例: GSKit API によってセキュア・クライアントを確立する
- 例: SSL_ API によってセキュア・セーバーを確立する
- 例: SSL_ API によってセキュア・クライアントを確立する

例: 非同期データ受信を使用する GSKit セキュア・サーバー

以下のコード例は、グローバル・セキュア・ツールキット (GSKit) API を使用して、セキュア・サーバーを確立するのに使用できます。サーバーはソケットをオープンし、セキュア環境を準備します。また接続要求を受け入れて処理し、クライアントとデータを交換して、セッションを終了します。クライアントもソケットをオープンし、セキュア環境をセットアップします。さらにサーバーを呼び出してセキュア接続を要求し、サーバーとデータを交換して、セッションを閉じます。以下の図と説明に、サーバー/クライアントのイベントのフローを示します。

注: 以下のプログラム例では AF_INET アドレス・ファミリーを使用しますが、AF_INET6 アドレス・ファミリーを使用するように変更することもできます。

ソケットのイベントのフロー: 非同期データ受信を使用するセキュア・サーバー


サーバー
QsoCreateIOCompletionPort ()
pthread_create ()
ワーカー・
スレッド
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_enum ()
gsk_secure_soc_startRecv () 完了
gsk_attribute_set_buffer ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_write ()
クライアントへ
socket ()
bind ()
listen ()
accept ()
クライアント
接続
gsk_secure_soc_read ()
クライアントから
ワーカー・スレッドの終了
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
gsk_secure_soc_init ()
client
gsk_secure_soc_init ()
gsk_secure_soc_startRecv ()
pthead_join ()
ワーカー・スレッドが
終了するまで待つ
gsk_secure_soc_close ()
gsk_environment_close ()
close (リスニング・ソケット)
close (受け入れソケット)
QsoDestroyIOCompletionPort ()

この図のクライアントの部分を表示するには、セキュア GSKit クライアントの図を参照してください。

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、サーバーとクライアントの例の関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。このフローは、以下のサンプル・アプリケーションでのソケット呼び出しを示しています。 .

1. **QsoCreateIOCompletionPort()** 関数が、入出力完了ポートを作成します。
2. **pthread_create** 関数が、クライアントからデータを受信したり、それをクライアントに送り返したりするためのワーカー・スレッドを作成します。ワーカー・スレッドは、ここで作成した入出力完了ポートにクライアント要求が到着するまで待機します。
3. SSL 環境へのハンドルを入手するための **gsk_environment_open()** への呼び出し。
4. SSL 環境の属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、GSK_OS400_APPLICATION_ID 値または GSK_KEYRING_FILE 値を設定するための、**gsk_attribute_set_buffer()** への呼び出し。どちらか一方の値のみを設定します。GSK_OS400_APPLICATION_ID 値を使用することを推奨します。 **gsk_attribute_set_enum()** を使用して、アプリケーション (クライアントまたはサーバー) のタイプ (GSK_SESSION_TYPE) も必ず設定してください。
5. **gsk_environment_init()** への呼び出し。この呼び出しは、SSL を処理するためのこの環境を初期設定し、この環境を使用して実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。
6. **socket** 関数がソケット記述子を作成します。そしてサーバーは、着信接続要求を受け入れることができるよう、標準的なソケット呼び出しのセット (**bind()**、**listen()**、および **accept()**) を発行します。
7. **gsk_secure_soc_open()** 関数が、セキュア・セッション用のストレージを入手し、属性のデフォルト値を設定し、保管してセキュア・セッションに関連した関数呼び出しで使用する必要のあるハンドルを戻します。
8. セキュア・セッションの属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、特定のソケットをこのセキュア・セッションに関連付けるための **gsk_attribute_set_numeric_value()** への呼び出し。
9. 暗号パラメーターの SSL ハンドシェーク・ネゴシエーションを開始するための **gsk_secure_soc_init()** への呼び出し。

   注: 通常は、サーバー・プログラムが SSL ハンドシェークに必要な証明書を提示しないと、通信は成功しません。またサーバーは、サーバー証明書に関連した秘密鍵と、証明書が保管されているキー・データベース・ファイルへアクセスできなければなりません。場合によっては、SSL ハンドシェーク処理中にクライアントも証明書を提示しなければならないこともあります。そうなるのは、クライアントが接続しているサーバーで、クライアント認証が使用可能にされている場合です。 **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) または **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API 呼び出しは、ハンドシェーク中に使用される証明書および秘密鍵の入手先のキー・データベース・ファイルを (それぞれ異なる方法で) 識別します。
10. **gsk_secure_soc_startRecv()** 関数が、セキュア・セッションで非同期受信操作を開始します。
11. **pthread_join** が、サーバーとワーカー・プログラムを同期化します。この関数はスレッドが終了するまで待機してから、スレッドを切り離し、スレッド終了状況をサーバーに戻します。
12. **gsk_secure_soc_close()** 関数が、セキュア・セッションを終了します。

13. **gsk_environment_close()** 関数が、SSL 関数をクローズします。
14. **close()** 関数が、listen ソケットを終了します。
15. **close()** が、受け入れた (クライアント接続) ソケットを終了します。
16. **QsoDestroyIOCompletionPort()** 関数が、完了ポートを破棄します。

ソケットのイベントのフロー: GSKit API を使用するワーカー・スレッド

1. サーバー・アプリケーションがワーカー・スレッドを作成した後、サーバーが **gsk_secure_soc_startRecv()** 呼び出しによって、クライアント・データを処理するよう、着信クライアント要求を送ってくるのを待ちます。 **QsoWaitForIOCompletionPort()** 関数は、サーバーによって指定された、提供された入出力完了ポートで待機します。
2. クライアント要求を受信すると、 **gsk_attribute_get_numeric_value()** 関数が、セキュア・セッションに関連するソケット記述子を取得します。
3. **gsk_secure_soc_write()** 関数が、セキュア・セッションを使用してメッセージをクライアントに送信します。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/* GSK Asyncronous Server Program using Application Id*/

/* "IBM grants you a nonexclusive copyright license   */
/* to use all programming code examples from which    */
/* you can generate similar function tailored to your */
/* own specific needs.                                */
/*                                                    */
/* All sample code is provided by IBM for illustrative*/
/* purposes only. These examples have not been        */
/* thoroughly tested under all conditions. IBM,       */
/* therefore, cannot guarantee or imply reliability,  */
/* serviceability, or function of these programs.     */
/*                                                    */
/* All programs contained herein are provided to you  */
/* "AS IS" without any warranties of any kind. The    */
/* implied warranties of non-infringement,            */
/* merchantability and fitness for a particular       */
/* purpose are expressly disclaimed.  "               */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:     */
/* CRTBNDC PGM(PROG/GSKSERVa)                         */
/*         SRCFILE(PROG/CSRC)                         */
/*         SRCMBR(GSKSERVa)                           */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define _MULTI_THREADED
#include "pthread.h"
#include "qsoasync.h"
#define Failure 0
#define Success 1
```

```
#define TRUE 1
#define FALSE 0

void *workerThread(void *arg);
/*********************************************************************/
/* Descriptive Name: Master thread will establish a client          */
/* connection and hand processing responsibility                    */
/* to a worker thread.                                              */
/* Note: Due to the thread attribute of this program, spawn() must  */
/* be used to invoke.                                               */
/*********************************************************************/
int main(void)
{
  gsk_handle my_env_handle=NULL;    /* secure environment handle */
  gsk_handle my_session_handle=NULL;    /* secure session handle */

  struct sockaddr_in address;
  int buf_len, on = 1, rc = 0;
  int sd = -1, lsd = -1, al = -1, ioCompPort = -1;
  int successFlag = FALSE;
  char buff[1024];
  pthread_t thr;
  void *status;
  Qso_OverlappedIO_t ioStruct;

  /*******************************************/
  /* Issue all of the command in a do/while    */
  /* loop so that clean up can happen at end   */
  /*******************************************/
  do
  {
    /*******************************************/
    /* Create an I/O completion port for this    */
    /* process.                          */
    /*******************************************/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
      perror("QsoCreateIOCompletionPort() failed");
      break;
    }
    /*******************************************/
    /* Create a worker thread                    */
    /* to process all client requests. The       */
    /* worker thread will wait for client        */
    /* requests to arrive on the I/O completion  */
    /* port just created.                        */
    /*******************************************/
    rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
    if (rc < 0)
    {
      perror("pthread_create() failed");
      break;
    }

    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
    if (rc != GSK_OK)
    {
      printf("gsk_environment_open() failed with rc = %d & errno = %d.\n",
             rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* set the Application ID to use */
    rc = errno = 0;
```

```
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_OS400_APPLICATION_ID,
                              "MY_SERVER_APP",
                              13);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n"
         ,rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* set this side as the server            */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_SESSION_TYPE,
                            GSK_SERVER_SESSION);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_enum() failed with rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
/* We will disable SSL_V2 for this example.           */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_enum() failed with rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* set the cipher suite to use.  By default our default list     */
/* of ciphers is enabled.  For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", /* SSL_RSA_WITH_RC4_128_SHA */
                              2);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n"
         ,rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* Initialize the secure environment */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("gsk_environment_init() failed with rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* initialize a socket to be used for listening */
lsd = socket(AF_INET, SOCK_STREAM, 0);
```

```
if (lsd < 0)
{
  perror("socket() failed");
  break;
}

/* set socket so can be reused immediately */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));
if (rc < 0)
{
  perror("setsockopt() failed");
  break;
}

/* bind to the local server address */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("bind() failed");
  break;
}

/* enable the socket for incoming client connections */
listen(lsd, 5);
if (rc < 0)
{
  perror("listen() failed");
  break;
}

/* accept an incoming client connection */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("accept() failed");
  break;
}

/* open a secure session */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
  printf("gsk_secure_soc_open() failed with rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* associate our socket with the secure session */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
  printf("and errno = %d.\n", errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
```

```
  break;
}

/* initiate the SSL handshake */
rc = errno = 0;
rc = gsk_secure_soc_init(my_session_handle);
if (rc != GSK_OK)
{
  printf("gsk_secure_soc_init() failed with rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/*********************************************/
/* Issue gsk_secure_soc_startRecv() to       */
/* receive client request.                   */
/* Note:                                     */
/* postFlag == on denoting request should    */
/* posted to the I/O completion port, even   */
/* if request is immediately available.      */
/* Worker thread will process client request.*/
/*********************************************/
/*********************************************/
/* initialize Qso_OverlappedIO_t structure - */
/* reserved fields must be hex 00's.         */
/*********************************************/
memset(&ioStruct, '\0', sizeof(ioStruct));
memset((char *) buff, 0, sizeof(buff));
ioStruct.buffer = buff;
ioStruct.bufferLength = sizeof(buff);

/*********************************************/
/* Store the session handle in the           */
/* Qso_OverlappedIO_t descriptorHandle field.*/
/* This area is used to house information    */
/* defining the state of the client          */
/* connection. Field descriptorHandle is     */
/* defined as a (void *) to allow the server */
/* to address more extensive client          */
/* connection state if needed.               */
/*********************************************/
ioStruct.descriptorHandle = my_session_handle;
ioStruct.postFlag = 1;
ioStruct.fillBuffer = 0;

rc = gsk_secure_soc_startRecv(my_session_handle,
                              ioCompPort,
                              &ioStruct);
if (rc != GSK_AS400_ASYNCHRONOUS_RECV)
{
  printf("gsk_secure_soc_startRecv() rc = %d & errno = %d.\n",rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/*********************************************/
/* This is where the server could loop back  */
/* to accept a new connection.               */
/*********************************************/

/*********************************************/
/* Wait for worker thread to finish          */
/* processing client connection.             */
/*********************************************/
rc = pthread_join(thr, &status);
```

```
    /* check status of the worker */
    if ( rc == 0 && (rc = __INT(status)) == Success)
    {
      printf("Success.\n");
      successFlag = TRUE;
    }
    else
    {
      perror("pthread_join() reported failure");
    }
  } while(FALSE);

  /* disable the SSL session */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* disable the SSL environment */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* close the listening socket */
  if (lsd > -1)
    close(lsd);
  /* close the accepted socket */
  if (sd > -1)
    close(sd);

  /* destroy the completion port */
  if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

  if (successFlag)
    exit(0);
  else
    exit(-1);
}


/*********************************************************************/
/* Function Name: workerThread                                       */
/*                                                                   */
/* Descriptive Name: Process client connection.                      */
/*                                                                   */
/* Note:  To make the sample more straight forward the main routine */
/*        handles all of the clean up although this function could  */
/*        be made responsible for the clientfd and session_handle.  */
/*********************************************************************/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort = -1, clientfd = -1;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten;
    gsk_handle client_session_handle = NULL;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;
    /*******************************************/
    /* I/O completion port is passed to this */
    /* routine. */
    /*******************************************/
    ioCompPort = *(int *)arg;
    /*******************************************/
    /* Wait on the supplied I/O completion port */
```

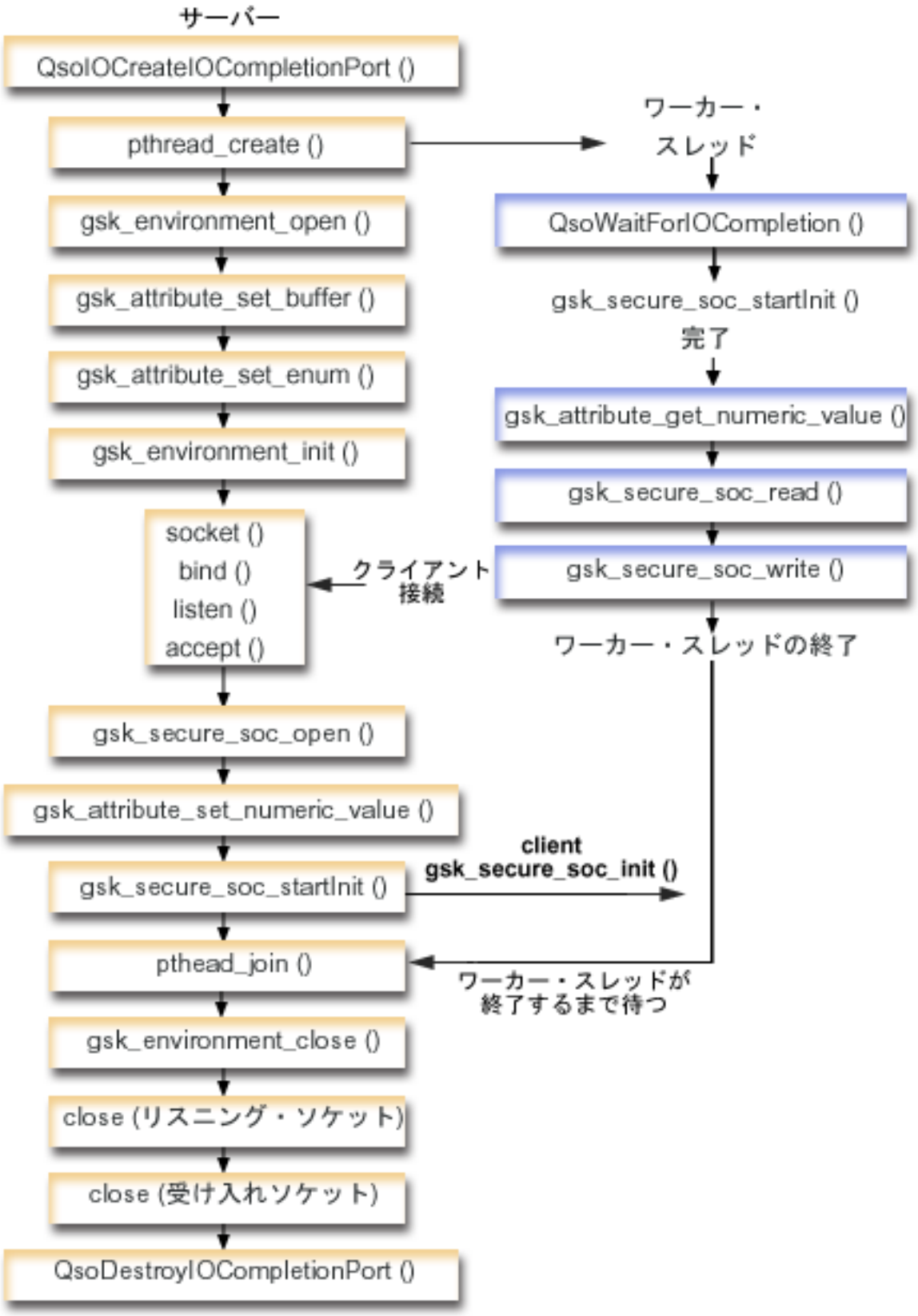
```
/* for a client request. */
/*******************************************/
waitTime.tv_sec = 500;
waitTime.tv_usec = 0;
rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
if ((rc == 1) &&
    (ioStruct.returnValue == GSK_OK) &&
    (ioStruct.operationCompleted == GSKSECURESOCSTARTRECV))
/*******************************************/
/* Client request has been received. */
/*******************************************/
;
else
{
  perror("QsoWaitForIOCompletion()/gsk_secure_soc_startRecv() failed");
  printf("ioStruct.returnValue = %d.\n", ioStruct.returnValue);
  return __VOID(Failure);
}

/* write results to screen */
printf("gsk_secure_soc_startRecv() received %d bytes, here they are:\n",
       ioStruct.secureDataTransferSize);
printf("%s\n",ioStruct.buffer);

/*******************************************/
/* Obtain the session handle associated      */
/* with the client connection.               */
/*******************************************/
client_session_handle = ioStruct.descriptorHandle;

/* get the socket associatied with the secure session */
rc=errno=0;
rc = gsk_attribute_get_numeric_value(client_session_handle,
                                     GSK_FD,
                                     &clientfd);
if (rc != GSK_OK)
{
  printf("gsk_attribute_get_numeric_value() rc = %d & errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  return __VOID(Failure);
}

/* send the message to the client using the secure session */
amtWritten = 0;
rc = gsk_secure_soc_write(client_session_handle,
                          ioStruct.buffer,
                          ioStruct.secureDataTransferSize,
                          &amtWritten);
if (amtWritten != ioStruct.secureDataTransferSize)
{
  if (rc != GSK_OK)
  {
    printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    return __VOID(Failure);
  }
  else
  {
    printf("gsk_secure_soc_write() did not write all data.\n");
    return __VOID(Failure);
  }
}

/* write results to screen */
printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
```

```
    printf("%s\n",ioStruct.buffer);

    return __VOID(Success);
} /* end workerThread */
```

例: 非同期ハンドシェークを使用する GSKit セキュア・サーバー

V5R2 から、OS/400 ソケットは **gsk_secure_soc_startInit()** API を導入しました。この API を使用すると、要求を非同期で処理できるセキュア・サーバー・アプリケーションを作成できます。以下のコード・サンプルに、この API の使用方法の例を示します。この例は、非同期データ受信を使用する GSKit セキュア・サーバーの例に似ていますが、この新しい API を使用してセキュア・セッションを開始します。

以下の図は、セキュア・サーバーで非同期ハンドシェークをネゴシエーションするのに使用される関数呼び出しを示します。


サーバー
QsoIOCreateIOCompletionPort ()
pthread_create ()
ワーカー・
スレッド
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_buffer ()
gsk_secure_soc_startInit ()
完了
gsk_attribute_set_enum ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_read ()
socket ()
bind ()
listen ()
accept ()
クライアント
接続
gsk_secure_soc_write ()
ワーカー・スレッドの終了
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
client
gsk_secure_soc_init ()
gsk_secure_soc_startInit ()
pthead_join ()
ワーカー・スレッドが
終了するまで待つ
gsk_environment_close ()
close (リスニング・ソケット)
close (受け入れソケット)
QsoDestroyIOCompletionPort ()

この図のクライアントの部分を表示するには、GSKit クライアントを参照してください。

ソケットのイベントのフロー: 非同期ハンドシェークを使用する GSKit セキュア・サーバー

このフローは、以下のサンプル・アプリケーションでのソケット呼び出しを示しています。

1. **QsoCreateIOCompletionPort()** 関数が、入出力完了ポートを作成します。
2. **pthread_create** 関数が、すべてのクライアント要求を処理するためのワーカー・スレッドを作成します。ワーカー・スレッドは、ここで作成した入出力完了ポートにクライアント要求が到着するまで待機します。
3. SSL 環境へのハンドルを入手するための **gsk_environment_open()** への呼び出し。
4. SSL 環境の属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、GSK_OS400_APPLICATION_ID 値または GSK_KEYRING_FILE 値を設定するための、**gsk_attribute_set_buffer()** への呼び出し。どちらか一方の値のみを設定します。GSK_OS400_APPLICATION_ID 値を使用することを推奨します。 **gsk_attribute_set_enum()** を使用して、アプリケーション (クライアントまたはサーバー) のタイプ (GSK_SESSION_TYPE) も必ず設定してください。
5. **gsk_environment_init()** への呼び出し。この呼び出しは、SSL を処理するためのこの環境を初期設定し、この環境を使用して実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。
6. **socket** 関数がソケット記述子を作成します。そしてサーバーは、着信接続要求を受け入れることができるよう、標準的なソケット呼び出しのセット (**bind()**、**listen()**、および **accept()**) を発行します。
7. **gsk_secure_soc_open()** 関数が、セキュア・セッション用のストレージを入手し、属性のデフォルト値を設定し、保管してセキュア・セッションに関連した関数呼び出しで使用する必要のあるハンドルを戻します。
8. セキュア・セッションの属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、特定のソケットをこのセキュア・セッションに関連付けるための **gsk_attribute_set_numeric_value()** への呼び出し。
9. **gsk_secure_soc_startInit()** 関数が、SSL 環境およびセキュア・セッションに設定された属性を使用して、セキュア・セッションのネゴシエーションを非同期に開始します。ここで制御がプログラムに戻ります。ハンドシェーク処理が完了すると、完了ポートに結果が通知されます。スレッドはさらに別の処理へと進んでいくことができますが、単純化するため、ワーカー・スレッドが完了するまでここで待機することにします。

   注: 通常は、サーバー・プログラムが SSL ハンドシェークに必要な証明書を提示しないと、通信は成功しません。またサーバーは、サーバー証明書に関連した秘密鍵と、証明書が保管されているキー・データベース・ファイルへアクセスできなければなりません。場合によっては、SSL ハンドシェーク処理中にクライアントも証明書を提示しなければならないこともあります。そうなるのは、クライアントが接続しているサーバーで、クライアント認証が使用可能にされている場合です。 **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) または **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API 呼び出しは、ハンドシェーク中に使用される証明書および秘密鍵の入手先のキー・データベース・ファイルを (それぞれ異なる方法で) 識別します。
10. **pthread_join** が、サーバーとワーカー・プログラムを同期化します。この関数はスレッドが終了するまで待機してから、スレッドを切り離し、スレッド終了状況をサーバーに戻します。
11. **gsk_secure_soc_close()** 関数が、セキュア・セッションを終了します。

12. **gsk_environment_close()** 関数が、SSL 関数をクローズします。
13. **close()** 関数が、listen ソケットを終了します。
14. **close()** が、受け入れた (クライアント接続) ソケットを終了します。
15. **QsoDestroyIOCompletionPort()** 関数が、完了ポートを破棄します。

ソケットのイベントのフロー: セキュア非同期要求を処理するワーカー・スレッド

1. サーバー・アプリケーションがワーカー・スレッドを作成した後、サーバーが処理対象の着信クライアント要求を送ってくるのを待ちます。 **QsoWaitForIOCompletionPort()** 関数は、サーバーによって指定された、提供された入出力完了ポートで待機します。この呼び出しは、**gsk_secure_soc_startInit()** が完了するまで待機します。
2. クライアント要求を受信すると、 **gsk_attribute_get_numeric_value()** 関数が、セキュア・セッションに関連するソケット記述子を取得します。
3. **gsk_secure_soc_read()** 関数が、セキュア・セッションを使用してメッセージをクライアントから受信します。
4. **gsk_secure_soc_write()** 関数が、セキュア・セッションを使用してメッセージをクライアントに送信します。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/* GSK Asyncronous Server Program using Application Id*/
/* and gsk_secure_soc_startInit()                      */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:     */
/* CRTBNDC PGM(MYLIB/GSKSERVSI)                       */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(GSKSERVSI)                          */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define _MULTI_THREADED
#include "pthread.h"
#include "qsoasync.h"
#define Failure 0
#define Success 1
#define TRUE 1
#define FALSE 0

void *workerThread(void *arg);
/*********************************************************************/
/* Descriptive Name: Master thread will establish a client          */
/* connection and hand processing responsibility                    */
/* to a worker thread.                                              */
/* Note: Due to the thread attribute of this program, spawn() must  */
/* be used to invoke.                                               */
/*********************************************************************/
int main(void)
{
```

```
gsk_handle my_env_handle=NULL;    /* secure environment handle */
gsk_handle my_session_handle=NULL;    /* secure session handle */

struct sockaddr_in address;
int buf_len, on = 1, rc = 0;
int sd = -1, lsd = -1, al, ioCompPort = -1;
int successFlag = FALSE;
pthread_t thr;
void *status;
Qso_OverlappedIO_t ioStruct;

/*********************************************/
/* Issue all of the command in a do/while    */
/* loop so that clean up can happen at end   */
/*********************************************/

do
{
  /*********************************************/
  /* Create an I/O completion port for this    */
  /* process.                          */
  /*********************************************/
  if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
  {
    perror("QsoCreateIOCompletionPort() failed");
    break;
  }
  /*********************************************/
  /* Create a worker thread               */
  /* to process all client requests. The    */
  /* worker thread will wait for client       */
  /* requests to arrive on the I/O completion   */
  /* port just created.                  */
  /*********************************************/
  rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
  if (rc < 0)
  {
    perror("pthread_create() failed");
    break;
  }

  /* open a gsk environment */
  rc = errno = 0;
  printf("gsk_environment_open()\n");
  rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
  {
  printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* set the Application ID to use */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_SERVER_APP",
                                13);
  if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n"
           ,rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }
```

```
/* set this side as the server            */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_SESSION_TYPE,
                            GSK_SERVER_SESSION);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
/* We will disable SSL_V2 for this example.           */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                         GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* set the cipher suite to use.  By default our default list     */
/* of ciphers is enabled.  For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", /* SSL_RSA_WITH_RC4_128_SHA */
                              2);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n"
         ,rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* Initialize the secure environment */
rc = errno = 0;
printf("gsk_environment_init()\n");
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* initialize a socket to be used for listening */
printf("socket()\n");
lsd = socket(AF_INET, SOCK_STREAM, 0);
if (lsd < 0)
{
  perror("socket() failed");
  break;
}

/* set socket so can be reused immediately */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
```

```
                sizeof(on));
if (rc < 0)
{
  perror("setsockopt() failed");
  break;
}

/* bind to the local server address */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
printf("bind()\n");
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("bind() failed");
  break;
}

/* enable the socket for incoming client connections */
printf("listen()\n");
listen(lsd, 5);
if (rc < 0)
{
  perror("listen() failed");
  break;
}

/* accept an incoming client connection */
al = sizeof(address);
printf("accept()\n");
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("accept() failed");
  break;
}

/* open a secure session */
rc = errno = 0;
printf("gsk_secure_soc_open()\n");
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
  printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}
/* associate our socket with the secure session */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
  printf("and errno = %d.\n", errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/*******************************************/
/* Issue gsk_secure_soc_startInit() to       */
/* process SSL Handshake flow asyncronously  */
/*******************************************/
```

```
  /*******************************************/
  /* initialize Qso_OverlappedIO_t structure - */
  /* reserved fields must be hex 00's.         */
  /*******************************************/
  memset(&ioStruct, '\0', sizeof(ioStruct));

  /*******************************************/
  /* Store the session handle in the           */
  /* Qso_OverlappedIO_t descriptorHandle field.*/
  /* This area is used to house information    */
  /* defining the state of the client          */
  /* connection. Field descriptorHandle is     */
  /* defined as a (void *) to allow the server */
  /* to address more extensive client          */
  /* connection state if needed.               */
  /*******************************************/
  ioStruct.descriptorHandle = my_session_handle;

  /* initiate the SSL handshake */
  rc = errno = 0;
  printf("gsk_secure_soc_startInit()\n");
  rc = gsk_secure_soc_startInit(my_session_handle, ioCompPort, &ioStruct);
  if (rc != GSK_OS400_ASYNCHRONOUS_SOC_INIT)
  {
    printf("gsk_secure_soc_startInit() rc = %d and errno = %d.\n",rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }
  else
    printf("gsk_secure_soc_startInit got GSK_OS400_ASYNCHRONOUS_SOC_INIT\n");

  /*******************************************/
  /* This is where the server could loop back  */
  /* to accept a new connection.               */
  /*******************************************/

  /*******************************************/
  /* Wait for worker thread to finish          */
  /* processing client connection.             */
  /*******************************************/
  rc = pthread_join(thr, &status);


  /* check status of the worker */
  if ( rc == 0 && (rc = __INT(status)) == Success)
  {
    printf("Success.\n");
    printf("Success.\n");
    successFlag = TRUE;
  }
  else
  {
    perror("pthread_join() reported failure");
  }
} while(FALSE);

/* disable the SSL session */
if (my_session_handle != NULL)
  gsk_secure_soc_close(&my_session_handle);

/* disable the SSL environment */
if (my_env_handle != NULL)
  gsk_environment_close(&my_env_handle);

/* close the listening socket */
if (lsd > -1)
  close(lsd);
```

```
  /* close the accepted socket */
  if (sd > -1)
    close(sd);

  /* destroy the completion port */
  if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

  if (successFlag)
    exit(0);

  exit(-1);
}


/*********************************************************************/
/* Function Name: workerThread                                       */
/*                                                                   */
/* Descriptive Name: Process client connection.                      */
/*                                                                   */
/* Note:  To make the sample more straight forward the main routine */
/*        handles all of the clean up although this function could  */
/*        be made responsible for the clientfd and session_handle.  */
/*********************************************************************/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort, clientfd;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten, amtRead;
    char buff[1024];
    gsk_handle client_session_handle;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;
    /*******************************************/
    /* I/O completion port is passed to this     */
    /* routine.                                */
    /*******************************************/
    ioCompPort = *(int *)arg;
    /*******************************************/
    /* Wait on the supplied I/O completion port  */
    /* for the SSL handshake to complete.        */
    /*******************************************/
    waitTime.tv_sec = 500;
    waitTime.tv_usec = 0;

    sleep(4);
    printf("QsoWaitForIOCompletion()\n");
    rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
    if ((rc == 1) &&
        (ioStruct.returnValue == GSK_OK) &&
        (ioStruct.operationCompleted == GSKSECURESOCSTARTINIT))
    /*******************************************/
    /* SSL Handshake has completed.              */
    /*******************************************/
    ;
    else
    {
      printf("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed.\n");
      printf("rc == %d, returnValue - %d, operationCompleted = %d\n",
             rc, ioStruct.returnValue, ioStruct.operationCompleted);
      perror("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed");
      return __VOID(Failure);
    }
```

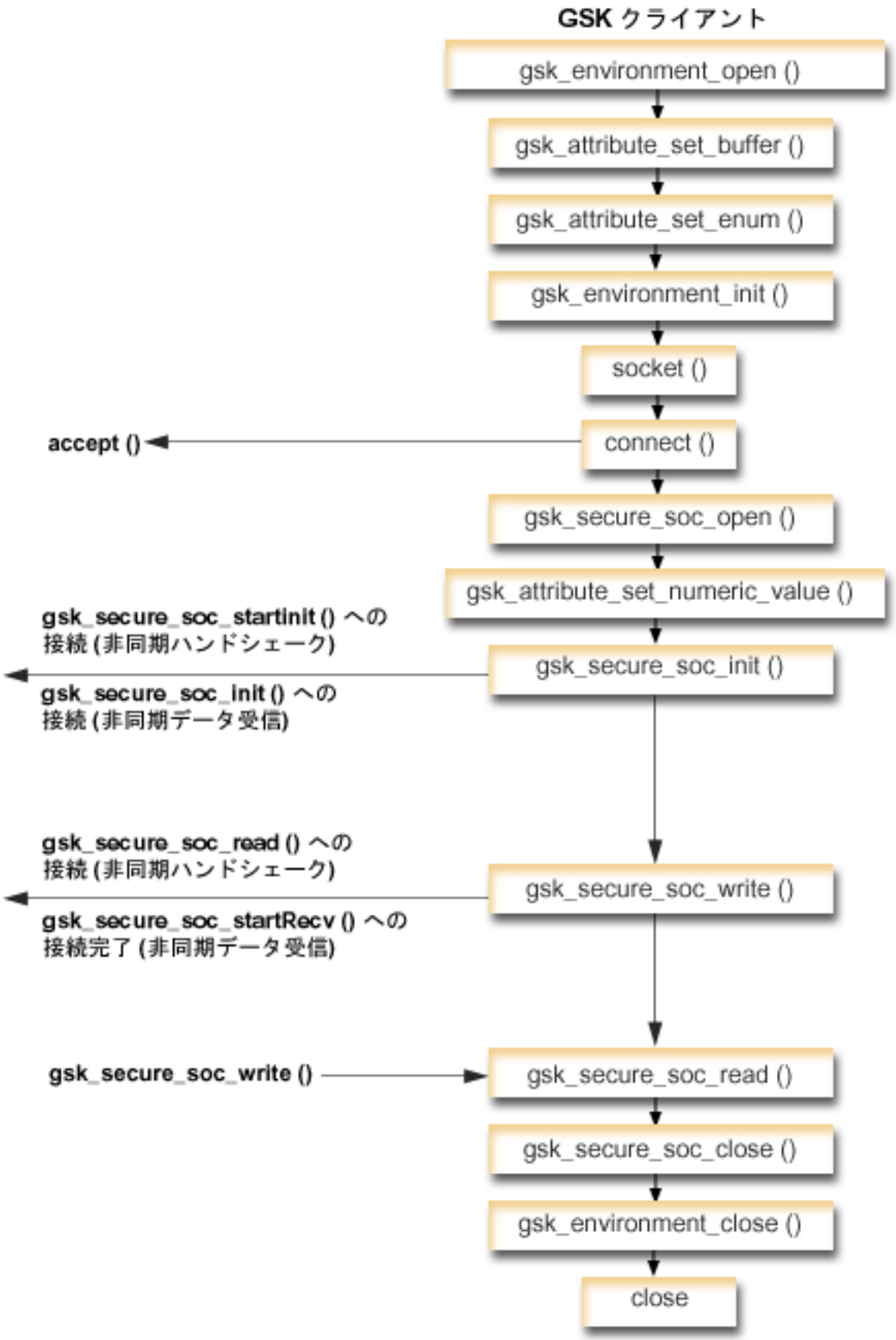
```
/*******************************************/
/* Obtain the session handle associated    */
/* with the client connection.             */
/*******************************************/
client_session_handle = ioStruct.descriptorHandle;

/* get the socket associatied with the secure session */
rc=errno=0;
printf("gsk_attribute_get_numeric_value()\n");
rc = gsk_attribute_get_numeric_value(client_session_handle,
                                     GSK_FD,
                                     &clientfd);
if (rc != GSK_OK)
{
  printf("gsk_attribute_get_numeric_value() rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  return __VOID(Failure);
}
/* memset buffer to hex zeros */
memset((char *) buff, 0, sizeof(buff));
amtRead = 0;
/* receive a message from the client using the secure session */
printf("gsk_secure_soc_read()\n");
rc = gsk_secure_soc_read(client_session_handle,
                         buff,
                         sizeof(buff),
                         &amtRead);

if (rc != GSK_OK)
{
  printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  return;
}

/* write results to screen */
printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
     amtRead);
printf("%s\n",buff);


/* send the message to the client using the secure session */
amtWritten = 0;
printf("gsk_secure_soc_write()\n");
rc = gsk_secure_soc_write(client_session_handle,
                          buff,
                          amtRead,
                          &amtWritten);
if (amtWritten != amtRead)
{
  if (rc != GSK_OK)
  {
    printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    return __VOID(Failure);
  }
  else
  {
    printf("gsk_secure_soc_write() did not write all data.\n");
    return __VOID(Failure);
  }
}
/* write results to screen */
printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
printf("%s\n",buff);
```

```
    return __VOID(Success);
}
/* end workerThread */
```

例: グローバル・セキュア・ツールキット (GSKit) API によってセキュア・クライアントを確立する

以下のコード・サンプルに、GSKit API を使用するクライアントの例を示します。コード例の使用については、『コードの特記事項情報』を参照してください。

以下の図は、GSKit API を使用するセキュア・クライアントの関数呼び出しを示しています。


GSK クライアント
gsk_environment_open ()
gsk_attribute_set_buffer ()
gsk_attribute_set_enum ()
gsk_environment_init ()
socket ()
connect ()
accept ()
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
gsk_secure_soc_startinit () への
接続 (非同期ハンドシェーク)
gsk_secure_soc_init ()
gsk_secure_soc_init () への
接続 (非同期データ受信)
gsk_secure_soc_read () への
接続 (非同期ハンドシェーク)
gsk_secure_soc_write ()
gsk_secure_soc_startRecv () への
接続完了 (非同期データ受信)
gsk_secure_soc_write ()
gsk_secure_soc_read ()
gsk_secure_soc_close ()
gsk_environment_close ()
close

ソケットのイベントのフロー: GSKit クライアント

このフローは、以下のサンプル・アプリケーションでのソケット呼び出しを示しています。このクライアントの例を、GSKit サーバーの例、および『例: 非同期ハンドシェークを使用する GSKit セキュア・サーバー』と一緒に使用してください。

1. **gsk_environment_open()** 関数が、SSL 環境へのハンドルを入手します。
2. SSL 環境の属性を設定するための gsk_attribute_set_xxxxx() への 1 回または複数回の呼び出し。少なくとも、GSK_OS400_APPLICATION_ID 値または GSK_KEYRING_FILE 値を設定するための、**gsk_attribute_set_buffer()** への呼び出し。どちらか一方の値のみを設定します。GSK_OS400_APPLICATION_ID 値を使用することを推奨します。 **gsk_attribute_set_enum()** を使用して、アプリケーション (クライアントまたはサーバー) のタイプ (GSK_SESSION_TYPE) も必ず設定してください。
3. **gsk_environment_init()** への呼び出し。この呼び出しは、SSL を処理するためのこの環境を初期設定し、この環境を使用して実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。
4. **socket** 関数がソケット記述子を作成します。その後、クライアントは、サーバー・アプリケーションに接続するために、**connect()** を発行します。
5. **gsk_secure_soc_open()** 関数が、セキュア・セッション用のストレージを入手し、属性のデフォルト値を設定し、保管してセキュア・セッションに関連した関数呼び出しで使用する必要のあるハンドルを戻します。
6. **gsk_attribute_set_numeric_value()** 関数が、特定のソケットとこのセキュア・セッションを関連付けます。
7. **gsk_secure_soc_init()** 関数が、SSL 環境およびセキュア・セッションに設定された属性を使用して、セキュア・セッションのネゴシエーションを非同期に開始します。
8. **gsk_secure_soc_write()** 関数が、セキュア・セッションのデータをワーカー・スレッドに書き込みます。

   注: GSKit サーバーの例の場合、この関数は、**gsk_secure_soc_startRecv()** 関数を完了したワーカー・スレッドに対してデータを書き込みます。非同期の例の場合は、完了した **gsk_secure_soc_startInit()** に対して書き込みます。
9. **gsk_secure_soc_read()** 関数が、セキュア・セッションを使用してメッセージをワーカー・スレッドから受信します。
10. **gsk_secure_soc_close()** 関数が、セキュア・セッションを終了します。
11. **gsk_environment_close()** 関数が、SSL 関数をクローズします。
12. **close()** 関数が、接続を終了します。

```
/* GSK Client Program using Application Id           */

/* This program assumes that the application id is   */
/* already registered and a certificate has been     */
/* associated with the appliciation id               */
/*                                                   */
/* No parameters, some comments and many hardcoded   */
/* values to keep it short and simple                */

/* use following command to create bound program:    */
/* CRTBNDC PGM(MYLIB/GSKCLIENT)                      */
/*         SRCFILE(MYLIB/CSRC)                       */
/*         SRCMBR(GSKCLIENT)                         */

#include <stdio.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define TRUE 1
#define FALSE 0

void main(void)
{
  gsk_handle my_env_handle=NULL;    /* secure environment handle */
  gsk_handle my_session_handle=NULL;    /* secure session handle */

  struct sockaddr_in address;
  int buf_len, rc = 0, sd = -1;
  int amtWritten, amtRead;
  char buff1[1024];
  char buff2[1024];

  /*  hardcoded IP address (change to make address were server program runs */
  char addr[16] = "1.1.1.1";

  /*********************************************/
  /* Issue all of the command in a do/while    */
  /* loop so that clean up can happen at end   */
  /*********************************************/
  do
  {
    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
    if (rc != GSK_OK)
    {
      printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
             rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* set the Application ID to use */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                  GSK_OS400_APPLICATION_ID,
                                  "MY_CLIENT_APP",
                                  13);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
             rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* set this side as the client (this is the default */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                GSK_SESSION_TYPE,
                                GSK_CLIENT_SESSION);
    if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
             rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }
```

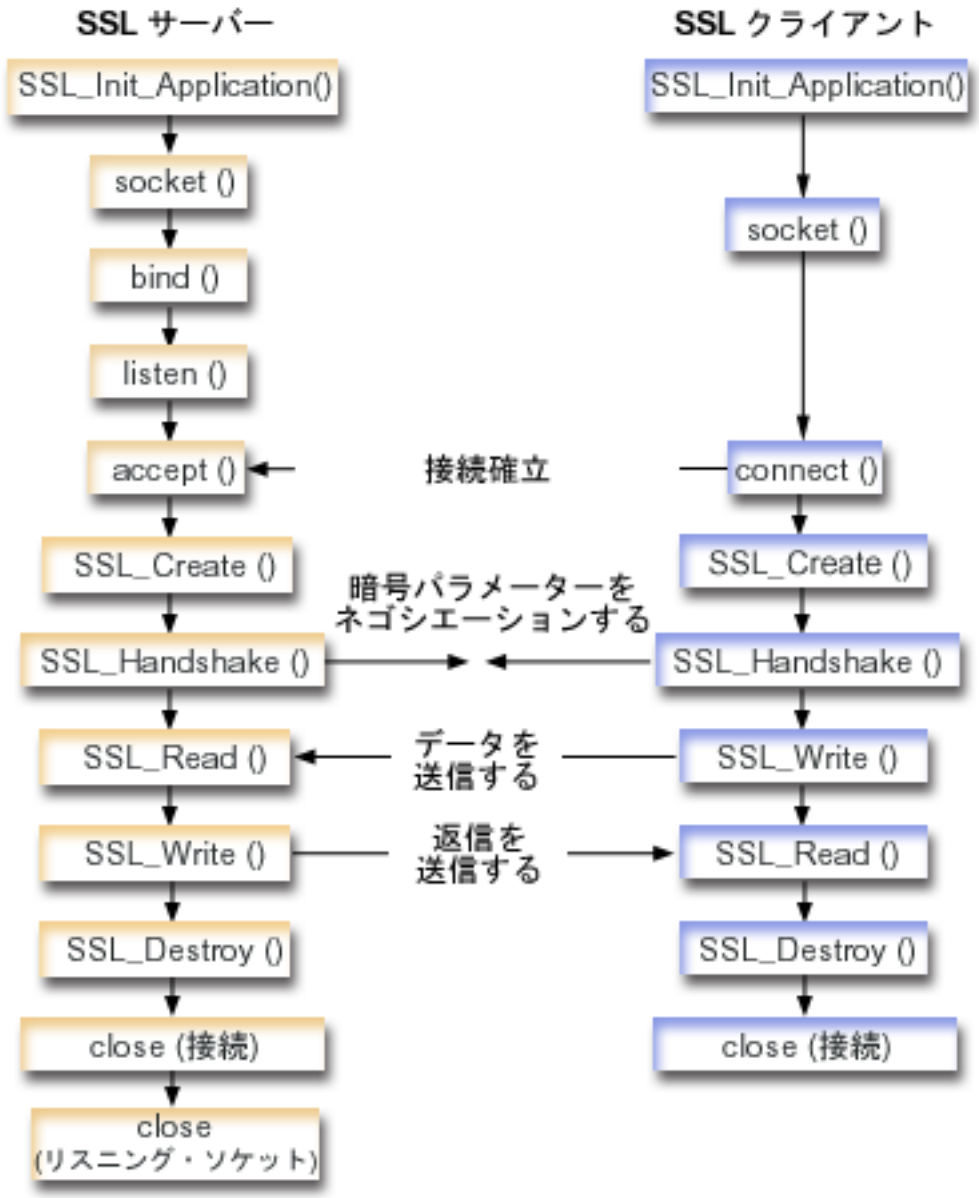
```
/* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
/* We will disable SSL_V2 for this example.           */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* set the cipher suite to use.  By default our default list     */
/* of ciphers is enabled.  For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", 7* SSL_RSA_WITH_RC4_128_SHA */
                              2);
if (rc != GSK_OK)
{
  printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* Initialize the secure environment */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
         rc,errno);
  printf("rc of %d means %s\n", rc, gsk_strerror(rc));
  break;
}

/* initialize a socket to be used for listening */
sd = socket(AF_INET, SOCK_STREAM, 0);
if (sd < 0)
{
  perror("socket() failed");
  break;
}

/* connect to the server using a set port number */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = inet_addr(addr);
rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("connect() failed");
  break;
}

/* open a secure session */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
  printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
         rc,errno);
```

```
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* associate our socket with the secure session */
  rc=errno=0;
  rc = gsk_attribute_set_numeric_value(my_session_handle,
                                       GSK_FD,
                                       sd);
  if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
    printf("and errno = %d.\n", errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* initiate the SSL handshake */
  rc = errno = 0;
  rc = gsk_secure_soc_init(my_session_handle);
  if (rc != GSK_OK)
  {
    printf("gsk_secure_soc_init() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* memset buffer to hex zeros */
  memset((char *) buff1, 0, sizeof(buff1));

  /* send a message to the server using the secure session */
  strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

  /* send the message to the client using the secure session */
  buf_len = strlen(buff1);
  amtWritten = 0;
  rc = gsk_secure_soc_write(my_session_handle, buff1, buf_len, &amtWritten);
  if (amtWritten != buf_len)
  {
    if (rc != GSK_OK)
    {
      printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }
    else
    {
      printf("gsk_secure_soc_write() did not write all data.\n");
      break;
    }
  }

  /* write results to screen */
  printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
  printf("%s\n",buff1);

/* memset buffer to hex zeros */
  memset((char *) buff2, 0x00, sizeof(buff2));

  /* receive a message from the client using the secure session */
  amtRead = 0;
  rc = gsk_secure_soc_read(my_session_handle, buff2, sizeof(buff2), &amtRead);

  if (rc != GSK_OK)
  {
    printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
```

```
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* write results to screen */
    printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
           amtRead);
    printf("%s\n",buff2);


  } while(FALSE);

  /* disable SSL support for the socket */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* disable the SSL environment */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* close the connection */
  if (sd > -1)
    close(sd);

  return;
}
```

例: SSL_ API によってセキュア・サーバーを確立する

セキュア・アプリケーションは、GSKit API だけでなく、SSL_ API を使って作成することもできます。この API は iSeries オペレーティング・システム固有のものです。 GSKit API の場合と同様、データを安全に交換するには、サーバーは有効な証明書を提供する必要があります。以下の図に、セキュア・サーバーを作成するのに使用されるソケットと SSL_ API を示します。複数の IBM @server プラットフォームをまたぐセキュア・アプリケーションを作成する場合には、GSKit API を使用してください。



ソケットのイベントのフロー: SSL_ API を使用するセキュア・サーバー

以下の説明は、SSL サーバーを実行可能にして SSL クライアントと通信する API 同士の関係を示しています。

1. 呼び出し **SSL_Init()** または **SSL_Init_Application()**。この呼び出しは、SSL 処理用のジョブ環境を初期設定し、現行ジョブで実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。どちらか一方の API のみを使用します。 **SSL_Init_Application()** API を使用することを推奨します。

   注: 以下のプログラム例は、**SSL_Init_Application** API を使用しています。
2. サーバーは **socket()** を呼び出してソケット記述子を取得します。

3. サーバーは **bind()**、**listen()**、および **accept()** を呼び出してサーバー・プログラムの接続を活動化します。
4. サーバーは **SSL_Create()** を呼び出して接続されたソケットについて SSL サポートをオンにします。
5. サーバーは **SSL_Handshake()** を呼び出して暗号パラメーターの SSL ハンドシェーク・ネゴシエーションを開始します。
6. サーバーは **SSL_Write()** および **SSL_Read()** を呼び出し、データを送受信します。
7. サーバーは **SSL_Destroy()** を呼び出してソケットの SSL サポートを使用不可にします。
8. サーバーは **close()** を呼び出して接続されているソケットを破棄します。

ソケットのイベントのフロー: SSL_ API を使用するセキュア・クライアント

1. 呼び出し **SSL_Init()** または **SSL_Init_Application()**。この呼び出しは、SSL 処理用のジョブ環境を初期設定し、現行ジョブで実行されるすべての SSL セッション用の SSL セキュリティー情報を設定します。どちらか一方の API のみを使用します。 **SSL_Init_Application** API を使用することを推奨します。

   注: 以下のプログラム例は、**SSL_Init_Application** API を使用しています。
2. クライアントは **socket()** を呼び出してソケット記述子を取得します。
3. クライアントは **connect()** を呼び出してクライアント・プログラムの接続を活動化します。
4. クライアントは **SSL_Create()** を呼び出して接続されたソケットについて SSL サポートをオンにします。
5. クライアントは **SSL_Handshake()** を呼び出して暗号パラメーターの SSL ハンドシェーク・ネゴシエーションを開始します。
6. クライアントは **SSL_Read()** および **SSL_Write()** を呼び出してデータを送受信します。
7. クライアントは **SSL_Destroy()** を呼び出してソケットの SSL サポートを使用不可にします。
8. クライアントは **close()** を呼び出して接続されているソケットを破棄します。

注: このサンプルでは AF_INET アドレス・ファミリーを使用します。しかし、AF_INET6 アドレス・ファミリーを使用するように変更することもできます。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/* SSL Server Program using SSL_Init_Application      */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:     */
/* CRTBNDC PGM(MYLIB/SSLSERVAPP)                      */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(SSLSERVAPP)                         */

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

void main(void)
{
```

```
SSLHandle *sslh;
SSLInitApp sslinit;

struct sockaddr_in address;
int buf_len, on = 1, rc = 0, sd, lsd, al;
char buff[1024];

/*  only want to use 1 cipher suite     */
unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

void * malloc_ptr = (void *) NULL;
unsigned int malloc_size = 8192;

/* memset sslinitapp structure to hex zeros */
memset((char *)&sslinit, 0, sizeof(sslinit));

/* fill in values for sslinit app structure */
sslinit.applicationID = "MY_SERVER_APP";
sslinit.applicationIDLen = 13;
sslinit.localCertificate = NULL;
sslinit.localCertificateLen = 0;
sslinit.cipherSuiteList = NULL;
sslinit.cipherSuiteListLen = 0;

/* allocate and set pointers for certificate buffer */
malloc_ptr = (void*) malloc(malloc_size);
sslinit.localCertificate = (unsigned char*) malloc_ptr;
sslinit.localCertificateLen = malloc_size;

/* initialize ssl call SSL_Init_Application */
rc = SSL_Init_Application(&sslinit);
if (rc != 0)
{
  printf("SSL_Init_Application() failed with rc = %d and errno = %d.\n",
         rc,errno);
  return;
}

/* initialize a socket to be used for listening */
lsd = socket(AF_INET, SOCK_STREAM, 0);
if (lsd < 0)
{
  perror("socket() failed");
  return;
}

/* set socket so can be reused immediately */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));
if (rc < 0)
{
  perror("setsockopt() failed");
  return;
}

/* bind to the local server address */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("bind() failed");
  close(lsd);
```

```
  return;
}

/* enable the socket for incoming client connections */
listen(lsd, 5);
              if (rc < 0)
{
  perror("listen() failed");
  close(lsd);
  return;
}

/* accept an incoming client connection */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("accept() failed");
  close(lsd);
  return;
}

/* enable SSL support for the socket */
sslh = SSL_Create(sd, SSL_ENCRYPT);
if (sslh == NULL)
{
  printf("SSL_Create() failed with errno = %d.\n", errno);
  close(lsd);
  close(sd);
  return;
}

/* set up parameters for handshake */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* initiate the SSL handshake */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_SERVER);
if (rc != 0)
{
  printf("SSL_Handshake() failed with rc = %d and errno = %d.\n",
      rc, errno);
  SSL_Destroy(sslh);
  close(lsd);
  close(sd);
  return;
}

/* memset buffer to hex zeros */
memset((char *) buff, 0, sizeof(buff));

/* receive a message from the client using the secure session */
rc = SSL_Read(sslh, buff, sizeof(buff));
if (rc < 0)
{
  printf("SSL_Read() rc = %d and errno = %d.\n",rc,errno);
  rc = SSL_Destroy(sslh);
  if (rc != 0)
    printf("SSL_Destroy() rc = %d and errno = %d.\n",rc,errno);
  close(lsd);
  close(sd);
  return;
}

/* write results to screen */
```

```
    printf("SSL_Read() read ...\n");
    printf("%s\n",buff);

    /* send the message to the client using the secure session */
    buf_len = strlen(buff);
    rc = SSL_Write(sslh, buff, buf_len);
    if (rc != buf_len)
    {
      if (rc < 0)
      {
        printf("SSL_Write() failed with rc = %d.\n",rc);
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
      }
      else
      {
        printf("SSL_Write() did not write all data.\n");
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
      }
    }

    /* write results to screen */
    printf("SSL_Write() wrote ...\n");
    printf("%s\n",buff);

    /* disable SSL support for the socket */
    SSL_Destroy(sslh);

    /* close the connection */
    close(sd);

    /* close the listening socket */
    close(lsd);

    return;
}
```

例: SSL_ API によってセキュア・クライアントを確立する

OS/400 ソケットは、GSKit API だけでなく、従来の SSL_ API もサポートします。これらの API は、iSeries 固有のサーバーとクライアント・アプリケーションとの間で、セキュア接続を確立します。このプログラムおよび対応するサーバー・アプリケーションのソケットのイベント・フローについて説明する図が、『例: SSL_ API によってセキュア・サーバーを確立する』に記載されています。以下の例では、SSL_ API を使用するクライアント・アプリケーションが、SSL_API を使用するサーバー・アプリケーションと通信できるようにします。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/* SSL Client Program using SSL_Init_Application      */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:     */
/* CRTBNDC PGM(MYLIB/SSLCLIAPP)                       */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(SSLCLIAPP                           */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <ssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

/*  Making this simple - no parameters  */
void main(void)
{
    SSLHandle *sslh;
    SSLInitApp sslinit;
    struct sockaddr_in address;
    int buf_len, rc = 0, sd;
    char buff1[1024];
    char buff2[1024];

    /*  only want to use 1 cipher suite     */
    unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

    /*  hardcoded IP address                */
    char addr[12] = "16.35.146.84";

    void * malloc_ptr = (void *) NULL;
    unsigned int malloc_size = 8192;

    /* memset sslinit structure to hex zeros */
    memset((char *)&sslinit, 0, sizeof(sslinit));

    /* fill in values for sslinitapp structure */
    /* using an existing app id                */
    sslinit.applicationID = "MY_CLIENT_APP";
    sslinit.applicationIDLen = 13;
    sslinit.localCertificate = NULL;
    sslinit.localCertificateLen = 0;
    sslinit.cipherSuiteList = NULL;
    sslinit.cipherSuiteListLen = 0;

    /* allocate and set pointers for certificate buffer */
    malloc_ptr = (void*) malloc(malloc_size);
    sslinit.localCertificate = (unsigned char*) malloc_ptr;
    sslinit.localCertificateLen = malloc_size;

    /* initialize ssl call SSL_Init_Application */
    rc = SSL_Init_Application(&sslinit);
    if (rc != 0)
    {
      printf("SSL_Init_Application() failed with rc = %d and errno = %d.\n",
           rc,errno);
      return;
    }

    /* initialize a socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
    {
      perror("socket() failed");
      return;
    }

    /* enable SSL support for the socket */
    sslh = SSL_Create(sd, SSL_ENCRYPT);
```

```
if (sslh == NULL)
{
  printf("SSL_Create() failed with errno = %d.\n", errno);
  close(sd);
  return;
}

/* connect to the server using a set port number */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = inet_addr(addr);
rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("connect() failed");
  close(sd);
  return;
}

/* set up to call handshake, setting cipher     */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* initiate the SSL handshake - as a CLIENT */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_CLIENT);
if (rc != 0)
{
  printf("SSL_Handshake() failed with rc = %d and errno = %d.\n",
       rc, errno);
  close(sd);
  return;
}

/* send a message to the server using the secure session */
strcpy(buff1,"Test of SSL_Write \n\n");
buf_len = strlen(buff1);
rc = SSL_Write(sslh, buff1, buf_len);
if (rc != buf_len)
{
  if (rc < 0)
  {
   printf("SSL_Write() failed with rc = %d and errno = %d.\n",rc,errno);
    SSL_Destroy(sslh);
    close(sd);
    return;
  }
  else
  {
    printf("SSL_Write() did not write all data.\n");
    SSL_Destroy(sslh);
    close(sd);
    return;
  }
}

/* write the results to the screen */
printf("SSL_Write() wrote ...\n");
printf("%s\n",buff1);

memset((char *) buff2, 0x00, sizeof(buff2));

/* receive the message from the server using the secure session */
rc = SSL_Read(sslh, buff2, buf_len);
if (rc < 0)
```

```
    {
      printf("SSL_Read() failed with rc = %d.\n",rc);
      SSL_Destroy(sslh);
      close(sd);
      return;
    }

    /* write the results to the screen */
    printf("SSL_Read() read ...\n");
    printf("%s\n",buff2);

    /* disable SSL support for the socket */
    SSL_Destroy(sslh);

    /* close the connection by closing the local socket */
    close(sd);
    return;
}
```

例: gethostbyaddr_r() を使用したスレッド・セーフ・ネットワーク・ルーチン

以下は、**gethostbyaddr_r()** を使用するプログラムの例です。名前の末尾に _r が付いている他のすべてのルーチンも類似したセマンティクスを持ち、スレッド・セーフです。このプログラム例では、IP アドレスをドット 10 進表記で表し、ホスト名を印刷します。

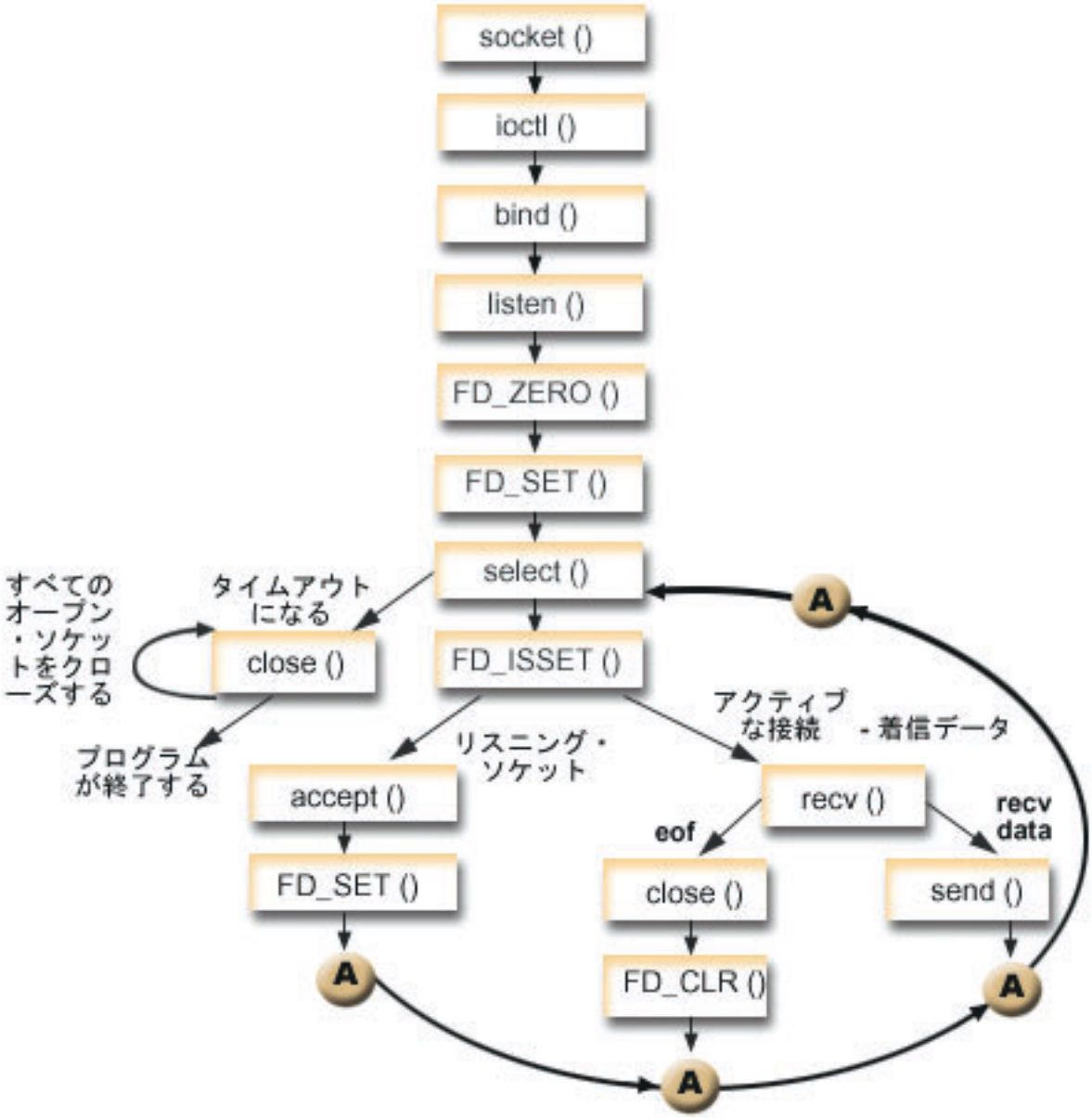
コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************/
/* Header files                                          */
/*********************************************************/
#include </netdb.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define HEX00 '\x00'
#define NUMPARMS 2
/*********************************************************/
/* Pass one parameter that is the IP address in          */
/* dotted decimal notation. The host name will be        */
/* displayed if found; otherwise, a message states       */
/* host not found.                                       */
/*********************************************************/
int main(int argc, char *argv[])
    {
    int rc;
    struct in_addr internet_address;
    struct hostent hst_ent;
    struct hostent_data  hst_ent_data;
    char dotted_decimal_address [16];
    char host_name[MAXHOSTNAMELEN];

    /*********************************************************/
    /* Verify correct number of arguments have been passed   */
    /*********************************************************/
    if (argc != NUMPARMS)
     {
      printf("Wrong number of parms passed\n");
      exit(-1);
     }
    /*********************************************************/
```

```
    /* Obtain addressability to parameters passed             */
    /*********************************************************/
    strcpy(dotted_decimal_address, argv[1]);

    /*********************************************************/
    /* Initialize the structure-field                        */
    /* hostent_data.host_control_blk with hexadecimal zeros  */
    /* before its initial use. If you require compatibility  */
    /* with other platforms, then you must initialize the    */
    /* entire hostent_data structure with hexadecimal zeros. */
    /*********************************************************/
    /* Initialize to hex 00 hostent_data structure           */
    /*********************************************************/
    memset(&hst_ent_data,HEX00,sizeof(struct hostent_data));

    /*********************************************************/
    /* Translate an Internet address from dotted decimal     */
    /* notation to 32-bit IP address format.                 */
    /*********************************************************/
    internet_address.s_addr=inet_addr(dotted_decimal_address);

    /*********************************************************/
    /* Obtain host name                                      */
    /*********************************************************/
    /*********************************************************/
    /* NOTE: The gethostbyaddr_r() returns an integer.       */
    /* The following are possible values:                    */
    /*   -1 (unsuccessful call)                              */
    /*   0 (successful call)                                 */
    /*********************************************************/
    rc=gethostbyaddr_r((char *) &internet_address,
                       sizeof(struct in_addr), AF_INET,
                       &hst_ent, &hst_ent_data);

    if (rc== -1)
{
         printf("Host name not found\n");
         exit(-1);
       }
    else
       {
        /*****************************************************/
        /*   Copy the host name to an output buffer          */
        /*****************************************************/
          (void) memcpy((void *) host_name,
        /*****************************************************/
        /* You must address all the results through the      */
        /* hostent structure hst_ent.                        */
        /* NOTE: Hostent_data structure hst_ent_data is just */
        /* a data repository that is used to support the     */
        /* hostent structure. Applications should consider   */
        /* hostent_data a storage area to put host level data */
        /* that the application does not need to access.     */
        /*****************************************************/
                       (void *) hst_ent.h_name,
                       MAXHOSTNAMELEN);
        /*****************************************************/
        /* Print the host name                               */
        /*****************************************************/
          printf("The host name is %s\n", host_name);

         }
       exit(0);
     }
```

例: 非ブロッキング入出力および select()

以下のサンプル・プログラムは、非ブロッキングと select() API を使用しています。この例で使用できる共通クライアント・ジョブのコードを含む例については、『例: 汎用クライアント』を参照してください。



ソケットのイベントのフロー: 非ブロッキング入出力および select() を使用するサーバー

この例で使用される呼び出しは、以下のとおりです。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_STREAM) を使用することも示します。
2. **ioctl()** 関数により、必要な待ち時間が満了する前にサーバーを再始動した場合に、ローカル・アドレスを再利用できるようになります。この例では、ソケットを非ブロッキングに設定します。また着信接続のすべてのソケットは listen ソケットからの状態を継承するので、非ブロッキングになります。
3. ソケット記述子が作成された後、**bind()** 関数がソケットの固有名を取得します。

4. **listen()** により、サーバーが着信クライアント接続を受け入れられるようになります。
5. サーバーは、着信接続要求を受け入れるために **accept()** 関数を使用します。 **accept()** 呼び出しは、着信接続の成功を待機して、無期限にブロックします。
6. **select()** 関数により、プロセスがイベントの発生を待機して、イベントが発生するとウェイクアップするようになります。この例の場合、**select()** 関数は、処理の準備の整ったソケット記述子を表す数値を戻します。

   0 プロセスがタイムアウトになることを表します。この例では、タイムアウトが 30 秒に設定されています。

   -1 処理が失敗したことを表します。

   1 処理の準備の整った記述子が 1 つだけあることを表します。この例では、1 が戻されると、FD_ISSET と後続のソケット呼び出しが一度だけ実行されます。

   n 処理を待っている記述子が複数あることを表します。この例では、n が戻されると、FD_ISSET と後続のコードがループし、サーバーが受信した順番に従って要求を処理します。
7. EWOULDBLOCK が戻されると、**accept()** と **recv()** 関数が完了します。
8. **send()** 関数が、クライアントにデータを送り返します。
9. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT  12345

#define TRUE             1
#define FALSE            0

main (int argc, char *argv[])
{
   int    i, len, rc, on = 1;
   int    listen_sd, max_sd, new_sd;
   int    desc_ready, end_server = FALSE;
   int    close_conn;
   char   buffer[80];
   struct sockaddr_in   addr;
   struct timeval       timeout;
   struct fd_set        master_set, working_set;

   /*************************************************************/
   /* Create an AF_INET stream socket to receive incoming       */
   /* connections on                                            */
   /*************************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("socket() failed");
      exit(-1);
   }

   /*************************************************************/
   /* Allow socket descriptor to be reuseable                   */
   /*************************************************************/
   rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
```

```
                (char *)&on, sizeof(on));
if (rc < 0)
{
   perror("setsockopt() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Set socket to be non-blocking.  All of the sockets for    */
/* the incoming connections will also be non-blocking since  */
/* they will inherit that state from the listening socket.   */
/*************************************************************/
rc = ioctl(listen_sd, FIONBIO, (char *)&on);
if (rc < 0)
{
   perror("ioctl() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Bind the socket                                           */
/*************************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port        = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Set the listen back log                                   */
/*************************************************************/
rc = listen(listen_sd, 32);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Initialize the master fd_set                              */
/*************************************************************/
FD_ZERO(&master_set);
max_sd = listen_sd;
FD_SET(listen_sd, &master_set);

/*************************************************************/
/* Initialize the timeval struct to 3 minutes.  If no        */
/* activity after 3 minutes this program will end.           */
/*************************************************************/
timeout.tv_sec  = 3 * 60;
timeout.tv_usec = 0;

/*************************************************************/
/* Loop waiting for incoming connects or for incoming data   */
/* on any of the connected sockets.                          */
/*************************************************************/
   do
```

```
{
   /*************************************************************/
   /* Copy the master fd_set over to the working fd_set.     */
   /*************************************************************/
   memcpy(&working_set, &master_set, sizeof(master_set));

   /*************************************************************/
   /* Call select() and wait 5 minutes for it to complete.   */
   /*************************************************************/
   printf("Waiting on select()...\n");
   rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

   /*************************************************************/
   /* Check to see if the select call failed.                */
   /*************************************************************/
   if (rc < 0)
   {
      perror("  select() failed");
      break;
   }

   /*************************************************************/
   /* Check to see if the 5 minute time out expired.         */
   /*************************************************************/
   if (rc == 0)
   {
      printf("  select() timed out.  End program.\n");
      break;
   }

   /*************************************************************/
   /* One or more descriptors are readable.  Need to         */
   /* determine which ones they are.                         */
   /*************************************************************/
   desc_ready = rc;
   for (i=0; i <= max_sd  &&  desc_ready > 0; ++i)
   {
      /*********************************************************/
      /* Check to see if this descriptor is ready            */
      /*********************************************************/
      if (FD_ISSET(i, &working_set))
      {
         /******************************************************/
         /* A descriptor was found that was readable - one   */
         /* less has to be looked for.  This is being done   */
         /* so that we can stop looking at the working set   */
         /* once we have found all of the descriptors that   */
         /* were ready.                                      */
         /******************************************************/
         desc_ready -= 1;

         /******************************************************/
         /* Check to see if this is the listening socket     */
         /******************************************************/
         if (i == listen_sd)
         {
            printf("  Listening socket is readable\n");
            /***************************************************/
            /* Accept all incoming connections that are      */
            /* queued up on the listening socket before we   */
            /* loop back and call select again.              */
            /***************************************************/
            do
            {
               /************************************************/
               /* Accept each incoming connection.  If       */
               /* accept fails with EWOULDBLOCK, then we     */
```

```
      /* have accepted all of them.  Any other      */
      /* failure on accept will cause us to end the */
      /* server.                                    */
      /**********************************************/
      new_sd = accept(listen_sd, NULL, NULL);
      if (new_sd < 0)
      {
         if (errno != EWOULDBLOCK)
         {
            perror("  accept() failed");
            end_server = TRUE;
         }
         break;
      }

      /**********************************************/
      /* Add the new incoming connection to the     */
      /* master read set                            */
      /**********************************************/
      printf("  New incoming connection - %d\n", new_sd);
      FD_SET(new_sd, &master_set);
      if (new_sd > max_sd)
         max_sd = new_sd;

      /**********************************************/
      /* Loop back up and accept another incoming   */
      /* connection                                 */
      /**********************************************/
   } while (new_sd != -1);
}

/*****************************************************/
/* This is not the listening socket, therefore an    */
/* existing connection must be readable              */
/*****************************************************/
else
{
   printf("  Descriptor %d is readable\n", i);
   close_conn = FALSE;
   /*************************************************/
   /* Receive all incoming data on this socket      */
   /* before we loop back and call select again.    */
   /*************************************************/
   do
   {
      /**********************************************/
      /* Receive data on this connection until the  */
      /* recv fails with EWOULDBLOCK.  If any other */
      /* failure occurs, we will close the          */
      /* connection.                                */
      /**********************************************/
      rc = recv(i, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         if (errno != EWOULDBLOCK)
         {
            perror("  recv() failed");
            close_conn = TRUE;
         }
         break;
      }

      /**********************************************/
      /* Check to see if the connection has been    */
      /* closed by the client                       */
      /**********************************************/
      if (rc == 0)
```

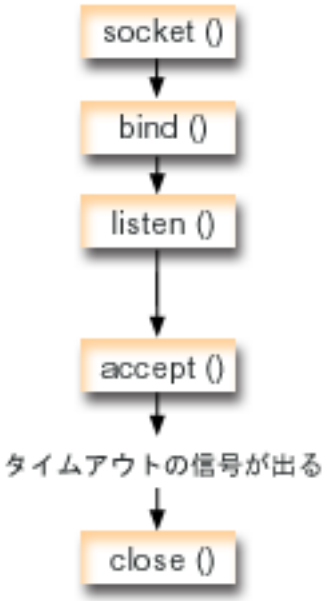
```
                {
                   printf("  Connection closed\n");
                   close_conn = TRUE;
                   break;
                }

                /*********************************************/
                /* Data was recevied                         */
                /*********************************************/
                len = rc;
                printf("  %d bytes received\n", len);

                /*********************************************/
                /* Echo the data back to the client          */
                /*********************************************/
                rc = send(i, buffer, len, 0);
                if (rc < 0)
                {
                   perror("  send() failed");
                   close_conn = TRUE;
                   break;
                }

             } while (TRUE);

             /***************************************************/
             /* If the close_conn flag was turned on, we need */
             /* to clean up this active connection.  This     */
             /* clean up process includes removing the        */
             /* descriptor from the master set and            */
             /* determining the new maximum descriptor value  */
             /* based on the bits that are still turned on in */
             /* the master set.                               */
             /***************************************************/
             if (close_conn)
             {
                close(i);
                FD_CLR(i, &master_set);
                if (i == max_sd)
                {
                   while (FD_ISSET(max_sd, &master_set) == FALSE)
                      max_sd -= 1;
                }
             }
          } /* End of existing connection is readable */
       } /* End of if (FD_ISSET(i, &working_set)) */
    } /* End of loop through selectable descriptors */

 } while (end_server == FALSE);

 /*************************************************************/
 /* Cleanup all of the sockets that are open                  */
 /*************************************************************/
 for (i=0; i <= max_sd; ++i)
 {
    if (FD_ISSET(i, &master_set))
       close(i);
 }
}
```

例: ブロック化ソケット API での信号の使用

信号を使用すると、プロセスまたはアプリケーションがブロックされると通知が出されるようにできます。信号により、ブロック処理に制限時間が設けられます。この例の場合、信号は、**accept()** 呼び出しの 5 秒後に発生します。通常なら、この呼び出しは無期限にブロックしますが、アラームを設定したため、5 秒間

だけブロックすることになります。ブロックされたプログラムはアプリケーションまたはサーバーのパフォーマンスを低下させる恐れがあるので、信号を使ってこの影響を少なくすることができます。以下の例では、信号をブロック化ソケット API で使用する方法を示します。

注: スレッド化されたサーバー・モデルで使用される非同期入出力を、一般的なモデルよりも推奨します。非同期入出力を使用する利点の詳細については、『非同期入出力』を参照してください。非同期入出力 API を使用するサンプル・プログラムについては、『例: 非同期入出力 API の使用』を参照してください。



ソケットのイベントのフロー: ブロック化ソケットでの信号の使用
以下の関数呼び出しのシーケンスは、ソケットが非活動状態になったら、信号を使用してアプリケーションにアラートする方法を示します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと UDP トランスポート (SOCK_DGRAM) を使用することも示します。
2. ソケット記述子が作成された後、**bind()** 関数が、ソケットの固有名を取得します。この例の場合、クライアント・アプリケーションはこのソケットに接続しないため、ポート番号は指定されません。このコードの断片は、**accept()** のようなブロック化 API を使用する、他のサーバー・プログラムで使用できます。
3. **listen()** 関数は、クライアントの接続要求を受け入れる態勢を示しています。 **listen()** 関数が発行された後、アラームが 5 秒で止まるよう設定されます。このアラームまたは信号は、**accept()** 呼び出しのブロックが発生すると、警告を出します。
4. **accept()** 関数は、クライアント接続要求を受け入れます。通常なら、この呼び出しは無期限にブロックしますが、アラームを設定したため、5 秒間だけブロックすることになります。アラームが止まると、accept 呼び出しは -1 を戻して終了し、EINTR という errno 値を戻します。
5. **close()** 関数が、オープンしているソケット記述子をすべて終了させます。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************************************/
/* Example shows how to set alarms for blocking socket APIs       */
/*********************************************************************/

/*********************************************************************/
/* Include files                                                     */
/*********************************************************************/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*********************************************************************/
/* Signal catcher routine.  This routine will be called when the  */
/* signal occurs.                                                    */
/*********************************************************************/
void catcher(int sig)
{
   printf("   Signal catcher called for signal %d\n", sig);
}

/*********************************************************************/
/* Main program                                                      */
/*********************************************************************/
int main(int argc, char *argv[])
{
   struct sigaction sact;
   struct sockaddr_in addr;
   time_t t;
   int sd, rc;

/*********************************************************************/
/* Create an AF_INET, SOCK_STREAM socket                          */
/*********************************************************************/
   printf("Create a TCP socket\n");
   sd = socket(AF_INET, SOCK_STREAM, 0);
   if (sd == -1)
   {
      perror("   socket failed");
      return(-1);
   }

/*********************************************************************/
/* Bind the socket.  A port number was not specified because      */
/* we are not going to ever connect to this socket.               */
/*********************************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   printf("Bind the socket\n");
   rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
   if (rc != 0)
   {
      perror("   bind failed");
      close(sd);
      return(-2);
   }

/*********************************************************************/
/* Perform a listen on the socket.                                */
/*********************************************************************/
   printf("Set the listen backlog\n");
   rc = listen(sd, 5);
   if (rc != 0)
   {
```

```
      perror("   listen failed");
      close(sd);
      return(-3);
   }

/*********************************************************************/
/* Set up an alarm that will go off in 5 seconds.                    */
/*********************************************************************/
   printf("\nSet an alarm to go off in 5 seconds. This alarm will cause the\n");
   printf("blocked accept() to return a -1 and an errno value of EINTR.\n\n");
   sigemptyset(&sact.sa_mask);
   sact.sa_flags = 0;
   sact.sa_handler = catcher;
   sigaction(SIGALRM, &sact, NULL);
   alarm(5);

/*********************************************************************/
/* Display the current time when the alarm was set                   */
/*********************************************************************/
   time(&t);
   printf("Before accept(), time is %s", ctime(&t));

/*********************************************************************/
/* Call accept.  This call will normally block indefinitely,         */
/* but because we have an alarm set, it will only block for          */
/* 5 seconds.  When the alarm goes off, the accept call will         */
/* complete with -1 and an errno value of EINTR.                     */
/*********************************************************************/
   errno = 0;
   printf("   Wait for an incoming connection to arrive\n");
   rc = accept(sd, NULL, NULL);
   printf("   accept() completed.  rc = %d, errno = %d\n", rc, errno);
   if (rc >= 0)
   {
      printf("   Incoming connection was recevied\n");
      close(rc);
   }
   else
   {
      perror("   errno string");
   }

/*********************************************************************/
/* Show what time it was when the alarm went off                     */
/*********************************************************************/
   time(&t);
   printf("After accept(), time is %s\n", ctime(&t));
   close(sd);
   return(0);
}
```

例: マルチキャスティングの使用

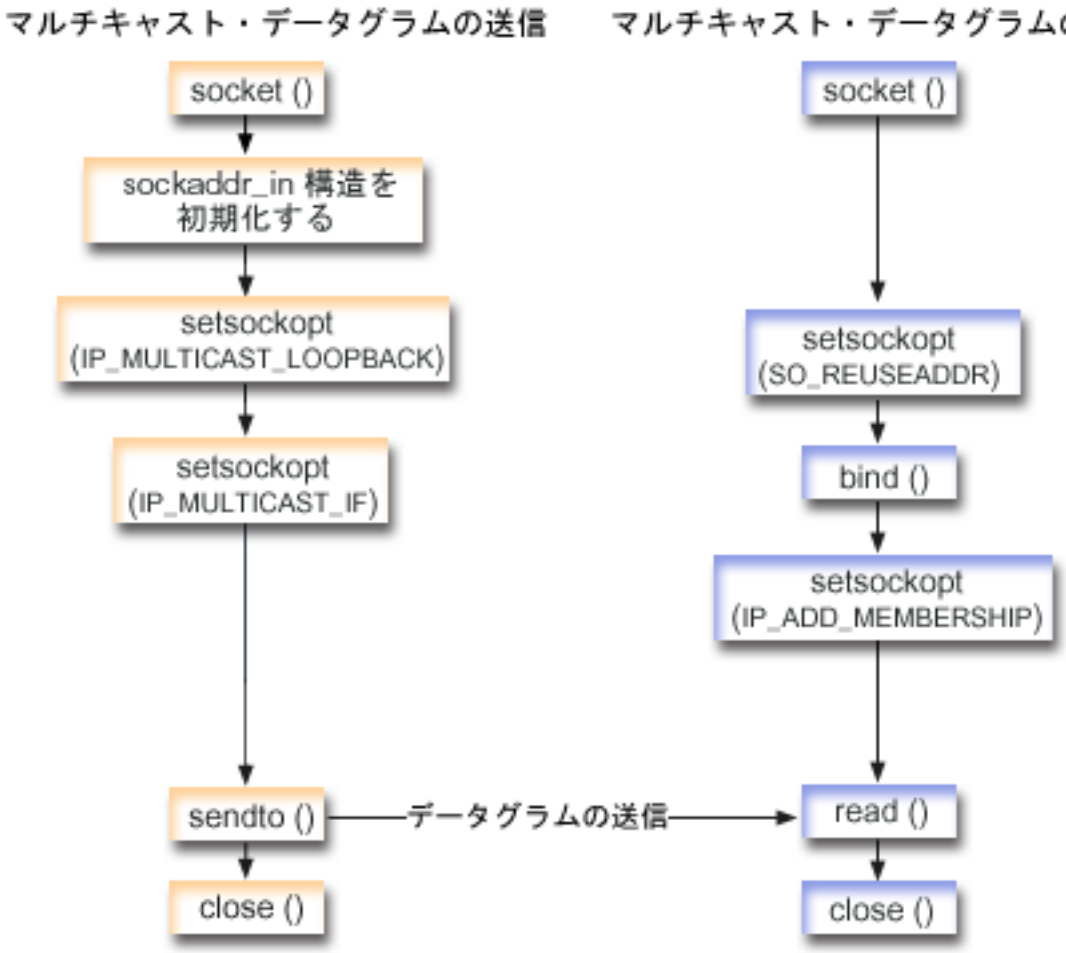
IP マルチキャスティングは、ネットワークにあるホストのグループが受信できるようにアプリケーションが単一の IP データグラムを送信する機能を提供します。グループにあるホストは、単一のサブネットに常駐する場合も、マルチキャスト機能のあるルーターに接続する異なるサブネットに位置する場合もあります。ホストはいつでもグループに結合したり分離したりできます。ホスト・グループでのメンバーの位置や数については、制限はありません。 224.0.0.1 から 239.255.255.255 の範囲のクラス D の IP アドレスは、ホスト・グループを識別します。

アプリケーション・プログラムは、**socket()** API およびコネクションレス型 SOCK_DGRAM タイプ・ソケットを使用することにより、マルチキャスト・データグラムを送受信できます。マルチキャストは、1 対多の伝送方式です。マルチキャストには、タイプ SOCK_STREAM のコネクション型ソケットを使用するこ

とはできません。タイプ SOCK_DGRAM のソケットが作成されると、アプリケーションは **setsockopt()** 関数を使用して、このソケットに関連するマルチキャスト特性を制御することができます。 setsockopt() 関数は、以下の IPPROTO_IP レベル・フラグを受け取ります。

- IP_ADD_MEMBERSHIP: 指定されたマルチキャスト・グループを結合させます。
- IP_DROP_MEMBERSHIP: 指定されたマルチキャスト・グループを外れます。
- IP_MULTICAST_IF: 発信マルチキャスト・データグラムが送信されるインターフェースを設定します。
- IP_MULTICAST_TTL: 発信マルチキャスト・データグラムについて IP ヘッダーの存続時間 (TTL) を設定します。
- IP_MULTICAST_LOOP: 発信マルチキャスト・データグラムのコピーがマルチキャスト・グループのメンバーであるかぎり、送信しているホストに送達されるようにするかどうかを指定します。

注: OS/400 ソケットは、AF_INET アドレス・ファミリーの IP マルチキャスティングをサポートします。



ソケットのイベントのフロー: マルチキャスト・データグラムの送信

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、マルチキャスト・データグラムを互いに送受信する 2 つのアプリケーションの関係の説明ともなっています。それぞれのフローに

は、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: マルチキャスト・データグラムの送信』では、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_DGRAM) を使用することも示します。このソケットが、データグラムをもう一方のアプリケーションに送信します。
2. sockaddr_in 構造が、宛先 IP アドレスおよびポート番号を指定します。この例の場合、アドレスは 225.1.1.1 でポート番号は 5555 です。
3. **setsockopt()** 関数が、IP_MULTICAST_LOOP ソケット・オプションを設定します。したがって、送信側のシステムは、送信するマルチキャスト・データグラムのコピーを受信しません。
4. **setsockopt()** 関数が、IP_MULTICAST_IF ソケット・オプションを使用します。このオプションは、マルチキャスト・データグラムを送信するローカル・インターフェースを定義します。
5. **sendto()** 関数が、指定されたグループ IP アドレスへ、マルチキャスト・データグラムを送信します。
6. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

ソケットのイベントのフロー: マルチキャスト・データグラムの受信

『例: マルチキャスト・データグラムの受信』では、以下の関数呼び出しのシーケンスを使用します。

1. **socket()** 関数が、端点を表すソケット記述子を戻します。ステートメントは、このソケットのために INET (インターネット・プロトコル) アドレス・ファミリーと TCP トランスポート (SOCK_DGRAM) を使用することも示します。このソケットが、データグラムをもう一方のアプリケーションに送信します。
2. **setsockopt()** 関数が、SO_REUSEADDR ソケット・オプションを、同じローカル・ポート番号に宛先があるデータグラムを複数のアプリケーションが受信するように設定します。
3. **bind()** 関数が、ローカル・ポート番号を指定します。この例の場合、マルチキャスト・グループに宛てられたデータグラムを受信するため、IP アドレスは INADDR_ANY と指定されます。
4. **setsockopt()** 関数が、IP_ADD_MEMBERSHIP ソケット・オプションを使用します。このオプションは、データグラムを受信するマルチキャスト・グループを結合します。グループを結合する際には、ローカル・インターフェースの IP アドレスと共にクラス D グループ・アドレスを指定します。システムは、マルチキャスト・データグラムを受信するそれぞれのローカル・インターフェースについて、IP_ADD_MEMBERSHIP ソケット・オプションを呼び出さなければなりません。この例の場合、マルチキャスト・グループ (225.1.1.1) は、ローカル 9.5.1.1 インターフェース上で結合されます。

   注: IP_ADD_MEMBERSHIP オプションは、マルチキャスト・データグラムを受信するローカル・インターフェースごとに呼び出すことが必要です。
5. **read()** 関数が、送信されているマルチキャスト・データグラムを読み取ります。
6. **close()** 関数が、オープンしているソケット記述子をすべてクローズします。

例: マルチキャスト・データグラムの送信

以下の例では、ソケットが以下にリストされている手順を実行して、マルチキャスト・データグラムを送信できるようにします。このプログラムのソケットのイベントのフローに関する説明を再検討したい場合、『例: マルチキャスティングの使用』を参照してください。

コード例の使用については、『コードの特記事項情報』を参照してください。

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

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>


struct in_addr        localInterface;
struct sockaddr_in    groupSock;
int                   sd;
int                   datalen;
char                  databuf[1024];

int main (int argc, char *argv[])
{

  /* ----------------------------------------------------------------*/
  /*                                                                 */
  /* Send Multicast Datagram code example.                           */
  /*                                                                 */
  /* ----------------------------------------------------------------*/

  /*
   * Create a datagram socket on which to send.
   */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("opening datagram socket");
    exit(1);
  }

  /*
   * Initialize the group sockaddr structure with a
   * group address of 225.1.1.1 and port 5555.
   */
  memset((char *) &groupSock, 0, sizeof(groupSock));
  groupSock.sin_family = AF_INET;
  groupSock.sin_addr.s_addr = inet_addr("225.1.1.1");
  groupSock.sin_port = htons(5555);

  /*
   * Disable loopback so you do not receive your own datagrams.
   */
  {
    char loopch=0;

    if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP,
                   (char *)&loopch, sizeof(loopch)) < 0) {
      perror("setting IP_MULTICAST_LOOP:");
      close(sd);
      exit(1);
    }
  }

  /*
   * Set local interface for outbound multicast datagrams.
   * The IP address specified must be associated with a local,
   * multicast-capable interface.
   */
  localInterface.s_addr = inet_addr("9.5.1.1");
  if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF,
                 (char *)&localInterface,
                 sizeof(localInterface)) < 0) {
    perror("setting local interface");
    exit(1);
  }


  /*
```

```
   * Send a message to the multicast group specified by the
   * groupSock sockaddr structure.
   */
  datalen = 10;
  if (sendto(sd, databuf, datalen, 0,
             (struct sockaddr*)&groupSock,
             sizeof(groupSock)) < 0)
  {
    perror("sending datagram message");
  }
}
```

例: マルチキャスト・データグラムの受信

以下の例では、ソケットが以下にリストされている手順を実行して、マルチキャスト・データグラムを受信できるようにします。このプログラムのソケットのイベント・フローに関する説明を再検討したい場合、『例: マルチキャスティングの使用』を参照してください。

コード例の使用については、『コードの特記事項情報』を参照してください。

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



struct sockaddr_in    localSock;
struct ip_mreq        group;
int                   sd;
int                   datalen;
char                  databuf[1024];

int main (int argc, char *argv[])
{

  /* ----------------------------------------------------------*/
  /*                                                           */
  /* Receive Multicast Datagram code example.                  */
  /*                                                           */
  /* ----------------------------------------------------------*/

  /*
   * Create a datagram socket on which to receive.
   */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("opening datagram socket");
    exit(1);
  }

  /*
   * Enable SO_REUSEADDR to allow multiple instances of this
   * application to receive copies of the multicast datagrams.
   */
  {
    int reuse=1;

    if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                   (char *)&reuse, sizeof(reuse)) < 0) {
      perror("setting SO_REUSEADDR");
      close(sd);
      exit(1);
    }
```

```
  }

  /*
   * Bind to the proper port number with the IP address
   * specified as INADDR_ANY.
   */
  memset((char *) &localSock, 0, sizeof(localSock));
  localSock.sin_family = AF_INET;
  localSock.sin_port = htons(5555);;
  localSock.sin_addr.s_addr  = INADDR_ANY;

  if (bind(sd, (struct sockaddr*)&localSock, sizeof(localSock))) {
    perror("binding datagram socket");
    close(sd);
    exit(1);
  }


  /*
   * Join the multicast group 225.1.1.1 on the local 9.5.1.1
   * interface.  Note that this IP_ADD_MEMBERSHIP option must be
   * called for each local interface over which the multicast
   * datagrams are to be received.
   */
  group.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
  group.imr_interface.s_addr = inet_addr("9.5.1.1");
  if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (char *)&group, sizeof(group)) < 0) {
    perror("adding multicast group");
    close(sd);
    exit(1);
  }

  /*
   * Read from the socket.
   */
  datalen = sizeof(databuf);
  if (read(sd, databuf, datalen) < 0) {
    perror("reading datagram message");
    close(sd);
    exit(1);
  }

}
```

例: DNS の更新および照会

以下の例では、ドメイン・ネーム・システム (DNS) レコードの照会方法と更新方法を示します。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*****************************************************************************/
/* This program updates a DNS using a transaction signature (TSIG) to     */
/* sign the update packet.  It then queries the DNS to verify success.    */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

/*****************************************************************************/
```

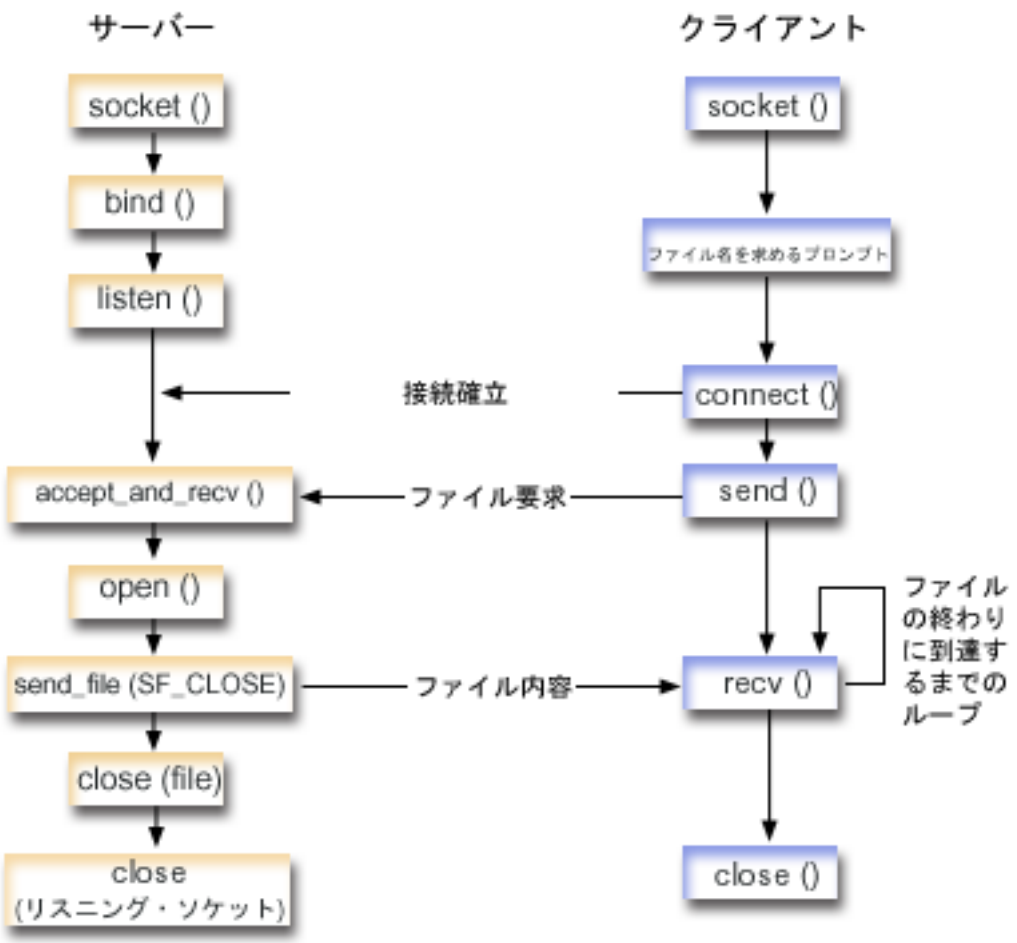
```
/* Declare update records - a zone record, a pre-requisite record, and    */
/* 2 update records                                                       */
/*************************************************************************/
ns_updrec update_records[] =
{
   {
      {NULL,&update_records[1]},
      {NULL,&update_records[1]},
      ns_s_zn,                  /* a zone record */
      "mydomain.ibm.com.",
      ns_c_in,
      ns_t_soa,
      0,
      NULL,
      0,
      0,
      NULL,
      NULL,
      0
   },
   {
      {&update_records[0],&update_records[2]},
      {&update_records[0],&update_records[2]},
      ns_s_pr,                 /* pre-req record */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_a,
      0,
      NULL,
      0,
      ns_r_nxdomain,    /* record must not exist */
      NULL,
      NULL,
      0
   },
   {
      {&update_records[1],&update_records[3]},
      {&update_records[1],&update_records[3]},
      ns_s_ud,                  /* update record */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_a,                   /* IPv4 address  */
      10,
      (unsigned char *)"10.10.10.10",
      11,
      ns_uop_add,                /* to be added  */
      NULL,
      NULL,
      0
   },
   {
      {&update_records[2],NULL},
      {&update_records[2],NULL},
      ns_s_ud,                  /* update record */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_aaaa,                /* IPv6 address  */
      10,
      (unsigned char *)"fedc:ba98:7654:3210:fedc:ba98:7654:3210",
      39,
      ns_uop_add,                /* to be added  */
      NULL,
      NULL,
      0
   }
};
```

```
/*****************************************************************************/
/* These two structures define a key and secret that must match the one   */
/* configured on the DNS :                                                   */
/*   allow-update {                                                          */
/*   key my-long-key.;                                                       */
/*   }                                                                       */
/*                                                                           */
/* This must be the binary equivalent of the base64 secret for               */
/* the key                                                                   */
/*****************************************************************************/
unsigned char secret[18] =
{
   0x6E,0x86,0xDC,0x7A,0xB9,0xE8,0x86,0x8B,0xAA,
   0x96,0x89,0xE1,0x91,0xEC,0xB3,0xD7,0x6D,0xF8
};

ns_tsig_key my_key = {
   "my-long-key",            /* This key must exist on the DNS */
   NS_TSIG_ALG_HMAC_MD5,
   secret,
   sizeof(secret)
};

void main()
{
   /*****************************************************************************/
   /* Variable and structure definitions.                                       */
   /*****************************************************************************/
   struct state res;
   int result, update_size;
   unsigned char update_buffer[2048];
   unsigned char answer_buffer[2048];
   int buffer_length = sizeof(update_buffer);

   /* Turn off the init flags so that the structure will be initialized   */
   res.options &= ~ (RES_INIT | RES_XINIT);

   result = res_ninit(&res);

   /* Put processing here to check the result and handle errors           */

   /* Build an update buffer (packet to be sent) from the update records  */
   update_size = res_nmkupdate(&res, update_records,
                               update_buffer, buffer_length);

   /* Put processing here to check the result and handle errors           */

   {
      char zone_name[NS_MAXDNAME];
      size_t zone_name_size = sizeof zone_name;
      struct sockaddr_in s_address;
      struct in_addr addresses[1];
      int number_addresses = 1;

   /* Find the DNS server that is authoritative for the domain            */
   /* that we want to update                                              */

      result = res_findzonecut(&res, "mypc.mydomain.ibm.com", ns_c_in, 0,
      zone_name, zone_name_size,
                               addresses, number_addresses);

   /* Put processing here to check the result and handle errors           */

   /* Check if the DNS server found is one of our regular DNS addresses   */
      s_address.sin_addr = addresses[0];
      s_address.sin_family = res.nsaddr_list[0].sin_family;
      s_address.sin_port = res.nsaddr_list[0].sin_port;
```

```
   memset(s_address.sin_zero, 0x00, 8);

   result = res_nisourserver(&res, &s_address);

/* Put processing here to check the result and handle errors          */

/* Set the DNS address found with res_findzonecut into the res        */
/* structure.  We will send the (TSIG signed) update to that DNS.     */
   res.nscount = 1;
   res.nsaddr_list[0] = s_address;

/* Send a TSIG signed update to the DNS                               */
   result = res_nsendsigned(&res, update_buffer, update_size,
                            &my_key,
                            answer_buffer, sizeof answer_buffer);

/* Put processing here to check the result and handle errors          */
}

/*********************************************************************/
/* The res_findzonecut(), res_nmkupdate(), and res_nsendsigned()      */
/* could be replaced with one call to res_nupdate() using             */
/* update_records[1] to skip the zone record:                         */
/*                                                                    */
/* result = res_nupdate(&res, &update_records[1], &my_key);           */
/*                                                                    */
/*********************************************************************/
/*********************************************************************/
/* Now verify that our update actually worked!                        */
/* We choose to use TCP and not UDP, so set the appropriate option now */
/* that the res variable has been initialized. We also want to ignore  */
/* the local cache and always send the query to the DNS server.       */
/*********************************************************************/

res.options |= RES_USEVC|RES_NOCACHE;

/* Send a query for mypc.mydomain.ibm.com address records             */
result = res_nquerydomain(&res,"mypc", "mydomain.ibm.com.",
                          ns_c_in, ns_t_a,
                          update_buffer, buffer_length);

/* Sample error handling and printing errors                          */
if (result == -1)
{
   printf("\nquery domain failed.  result = %d \nerrno: %d: %s \
            \nh_errno: %d: %s",
           result,
          errno, strerror(errno),
           h_errno, hstrerror(h_errno));
}
/*********************************************************************/
/* The output on a failure will be:                                   */
/*                                                                    */
/* query domain failed.  result = -1                                  */
/* errno: 0: There is no error.                                       */
/* h_errno: 5: Unknown host                                           */
/*********************************************************************/
return;
}
```

例: send_file() および accept_and_recv() API を使用したファイル・データの転送

以下の例では、**send_file()** および **accept_and_recv()** API を使用して、サーバーがクライアントと通信できるようにします。



ソケットのイベントのフロー: ファイルの内容を送信するサーバー

以下のソケット呼び出しのシーケンスは、図の説明となっています。これはまた、ファイルを互いに送受信する 2 つのアプリケーションの関係の説明ともなっています。それぞれのフローには、特定の API の使用上の注意へのリンクが含まれています。特定の API の使用に関する詳細な説明を参照するために、これらのリンクを使用できます。『例: accept_and_recv() および send_file() API を使用したファイルの内容の送信』では、以下の関数呼び出しのシーケンスを使用します。

1. サーバーは、**socket()**、**bind()**、および **listen()** を呼び出して、listen するソケットを作成します。
2. サーバーは、ローカルおよびリモート・アドレス構造を初期化します。
3. サーバーは **accept_and_recv()** を呼び出して着信接続を待機し、最初のデータ・バッファーがこの接続に到着するように待機します。この呼び出しは、受信したバイト数、さらにこの接続に関連したローカルおよびリモート・アドレスを戻します。この呼び出しは、**accept()**、**getsockname()**、および **recv()** API の組み合わせです。
4. サーバーは **open()** を呼び出して、クライアント・アプリケーションから **accept_and_recv()** のデータとして名前を取得したファイルをオープンします。
5. **memset()** 関数を使用して、sf_parms 構造のすべてのフィールドを初期値 0 に設定します。サーバーはファイル記述子フィールドを **open()** が戻した値に設定します。サーバーはファイルのバイト・フィールドを -1 に設定して、サーバーがこのファイル全体を送信するように指示します。システムはファイル全体を送信しており、したがって、ファイル・オフセット・フィールドを割り当てる必要はありません。

6. サーバーは **send_file()** を呼び出してファイルの内容を転送します。 **send_file()** は、ファイル全体が送信されるか、または割り込みが発生するまで完了しません。**send_file()** の方が効果的であるといえます。アプリケーションはファイルが終了するまで **read()** および **send()** ループに入る必要がないためです。
7. サーバーは、**send_file()** API に SF_CLOSE フラグを指定します。 SF_CLOSE フラグは、ファイルの最後のバイトおよびトレーラー・バッファー (指定されている場合) が正常に送信された場合には、自動的にソケット接続をクローズするべきであることを **send_file()** API に通知します。 SF_CLOSE フラグが指定されている場合には、アプリケーションが **close()** を呼び出す必要はありません。

ソケットのイベントのフロー: クライアントのファイル要求

『例: クライアントのファイル要求』では、以下の関数呼び出しのシーケンスが使用されます。

1. このクライアント・プログラムでは 0 から 2 つのパラメーターを取ります。

   最初のパラメーター (指定されている場合) は、ドット 10 進数の IP アドレスまたはサーバー・アプリケーションのあるホスト名です。

   2 番目のパラメーター (指定されている場合) は、クライアントがサーバーから取得しようとしているファイルの名前です。サーバー・アプリケーションは、指定されたファイルの内容をクライアントに送信します。ユーザーがパラメーターを指定しない場合には、クライアントはサーバーの IP アドレスに INADDR_ANY を使用します。ユーザーが 2 番目のパラメーターを指定しない場合には、プログラムはファイル名を入力するようにユーザーにプロンプトを出します。
2. クライアントは **socket()** を呼び出してソケット記述子を作成します。
3. クライアントは **connect()** を呼び出してサーバーへの接続を確立します。ステップ 1 でサーバーの IP アドレスを取得しています。
4. クライアントは **send()** を呼び出して、サーバーに取得したいファイル名を伝えます。ステップ 1 でファイルの名前を取得しています。
5. クライアントは「do」ループに入り、ファイルの終わりに達するまで **recv()** を呼び出します。 **recv()** で戻りコードが 0 の場合は、サーバーが接続をクローズしたことを意味します。
6. クライアントは **close()** を呼び出してソケットをクローズします。

例: accept_and_recv() および send_file() API を使用したファイルの内容の送信

以下の例では、**send_file()** および **accept_and_recv()** API を使用して、サーバーが以下にリストされている手順を実行して、クライアントと通信できるようにします。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/************************************************/
/* Server example send file data to client      */
/************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    i, num, rc, flag = 1;
   int    fd, listen_sd, accept_sd = -1;
```

```
size_t local_addr_length;
size_t remote_addr_length;
size_t total_sent;

struct sockaddr_in   addr;
struct sockaddr_in   local_addr;
struct sockaddr_in   remote_addr;
struct sf_parms      parms;

char   buffer[255];

/************************************************/
/* If an argument is specified, use it to       */
/* control the number of incoming connections   */
/************************************************/
if (argc >= 2)
   num = atoi(argv[1]);
else
   num = 1;

/************************************************/
/* Create an AF_INET stream socket to receive   */
/* incoming connections on                      */
/************************************************/
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if (listen_sd < 0)
{
   perror("socket() failed");
   exit(-1);
}

/************************************************/
/* Set the SO_REUSEADDR bit so that you do not  */
/* have to wait 2 minutes before restarting     */
/* the server                                   */
/************************************************/
rc = setsockopt(listen_sd,
                SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&flag,
                sizeof(flag));
if (rc < 0)
{
   perror("setsockop() failed");
   close(listen_sd);
   exit(-1);
}

/************************************************/
/* Bind the socket                              */
/************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family      = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port        = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
   perror("bind() failed");
   close(listen_sd);
   exit(-1);
}

/************************************************/
/* Set the listen backlog                       */
```

```
/*********************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("listen() failed");
   close(listen_sd);
   exit(-1);
}

/*********************************************/
/* Initialize the local and remote addr lengths  */
/*********************************************/
local_addr_length  = sizeof(local_addr);
remote_addr_length = sizeof(remote_addr);

/*********************************************/
/* Inform the user that the server is ready      */
/*********************************************/
printf("The server is ready\n");

/*********************************************/
/* Go through the loop once for each connection  */
/*********************************************/
for (i=0; i < num; i++)
{
   /*********************************************/
   /* Wait for an incoming connection            */
   /*********************************************/
   printf("Iteration: %d\n", i+1);
   printf("  waiting on accept_and_recv()\n");

   rc = accept_and_recv(listen_sd,
                        &accept_sd,
                        (struct sockaddr *)&remote_addr,
                        &remote_addr_length,
                        (struct sockaddr *)&local_addr,
                        &local_addr_length,
                        &buffer,
                        sizeof(buffer));
   if (rc < 0)
   {
      perror("accept_and_recv() failed");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("  Request for file: %s\n", buffer);

   /*********************************************/
   /* Open the file to retrieve                  */
   /*********************************************/
   fd = open(buffer, O_RDONLY);
   if (fd < 0)
   {
      perror("open() failed");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }

   /*********************************************/
   /* Initialize the sf_parms structure          */
   /*********************************************/
   memset(&parms, 0, sizeof(parms));
   parms.file_descriptor = fd;
   parms.file_bytes      = -1;
```

```
      /*********************************************/
      /* Initialize the counter of the total number */
      /* of bytes sent                              */
      /*********************************************/
      total_sent = 0;

      /*********************************************/
      /* Loop until the entire file has been sent   */
      /*********************************************/
      do
      {
         rc = send_file(&accept_sd, &parms, SF_CLOSE);
         if (rc < 0)
         {
            perror("send_file() failed");
            close(fd);
            close(listen_sd);
            close(accept_sd);
            exit(-1);
         }
         total_sent += parms.bytes_sent;

      } while (rc == 1);

      printf("  Total number of bytes sent: %d\n", total_sent);

      /*********************************************/
      /* Close the file that is sent out           */
      /*********************************************/
      close(fd);
   }

   /*********************************************/
   /* Close the listen socket                    */
   /*********************************************/
   close(listen_sd);

   /*********************************************/
   /* Close the accept socket                    */
   /*********************************************/
   if (accept_sd != -1)
      close(accept_sd);
}
```

例: クライアントのファイル要求

以下の例では、クライアントはサーバーからファイルを要求し、サーバーがそのファイルの内容を送信するのを待ちます。

コード例の使用については、『コードの特記事項情報』を参照してください。

```
/*********************************************/
/* Client example requests file data from server */
/*********************************************/
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    rc, sockfd;
```

```
char   filename[256];
char   buffer[32 * 1024];

struct sockaddr_in   addr;
struct hostent      *host_ent;

/*************************************************/
/* Initialize the socket address structure       */
/*************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port   = htons(SERVER_PORT);

/*************************************************/
/* Determine the host name and IP address of the */
/* machine the server is running on              */
/*************************************************/
if (argc < 2)
{
   addr.sin_addr.s_addr = htonl(INADDR_ANY);
}
else if (isdigit(*argv[1]))
{
   addr.sin_addr.s_addr = inet_addr(argv[1]);
}
else
{
   host_ent = gethostbyname(argv[1]);
   if (host_ent == NULL)
   {
      printf("Host not found!\n");
      exit(-1);
   }
   memcpy((char *)&addr.sin_addr.s_addr,
          host_ent->h_addr_list[0],
          host_ent->h_length);
}

/*************************************************/
/* Check to see if the user specified a file name */
/* on the command line                            */
/*************************************************/
if (argc == 3)
{
   strcpy(filename, argv[2]);
}
else
{
   printf("Enter the name of the file:\n");
   gets(filename);
}

/*************************************************/
/* Create an AF_INET stream socket               */
/*************************************************/
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
   perror("socket() failed");
   exit(-1);
}
printf("Socket completed.\n");

/*************************************************/
/* Connect to the server                         */
/*************************************************/
```

```
   rc = connect(sockfd,
                (struct sockaddr *)&addr,
                sizeof(struct sockaddr_in));
   if (rc < 0)
   {
      perror("connect() failed");
      close(sockfd);
      exit(-1);
   }
   printf("Connect completed.\n");

   /*************************************************/
   /* Send the request over to the server           */
   /*************************************************/
   rc = send(sockfd, filename, strlen(filename) + 1, 0);
                  if (rc < 0)
   {
      perror("send() failed");
      close(sockfd);
      exit(-1);
   }
   printf("Request for %s sent\n", filename);

   /*************************************************/
   /* Receive the file from the server              */
   /*************************************************/
   do
   {
      rc = recv(sockfd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("recv() failed");
         close(sockfd);
         exit(-1);
      }
      else if (rc == 0)
      {
         printf("End of file\n");
         break;
      }
      printf("%d bytes received\n", rc);

   } while (rc > 0);

   /*************************************************/
   /* Close the socket                              */
   /*************************************************/
   close(sockfd);
}
```

Xsocket ツール

Xsocket ツールは、iSeries に付属している多くのツールの 1 つです。すべてのツールは QUSRTOOL ライブラリーに保管されています。 Xsocket を使用すれば、プログラマーはソケット API を対話式に処理できます。 Xsocket ツールによって、以下のタスクを実行できます。

- Socket API について学習する
- 特定のシナリオを対話式に再作成してデバッグに役立てる

注: Xsocket ツールは、「現状のまま」の形式で出荷されています。

Xsocket の前提条件

- ILE C/400® 言語がインストールされている。

- ライセンス・プログラム 5722-SS1「オープンネス」がインストールされている。
- ライセンス・プログラム 5722-DG1「IBM HTTP Server」がインストールされている。

  注: Web ブラウザーで Xsocket を使用することを計画している場合には、これが必要です。
- ライセンス・プログラム 5722-JV1「JDK (Java 開発キット)」がインストールされている。

  注: Web ブラウザーで Xsocket を使用することを計画している場合には、これが必要です。

Xsocket ツールをインストールして使用するには、以下のトピックを利用してください。

Xsocket の構成
このトピックでは、Xsocket ツールの作成方法を説明します。これはソケット・プログラムを設計してコンパイルするために役立ちます。

Xsocket の使用
このトピックでは、Xsocket ツールの使用方法を説明します。

Xsocket のカスタマイズ
このトピックでは、Xsocket ツールのカスタマイズ方法を説明します。

Xsocket の構成

2 つのバージョンのツールを作成することができます。 1 つ目は、iSeries 固有のクライアントです。最初の説明は、この固有バージョンを作成するためのものです。 2 つ目のバージョンは Web ブラウザーをクライアントとして使用します。 Web ブラウザー・クライアントを使用する場合、まず固有バージョン用のセットアップ手順を最後まで実行しておく必要があります。

Xsocket ツールを作成するには、以下のステップを実行します。

1. ツールをアンパックするには、コマンド行から次のように入力します。

   ```
   CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)
   ```

   注: 左の ' と右の ' の間に 10 文字の間隔を開ける必要があります。
2. ライブラリー・リストに QUSRTOOL ライブラリーを追加するために、コマンド行に次のように入力します。

   ```
   ADDLIBLE QUSRTOOL
   ```
3. コマンド行で次のように入力し、Xsocket プログラム・ファイルを作成するライブラリーを作成します。

   ```
   CRTLIB <library-name>
   ```

   <library-name> は、Xsocket ツール・オブジェクトを作成するライブラリーです。たとえば、

   ```
   CRTLIB MYXSOCKET
   ```

   は有効なライブラリー名です。

   注: Xsocket ツール・オブジェクトを QUSRTOOL ライブラリーに追加しないでください。そのディレクトリー内の他のツールの使用を妨げる可能性があります。
4. ライブラリー・リストにこのライブラリーを追加するために、コマンド行に次のように入力します。

   ```
   ADDLIBLE <library-name>
   ```

<library-name> は、ステップ 3 で作成したライブラリーです。たとえば、MYXSOCKET というライブラリー名を使用した場合は、次のように入力します。

```
ADDLIBLE MYXSOCKET
```

5. コマンド行で次のように入力することにより、Xsocket ツールを自動的にインストールするインストール・プログラム TSOCRT を作成します。

   ```
   CRTCLPGM <library-name>/TSOCRT QUSRTOOL/QATTCL
   ```

   <library-name> は、ステップ 3 で作成したライブラリーです。たとえば、MYXSOCKET というライブラリー名を使用した場合は、次のように入力します。

   ```
   CRTCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
   ```

6. インストール・プログラムを呼び出すために、コマンド行で次のように入力します。

   ```
   CALL TSOCRT library-name
   ```

   library-name の場所には、ステップ 3 で作成したライブラリーを使用します。たとえば、MYXSOCKET ライブラリーにツールを作成する場合は、次のように入力します。

   ```
   CALL TSOCRT MYXSOCKET
   ```

   注: これが完了するまで、数分かかる場合があります。

ジョブ制御 (*JOBCTL) 特殊権限なしに TSOCRT を呼び出してソケット・ツールを作成すると、**givedescriptor()** ソケット関数は、実行しているものとは異なるジョブに記述子を渡そうとして、エラーを戻します。

TSOCRT は、1 つの CL プログラム、1 つの ILE C/400 プログラム (2 つのモジュールが作成される)、2 つの ILE C/400 サービス・プログラム (2 つのモジュールが作成される)、および 3 つの表示装置ファイルを作成します。このツールの使用する場合はいつも、ライブラリーをライブラリー・リストに追加してください。このツールで作成されたすべてのオブジェクトの名前には、TSO という接頭部が付きます。

次に実行する内容:

固有 Xsocket の使用
固有の Xsocket を使用する場合は、このステップに進みます。このトピックでは、固有 Xsocket ツールの使用の基本を扱います。

注: 固有バージョンは、GSKit セキュア・ソケット API をサポートしていません。この API を使用するソケット・プログラムを作成する場合、このツールのブラウザー・ベースのバージョンを使用してください。

Web ブラウザーを使用するよう Xsocket を構成する

注: このステップはオプションです。

固有の Xsocket のセットアップで作成されるオブジェクト

以下は、インストール・プログラムによって作成されるオブジェクトをリストした表です。作成されるすべてのオブジェクトは、指定されるライブラリーの中にあります。

表 20. Xsocket のインストール中に作成されるオブジェクト

オブジェクト名	メンバー名	ソース・ファイル名	オブジェクト・タイプ	拡張子	説明
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	JSP と TSOSTSOC の間のインターフェースで使用するモジュール
TSODLT	TSODLT	QATTCL	*PGM	CLP	ツール・オブジェクトおよびソース・ファイルのメンバー (またはその一方) を削除する制御言語プログラム。
TSOXSOCK	N/A	N/A	*PGM	C	SOCKETS 対話式ツールで使用するメインプログラム。
TSOXGJOB	N/A	N/A	*SRVPGM	C	SOCKETS 対話式ツールのサポートで使用されるサービス・プログラム。
TSOJNI	N/A	N/A	*SRVPGM	C	SOCKETS 対話式ツールのサポートのため、JSP と TSOSTSOC の間のインターフェースで使用されるサービス・プログラム。
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	TSOXSOCK プログラムの作成で使用されるモジュール。ソース・ファイルには main() ルーチンが含まれます。

表 20. *Xsocket* のインストール中に作成されるオブジェクト *(続き)*

TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	TSOXSOCK プログラムの作成で使用されるモジュール。ソース・ファイルには、ソケット関数を実際に呼び出すルーチンが含まれます。
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	TSOXGJOB サービス・プログラムの作成で使用されるモジュール。ソース・ファイルには、内部ジョブを識別するルーチンが含まれます。この内部ジョブ ID は、ジョブ名、ユーザー ID、およびジョブ番号で構成されます。
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	ソケット関数を含むメイン・ウィンドウ用に Xsocket ツールが使用するディスプレイ・ファイル。
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	種々のソケット関数のサポートに Xsocket ツールが使用するディスプレイ・ファイル。
TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	メニュー・バーをサポートするため、Xsocket ツールが使用するディスプレイ・ファイル。
QATTIFS2	N/A	N/A	*FILE	PF-DTA	Tomcat Server が使用する JAR ファイルが含まれます。

Web ブラウザーを使用するよう Xsocket を構成する

以下の手順を実行することにより、Web ブラウザーから Xsocket ツールにアクセスできるようになります。同一システム上でこれらの指示を複数回実装することにより、異なるサーバー・インスタンスを作成す

ることができます。これにより、異なる listen ポートで、複数のバージョンを同時に実行できるようになります。 Web ブラウザーを使用できるよう Xsocket を構成するには、以下のタスクを実行する必要があります。

1. HTTP サーバー (powered by Apache) の構成
2. Tomcat の構成
3. 構成ファイルの更新
4. ブラウザーにおける Xsocket ツールのテスト

HTTP サーバー (powered by Apache) の構成

Web ブラウザーを構成して Xsocket ツールで作業する前に、まず固有 Xsocket の構成を完了する必要があります。以下のステップで HTTP サーバー (powered by Apache) を構成すると、Web ブラウザーで Xsocket ツールを使用できます。

1. HTTP 管理インスタンスが QHTTPSVR サブシステム下で実行されていることを確認します。実行されていない場合、以下の CL コマンドでこれを開始できます。

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```

2. Web ブラウザーで、次のように入力します。

   ```
   http://<system_name>:2001/
   ```

   ここで、<system_name> は iSeries のマシン名です。たとえば、http://myiSeries:2001/ などです。
3. iSeries の「タスク (Tasks)」ページで、「**IBM HTTP Server for iSeries**」を選択します。
4. トップ・メニューから「**セットアップ (Setup)**」タブを選択します。
5. 「**新規 HTTP サーバーの作成 (Create New HTTP Server)**」をクリックします。
6. 「**HTTP サーバー (powered by Apache)(HTTP server (powered by Apache))**」を選択し、「**次へ (Next)**」をクリックします。
7. サーバー・インスタンスの名前を入力します。たとえば、このインスタンスはブラウザーで Xsocket ツールを実行するので、xsocket という名前を使用できます。「**次へ (Next)**」をクリックします。
8. 「**いいえ (No)**」を選択します。これにより、既存のサーバーに基づかない新しいサーバー・インスタンスが作成されます。「**次へ (Next)**」をクリックします。
9. 「**次へ (Next)**」をクリックし、デフォルトのサーバー・ルート・ディレクトリーを受け入れます。
10. 「**次へ (Next)**」をクリックし、デフォルトの文書ルート・ディレクトリーを受け入れます。
11. 使用する IP アドレスおよび使用可能なポートを選択します。 1024 より大きいポート番号を使用してください。「**次へ (Next)**」をクリックします。

    注: デフォルトのポート番号 80 は選択しないでください。
12. 「**はい (yes)**」または「**いいえ (no)**」を選択して、このサーバーのアクセス・ログを作成するかどうかを指示します。「**次へ (Next)**」をクリックします。
13. 次のページには、HTTP サーバー (powered by Apache) の構成設定が表示されます。これらの設定が正しいなら、「**終了 (Finish)**」をクリックします。
14. 「**新しく作成したサーバーの管理 (Manage newly created server)**」をクリックします。これで、Apache による構成が完了しました。

次に実行する内容:

Tomcat の構成

Tomcat の構成

HTTP サーバー (powered by Apache) のサーバー・インスタンスを構成した後、Tomcat を構成して Web ブラウザー内で Xsocket ツールを立ち上げることが必要です。

1. 「**ダイナミック・コンテンツ (Dynamic content)**」の見出しの下にある、「**ASF Tomcat セットアップ・サーバー・タスク (ASF Tomcat Setup Server Task)**」を選択します。
2. 「**この HTTP サーバー用のサーブレットを使用可能にする (Enable servlets for this HTTP Server)**」を選択します。これは、ワーカー定義ファイルに書き込まれます。「**次へ (Next)**」をクリックします。
3. 「**次へ (Next)**」をクリックして、「**ワーカー定義 (Workers Definition)**」 ページのデフォルトを受け入れます。
4. 「**ワーカー・マッピングへの URL (URL to Worker Mapping)**」ページで、「**追加 (Add)**」をクリックします。
5. 「**URL (マウント・ポイント)(URL(Mount Point))**」欄に、`/xsock` と入力します。「**続く (Continue)**」をクリックします。
6. 「**追加 (Add)**」をクリックします。
7. 「**URL (マウント・ポイント)(URL(Mount Point))**」欄に、`/xsock/*` と入力します。「**続く (Continue)**」をクリックします。
8. 「**次へ (Next)**」をクリックします。
9. 「**処理中アプリケーションのコンテキスト定義 (In-Process Application Context Definition)**」ページで、「**追加 (Add)**」をクリックします。
10. 「**URL パス (URL Path)**」欄に、`/xsock` と入力します。
11. 「**アプリケーション基本ディレクトリー (Application Base Directory)**」欄に、`webapps/xsock` と入力します。
12. 「**続く (Continue)**」をクリックします。さらに他の情報を構成する必要があることを示す、警告メッセージが表示されます。
13. 「**アプリケーションの構成 (Configure Application)**」の下にある「**構成 (Configure)**」をクリックします。
14. オープンした新しいブラウザー・ウィンドウの「**オブジェクトのタイムアウト期間 (Session Object timeout)**」フィールドで、3 日を選択します。

    注: これが推奨される値ですが、「**オブジェクトのタイムアウト期間 (Session Object timeout)**」で他の値を指定することもできます。
15. 「**追加 (Add)**」をクリックして、サーブレット定義を追加し以下のステップを完了します。
    a. 「**サーブレット・クラス名 (Servlet class name)**」に、`com.ibm.iseries.xsocket.XSocketServlet` と入力します。
    b. 「**URL パターン (URL patterns)**」に、`/*` と入力します。
    c. 「**始動負荷シーケンス (Startup load sequence)**」を 3 に設定します。
    d. 「**続く (Continue)**」をクリックします。
    e. 「**OK**」をクリックします。これにより、ブラウザー・ウィンドウがクローズします。
16. Tomcat セットアップのメイン・ウィンドウで、「**次へ (Next)**」をクリックします。
17. 「**終了 (Finish)**」をクリックします。
18. 「**OK**」をクリックします。これで、Xsocket ツールの Tomcat 構成が完了しました。

次に実行する内容:

構成ファイルの更新

構成ファイルの更新

HTTP サーバー (powered by Apache) で Tomcat を構成し、Xsocket ツールを実行できるようにした後、インスタンスのいくつかの構成ファイルに手動で変更を加える必要があります。 3 つのファイルを更新する必要があります。web.xml ファイル、JAR ファイル、および httpd.conf です。このステップを完了するには、以下の情報が必要です。

- Xsocket アプリケーション・ファイルを含む、ライブラリー名。固有クライアント用の最初の Xsocket の構成時に、このファイルが作成されます。
- HTTP サーバー (powered by Apache) の構成時に作成した、サーバー名。

1. **web.xml ファイルの更新**
   a. コマンド行から、次のように入力します。

      ```
      wrklnk '/www/<server_name>/webapps/xsock/WEB-INF/web.xml'
      ```

      ここで、<server_name> は Apache 構成時に作成したサーバー・インスタンスの名前です。たとえば、サーバー名として xsocks を選んだ場合、次のように入力します。

      ```
      wrklnk '/www/xsocks/webapps/xsock/WEB-INF/web.xml'
      ```

   b. 2 を押してこのファイルを編集します。
   c. web.xml ファイルの </servlet-class> 行を検索します。
   d. この行の後に、以下のコードを挿入します。

      ```
      <init-param>
                <param-name>library</param-name>
                <param-value>XXXX</param-value>
      </init-param>
      ```

      XXXX と表示されている場所に、Xsocket の構成時に作成したライブラリー名を挿入します。
   e. ファイルを保管し、編集セッションを終了します。
2. **JAR ファイルの移動**
   a. コマンド行から、次のようにこのコマンドを入力します。

      ```
      CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/TSOXSOCK.MBR')
         TOOBJ('/www/<server_name>/webapps/xsock/WEB-INF/lib/tsoxsock.jar')
         FROMCCSID(*OBJ) TOCCSID(819) OWNER(*NEW)
      ```

      ここで、XXXX は Xsocket の構成時に作成したライブラリー名で、 <server_name> は HTTP サーバー (powered by Apache) の構成時に作成したサーバー・インスタンスの名前です。
3. **httpd.conf ファイルへの権限検査の追加** (このステップはオプションです。)
   これにより、Apache は Xsocket Web アプリケーションにアクセスしようとするユーザーの認証を行うようになります。

   注: UNIX ソケットを作成するために書き込みアクセスする際にも必要です。
   a. コマンド行から、次のように入力します。

      ```
      wrklnk '/www/<server_name>/conf/httpd.conf'
      ```

      ここで、<server_name> は Apache 構成時に作成したサーバー・インスタンスの名前です。たとえば、サーバー名として xsocks を選んだ場合、次のように入力します。

      ```
      wrklnk '/www/xsocks/conf/httpd.conf''
      ```

b. 2 を押してこのファイルを編集します。

c. ファイルの末尾に、以下の行を挿入します。

```
<Location /xsock>
   AuthName "X Socket"
   AuthType Basic
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
   Require valid-user
   order allow,deny
   allow from all
</Location>
```

d. ファイルを保管し、編集セッションを終了します。

次に実行する内容:

Web ブラウザーにおける Xsocket ツールのテスト

Web ブラウザーにおける Xsocket ツールのテスト

構成ファイルへの手動での更新を完了すると、ブラウザーで Xsocket ツールをテストできるようになります。

1. サーバー・インスタンスを開始するには、コマンド行で以下のコマンドを入力します。

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<server_name>)
   ```

   ここで、<server_name> は Apache 構成時に作成したサーバー・インスタンスの名前です。

   注: 多少時間がかかります。

2. コマンド行インターフェースから WRKACTJOB コマンドを発行して、状況を検査します。server_name のジョブがすべて SIGW 状況にある場合は、次のステップを省略できます。

3. ブラウザーで、次の URL を入力します。

   ```
   http://<system_name>:<port>/xsock/index
   ```

   ここで、<system_name> および <port> は、Apache 構成時に選んだサーバー・インスタンス名およびポート番号です。

4. プロンプトが出される際、サーバーのユーザー名とパスワードを入力してください。 Xsocket の Web クライアントが表示されるはずです。

Xsocket の使用

現在、Xsocket ツールを使用する 2 つの方法があります。固有クライアントからこのツールを使用する方法と、Web ブラウザー内でこのツールを使用する方法です。固有バージョンの Xsocket を使用するには、Xsocket ツールを構成する必要があります。ブラウザー環境におけるこのツールの使用を望む場合には、固有クライアント用に Xsocket ツールを構成することに加え、『Web ブラウザーを使用するよう Xsocket を構成する』のステップも完了する必要があります。このツールの 2 つのバージョンには、類似した概念が多数あります。どちらのツールもソケット呼び出しを対話式で発行でき、どちらのツールでも発行したソケット呼び出しの errno が示されます。ただし、インターフェースは若干異なります。以下の説明は、両方の環境において Xsocket ツールを使用する方法を示します。

注: GSKit セキュア・ソケット API を使用するソケット・プログラムを処理する場合は、このツールの Web バージョンを使用する必要があります。

以下の説明では、それぞれのツールを使用する方法について記述します。

- 固有 Xsocket の使用
- ブラウザー・ベースの Xsocket の使用

固有 Xsocket の使用

このステップを実行する前にネイティブの Xsocket のための『構成』のステップすべてを完了したことを確認してください。固有クライアントで Xsocket を使用するには、以下のステップを完了します。

1. コマンド行で以下のコマンドを発行し、Xsocket ツールが存在するライブラリーをライブラリー・リストに追加します。

   ```
   ADDLIBLE <library-name>
   ```

   ここで、<library-name> は固有 Xsocket の構成時に作成したライブラリーの名前です。たとえば、ライブラリーの名前が MYXSOCKET の場合、次のように入力します。

   ```
   ADDLIBLE MYXSOCKET
   ```

2. コマンド行インターフェースで、次のように入力します。

   ```
   CALL TSOXSOCK
   ```

3. Xsocket ウィンドウが表示されます。このウィンドウでは、メニュー・バーと選択フィールドを使用してすべてのソケット・ルーチンにアクセスできます。このウィンドウは、ソケット関数を選択した後に必ず表示されます。このインターフェースを使用して、すでに存在しているソケット・プログラムを選択できます。新しいソケットを処理するには、以下を実行してください。
   a. ソケット関数のリストで、「**ソケット (socket)**」を選択し、「**実行 (Enter)**」を押します。
   b. 表示される「**socket() プロンプト (socket() prompt)**」ウィンドウで、ソケットに適切なアドレス・ファミリーおよびプロトコルを選択し、「**実行 (Enter)**」を押します。
   c. 「**記述子**」を選択し、「**記述子の選択 (Select descriptor)**」を選択します。

      注: 他のソケット記述子が既に存在する場合、これによって活動状態のソケット記述子のリストが表示されます。

   d. 表示されるリストから、作成したソケット記述子を選択します。

      注: 他のソケット記述子が存在する場合、ツールはソケット関数を最新のソケット記述子に自動的に適用します。

4. ソケット関数のリストから、使用するソケット関数を選択します。ステップ 3c で選択したソケット記述子はこのソケット関数で使用されます。ソケット関数を選択すると、ソケット関数についての特定の情報を指定できる一連のウィンドウが表示されます。たとえば、**connect()** を選択すると、表示されるウィンドウで、アドレス長、アドレス・ファミリー、およびアドレス・データを指定することが必要になります。次いで、選択したソケット関数が、提供した情報を指定されて呼び出されます。ソケット関数で生じるすべてのエラーは、errno としてユーザーに表示されます。

注:

1. Xsocket ツールは、DDS のグラフィカルな表示をサポートしています。したがって、データの入力方法、およびウィンドウ/パネルからの選択方法は、グラフィカルなディスプレイ装置とは非グラフィカルなディスプレイ装置のどちらを使用するかに依存します。たとえば、グラフィカルなディスプレイでは、ソケット関数の選択フィールドはチェック・ボックスとして現れますが、非グラフィカルなディスプレイでは、単一のフィールドが現れます。
2. ソケット上では利用可能なのに、ツールに実装されていない **ioctl()** 要求があることに注意してください。

Web ブラウザーにおける Xsocket の使用

Web ブラウザーで Xsocket ツールを使用する前に、Xsocket のすべての固有構成および必要なすべての Web ブラウザー構成が完了したことを確認してください。 Web ブラウザーで Xsocket ツールを使用するには、以下のステップを完了します。

1. Web ブラウザーで、次のように入力します。

   ```
   http://server-name:2001/
   ```

   ここで server-name は、サーバー・インスタンスを含む iSeries の名前です。
2. 「**管理 (Administration)**」を選択します。
3. 左のナビゲーションから、「**HTTP サーバーの管理 (Manage HTTP Servers)**」を選択します。
4. ご使用のインスタンス名を選択し、「**開始 (Start)**」をクリックします。以下のように入力し、コマンド行からサーバー・インスタンスを開始することもできます。

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<instance_name>)
   ```

   ここで、<instance_name> は Apache 構成で作成した HTTP サーバーの名前です。たとえば、サーバー・インスタンス名に xsocks を使用できます。
5. Xsocket Web アプリケーションにアクセスするには、ブラウザーで以下の URL を入力します。

   ```
   http://<system_name>:<port>/xsock/index
   ```

   ここで、<system_name> は iSeries のマシン名で、<port> は HTTP インスタンスを作成した際に指定したポートです。たとえば、システム名が myiSeries で、HTTP サーバー・インスタンスがポート 1025 で listen している場合、次のように入力します。

   ```
   http://myiSeries:1025/xsock/index
   ```
6. Xsocket ツールが Web ブラウザーにロードされると、既存のソケット記述子を処理したり、新規の記述子を作成したりすることができます。このツールの 2 つのバージョンには、類似した概念が多数あります。どちらのツールもソケット呼び出しを対話式で発行でき、どちらのツールでも発行したソケット呼び出しの errno が示されます。ただし、インターフェースは若干異なります。新しいソケット記述子を作成するには、次のように実行できます。
   a. 「**Xsocket メニュー (Xsocket Menu)**」から、「**ソケット (socket)**」を選択します。
   b. 表示される「**Xsocket 照会 (Xsocket Query)**」ウィンドウで、このソケット記述子に適切なアドレス・ファミリー、ソケット・タイプ、およびプロトコルを選択します。「**実行依頼 (Submit)**」をクリックします。
   c. このページを再ロードすると、新規ソケット記述子が「**ソケット (Socket)**」プルダウン・メニューに表示されます。
   d. 「**Xsocket メニュー (Xsocket Menu)**」から、このソケット記述子に適用する関数呼び出しを選択します。 Xsocket ツールの固有バージョンを使用する場合と同様、ソケット記述子を選択しなかった場合、このツールは関数呼び出しに最新のソケット記述子を自動的に適用します。

Xsocket ツールによって作成されたオブジェクトの削除

Xsocket ツールによって作成されたオブジェクトを削除することが必要となる可能性もあります。インストール・プログラムによって、TSODLT という名前のプログラムが作成されます。このプログラムは、ツールによって作成されたオブジェクトを除去したり (ライブラリーと TSODLT は除く)、Xsocket ツールによって使用されるソース・メンバーを除去したりすることができます。以下の一連のコマンドを使用すれば、これらのオブジェクトを削除できます。

ツールによって使用されるソース・メンバーのみを削除するには、以下のコマンドを入力します。

```
CALL TSODLT   (*YES *NONE)
```

ツールが作成するオブジェクトのみを削除するには、以下のコマンドを入力します。

```
CALL TSODLT   (*NO  library-name)
```

ツールによって作成されるソース・メンバーおよびオブジェクトの両方を削除するには、以下のコマンドを入力します。

```
CALL TSODLT   (*YES library-name)
```

Xsocket のカスタマイズ

ソケット・ネットワーク・ルーチンの追加サポート (たとえば、**inet_addr()**) を追加することによって、Xsocket ツールを変更できます。独自の必要に応えるようにこのツールをカスタマイズする場合は、QUSRTOOL ライブラリーは変更しないことを推奨します。このライブラリーを変更するのではなく、ソース・ファイルを別のライブラリーにコピーし、そこで変更を加えてください。これによって、QUSRTOOL ライブラリーのオリジナルのファイルが保存されるので、将来必要となる場合にそれらのファイルを使用できます。TSOCRT プログラムを使用して、変更後にツールを再コンパイルできます (ソース・ファイルが別のライブラリーにコピーされている場合は、そのライブラリーを使用できるように TSOCRT も変更する必要があります)。ツールを作成する前に、TSODLT プログラムを使用してツール・オブジェクトの古いバージョンを削除してください。

保守容易性ツール

ソケットとセキュア・ソケットの使用は増大しており、e-business アプリケーションおよびサーバーに対応するようになっているので、現在の保守容易性ツールは、この要求に対応していく必要があります。新しい拡張された保守容易性ツールを使用することによって、ソケット・プログラムに対するトレースを実行して、ソケットおよび SSL 対応アプリケーション内のエラーに対するソリューションを見つけることができます。これらのツールは、プログラマーとサポート・センターの担当者が、IP アドレスやポート情報などのソケットの特徴を選択することによって、ソケットの問題点を特定するために役立ちます。

以下の表では、これらのサービス・ツールについてそれぞれ概説します。

表 21. ソケットおよびセキュア・ソケットの保守容易性ツール

保守容易性ツール	説明
LIC トレース・フィルター (TRCINT および TRCCNN)	ソケットの選択トレースを行えるようにします。ソケット・トレースをアドレス・ファミリー、ソケット・タイプ、プロトコル、IP アドレス、およびポート情報に制限できるようになりました。トレースを Socket API の特定のカテゴリーのみに、また SO_DEBUG ソケット・オプション・セットのあるソケットのみに制限することもできます。 V5R2 から、LIC トレースは、スレッド、タスク、ユーザー・プロファイル、ジョブ名、またはサーバー名でフィルターに掛けられるようになりました。
STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR)) によるトレース・ジョブ	V5R2 から、STRTRC コマンドに追加のパラメーターが用意されました。このパラメーターは、他のソケットに関連しないトレース・ポイントから分けられた出力を生成します。この出力には、ソケット操作中にエラーが発生した場合の戻りコードと error 情報が含まれます。詳細は、Information Center の『STRTRC (Start Trace) Command Description』を参照してください。

表 21. ソケットおよびセキュア・ソケットの保守容易性ツール (続き)

操作状態記録装置のトレース	ソケット LIC コンポーネント・トレースに、実行された各ソケット操作の操作状況記録装置のダンプが含まれるようになりました。
関連したジョブ情報	サービス担当者とプログラマーが、接続されたソケットまたは listen 中のソケットに関連したすべてのジョブを検出できるようにします。この情報は、AF_INET または AF_INET6 というアドレス・ファミリーを使用するソケット・アプリケーション用の NETSTAT を使用して表示できます。
NETSTAT 接続状況 (オプション 3)。SO_DEBUG を使用可能にする。	ソケット・アプリケーションで SO_DEBUG ソケット・オプションが設定されているときに、拡張低レベル・デバッグ情報を提供します。
セキュア・ソケット戻りコードおよびメッセージ処理	2 つの SSL_ API により、標準化されたセキュア・ソケット戻りコードメッセージを提示します。これらの 2 つの API は、**SSL_Strerror()** と **SSL_Perror()** です。加えて、**gsk_strerror()** は、GSKit API に同様の機能を提供します。また resolver ルーチンからの戻りコード情報を提供する **hstrerror()** API もあります。
パフォーマンス・データ・コレクション (PDC) トレース・ポイント	アプリケーションからソケットを経て TCP/IP スタックまでのデータ・フローをトレースします。

関連情報

以下にリストされているのは、ソケット・プログラムに関する詳細情報を提供する IBM レッドブック™ (PDF 形式)、 Web サイト、および Information Center のトピックです。どの PDF も、表示または印刷ができます。

IBM レッドブック

- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More
- IBM @server iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements

Request For Comments

- **IPv6**
  - RFC 2553: ″Basic Socket Interface Extensions for IPv6″
  - RFC 2292: ″Advanced Sockets API for IPv6″
- **ドメイン・ネーム・システム**
  - RFC 1034: ″ドメイン・ネーム - 概念および機能″ (RFC 1034: ″Domain names - concepts and facilities″)
  - RFC 1035: ″ドメイン・ネーム - インプリメンテーションおよび仕様″ (RFC 1035: ″Domain names - implementation and specification″)

- RFC 2136: ″ドメイン・ネーム・システムにおける動的更新 (DNS UPDATE)″ (RFC 2136: ″Dynamic Updates in the Domain Name System (DNS UPDATE)″)
- RFC 2181: ″DNS 仕様の説明″ (RFC 2181: ″Clarifications to the DNS Specification″)
- RFC 2308: ″DNS QUERY の負のキャッシュ (DNS NCACHE)″ (″RFC 2308: ″Negative Caching of DNS Queries (DNS NCACHE)″)
- RFC 2845: ″Secret Key Transaction Authentication for DNS (TSIG)″

- **Secure Sockets Layer/Transport Layer Security**
  - RFC 2246: ″TLS プロトコル バージョン 1.0 ″ (RFC 2246: ″The TLS Protocol Version 1.0 ″)

他の Web リソース

- Technical Standard: Networking Services (XNS), Issue 5.2 Draft 2.0

コードの特記事項情報

本書には、プログラミングの例が含まれています。

IBM は、お客様に、すべてのプログラム・コードのサンプルを使用することができる非独占的な著作使用権を許諾します。お客様は、このサンプル・コードから、お客様独自の特別のニーズに合わせた類似のプログラムを作成することができます。

すべてのサンプル・コードは、例として示す目的でのみ、IBM により提供されます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

ここに含まれるすべてのプログラムは、「現存するままの状態」で提供され、いかなる保証も適用されません。商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任の保証の適用も一切ありません。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、 IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、 IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

商標

以下は、IBM Corporation の商標です。

Anynet
C/400
IBMiSeries
Language Environment

OS/400
OS/400
Redbooks

Microsoft®、 Windows®、 Windows NT、および Windows ロゴは、 Microsoft Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

資料に関するご使用条件

お客様がダウンロードされる資料につきましては、以下の条件にお客様が同意されることを条件にその使用が認められます。

個人使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

商業的使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任なしで提供されます。

これらの資料の著作権はすべて、IBM Corporation に帰属しています。

お客様が、このサイトから資料をダウンロードまたは印刷することにより、これらの条件に同意されたものとさせていただきます。

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12