IBM

iSeries

套接字编程

IBM

eserver

iSeries

套接字编程

© Copyright International Business Machines Corporation 2001, 2002. All rights reserved.

目录

第 1 部分 套接字编程 1

| 第 1 章 V5R2 的新增内容 3

第 2 章 打印本主题 5

| 第 3 章 套接字编程的先决条件 7

第 4 章 套接字是如何工作的? 9

第 5 章 套接字特征 11
套接字地址结构 12
套接字地址系列 13
AF_INET 地址系列 13
| AF_INET6 地址系列 14
AF_UNIX 地址系列 15
AF_UNIX_CCSID 地址系列 15
AF_TELEPHONY 地址系列 16
套接字类型 17
套接字协议 17

第 6 章 基本套接字设计 19
创建面向连接的套接字 19
| 示例：面向连接的服务器 20
| 示例：面向连接的客户机 23
创建无连接的套接字 26
示例：无连接的服务器 27
示例：无连接的客户机 29
设计使用地址系列的应用程序 31
使用 AF_INET 地址系列 31
| 使用 AF_INET6 地址系列 31
| 使用 AF_UNIX 地址系列 32
使用 AF_UNIX_CCSID 地址系列 38
使用 AF_TELEPHONY 地址系列 45

第 7 章 套接字概念 51
异步 I/O 51
安全套接字 53
| Global Secure ToolKit（GSKit）API 53
| SSL_ API 55
| 安全套接字 API 错误代码消息 56
客户机 SOCKS 支持 58
线程安全 60
非阻塞 I/O 61
信号 61
IP 多点广播 63
文件数据传输 — send_file() 和 accept_and_recv() 63
带外数据 63
I/O 多路复用 — select() 64
套接字网络函数 65

© Copyright IBM Corp. 2001, 2002

域名系统（DNS）支持 65
| 环境变量 66
| 数据高速缓存 67
Berkeley Software Distributions（BSD）兼容性 67
| UNIX 98 兼容性 69
在进程间传递描述符 — sendmsg() 和 recvmsg() 71

| **第 8 章 套接字方案: 创建应用程序以接受 IPv4 和 IPv6 客户机. 75**
| 示例: 接受来自 IPv6 和 IPv4 客户机的连接. 76
| 示例: IPv4 或 IPv6 客户机 80

第 9 章 套接字应用程序设计建议 85

第 10 章 示例: 套接字应用程序设计 87
| 示例: 面向连接的设计 87
| 示例: 编写迭代服务器程序 88
| 示例: 使用 spawn() API 创建子进程 91
| 示例: 在进程间传递描述符 96
| 示例: 使用多 accept() API 处理入局请求 103
| 示例: 一般客户机 107
示例: 使用异步 I/O 109
示例: 建立安全连接 115
| 示例: 使用异步数据接收的 GSKit 安全服务器 116
| 示例: 使用异步握手的 GSKit 安全服务器 125
| 示例: 使用 Global Secure ToolKit（GSKit）API 建立安全客户机. 134
示例: 使用 SSL_ API 建立安全服务器 140
示例: 使用 SSL_ API 建立安全客户机 145
示例: 对线程安全网络例程使用 gethostbyaddr_r(). 148
示例: 非阻塞 I/O 和 select() 149
示例: 将信号与阻塞套接字 API 配合使用 155
示例: 使用多点广播 158
示例: 发送多点广播数据报. 160
示例: 接收多点广播数据报. 161
示例: 更新和查询 DNS 163
| 示例: 使用 send_file() 和 accept_and_recv() API 传输文件数据 166
| 示例: 使用 accept_and_recv() 和 send_file() API 发送文件内容 167
| 示例: 客户机对文件的请求. 170

| **第 11 章 Xsockets 工具 173**
| 配置 Xsockets 173
| 本地 Xsocket 安装创建的内容. 174
| 将 Xsockets 配置为使用 Web 浏览器 176
| 配置 HTTP Server（基于 Apache）. 176
| 配置 Tomcat 177
| 更新配置文件. 177
| 在 Web 浏览器中测试 Xsockets 工具 179
| 使用 Xsockets 179
| 使用本地 Xsockets 179
| 在 Web 浏览器中使用 Xsockets 180
| 删除 Xsocket 工具创建的对象. 181
| 定制 Xsockets 181

第 12 章 可服务性工具. 183

第 13 章 相关信息 185
第 14 章 代码不保证声明信息 187

第 1 部分 套接字编程

套接字是一个通信连接点（端点），您可在网络中对其命名和寻址。使用套接字的进程可驻留在同一系统上或不同网络的不同系统上。套接字对于独立应用程序和网络应用程序都很有用。套接字允许您在同一机器上或通过网络在进程之间交换信息，将工作分配至最有效的机器，并能够很容易的访问中央数据。套接字应用程序接口（API）是 TCP/IP 的网络标准。很多操作系统都支持套接字 API。OS/400 套接字支持多种传输和联网协议。套接字系统函数和套接字网络函数是线程安全的。

套接字编程显示如何使用套接字 API 来建立远程线程与本地线程之间的通信链路。使用“集成语言环境”（ILE）C 的程序员可使用该信息开发套接字应用程序。还可通过其它 ILE 语言（如 RPG）编写用于套接字 API 的代码。有关 ILE RPG 的更多信息，参见 IBM 红皮书 Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More。

Java 也支持套接字编程接口。有关详细信息，参见“信息中心”中的 Java 主题。

套接字编程主题:

下列主题提供了一些概念、设计建议和示例，以帮助您开发套接字应用程序。参见:

- **V5R2 的新增内容**
  使用此页了解与套接字编程有关的新功能。本主题提供了一些链接，这些链接指向的页面提供有关这些增强功能的更加详细的信息。
- **打印本主题**
  使用此页打印或下载套接字编程信息的“可移植文档格式”（PDF）版本。
- **套接字编程的先决条件**
  本主题讨论在使用套接字 API 编写应用程序之前必须完成的必需任务。
- **基本套接字设计**
  本主题提供最基本的套接字类型的示例程序的概述。使用指向举例说明基本套接字设计策略的示例的样本程序的链接。
- **套接字概念**
  本主题描述较高级的套接字概念，如“异步输入／输出”（I/O）和 Global Secure Toolkit（GSKit）。使用本主题中的链接来复查与这些概念相关联的样本程序。
- **套接字方案: 创建应用程序以接受 IPv4 和 IPv6 客户机**
  本主题描述想要使用 AF_INET6 地址系列的典型情况。这一地址系列是 V5R2 的新增内容，提供对网际协议 V6（IPv6）的支持。IPv6 支持 128 位 IP 地址。本主题还提供了规划信息和指向示例程序的链接，您可以使用这些示例程序来实现使用 AF_INET6 地址系列的套接字应用程序。
- **套接字应用程序设计建议**
  本主题提供设计更有效的套接字应用程序的提示。
- **示例: 套接字应用程序设计**
  本主题提供可用来创建套接字应用程序的示例套接字程序。

  注: 此信息包含示例代码。有关使用这些样本程序的详细信息，参见代码不保证声明信息。
- **Xsockets 工具**
  本主题提供 Xsockets 工具的描述，套接字程序员可使用 Xsockets 工具来帮助开发套接字应用程序。使用本主题中的链接阅读有关安装和使用该工具的指示信息。

© Copyright IBM Corp. 2001, 2002

- **可服务性工具**
  本主题提供套接字的可服务性工具的描述。
- **相关信息**
  本主题提供其它套接字信息的链接和描述。

第 1 章 V5R2 的新增内容

本主题的重点放在针对 iSeries 套接字的增强功能上。套接字编程信息包含基本和高级概念，同时还提供了样本程序和设计建议。此外，为高级和初级套接字程序员重新组织了套接字编程主题的结构。它还包含一些方案，这些方案为应用程序员提供有关使用特定示例程序的规范。以下列表描述了一些新的 iSeries 套接字增强功能:

IPv6 支持

这是本发行版的新增内容，程序员现在可以使用新的 AF_INET6 地址系列编写那些使用 IPv6 地址的应用程序。除了 AF_INET6 地址系列之外，现有 API 已更改为支持新的地址系列，同时还添加了新的 API 并定义了新的结构。

- AF_INET6 地址系列
  本主题描述新的地址系列及其地址结构。
- 方案：创建应用程序以接受 IPv4 和 IPv6 客户机
  本主题描述以下客户情况：程序员打算设计并编写使用 AF_INET6 地址系列的应用程序以接受 IPv6 连接。本主题包含样本程序，程序员可更改这些程序以满足他们自己的需要。

X/Open Single UNIX® 规范兼容性

OS/400 套接字提供与 X/Open Single UNIX 规范兼容的支持。OS/400 套接字一直在更改结构、类型定义和功能原型以与这些规范相匹配。程序员可选择使用缺省 OS/400 套接字（这些套接字是基于 Berkeley Software Distributions (BSD) 4.3 套接字的）或指定 _XOPEN_SOURCE 宏以选择 UNIX 98 兼容接口。

- UNIX 98 兼容性
  本主题描述 UNIX 98 与基本 OS/400 套接字之间的差别和兼容性问题。

“安全套接字层”性能改进

这是本发行版的新增内容，包括对加快安全连接速度性能方面的一些改进。目前应用程序员可通过 iSeries 上的三个单独的接口访问 SSL 支持:

1. Global Secure Toolkit（GSKit）API
2. SSL_ API
3. Java SSL 接口

Xsocket 工具更新

Xsockets 是本发行版的新增内容，它是一种交互式工具，是随 QUSRTOOL 一起交付的，现在可通过 Web 浏览器界面来访问。已经添加了新的指示信息，以显示如何将该工具设置为使用 Web 浏览器。还可从命令行界面使用该工具；但是，只能从此新界面访问新增功能，如 GSKit。有关这些更改的详细信息，参见以下主题:

- Xsockets 工具

除去对 IPX/SPX 和 AF_NS 地址系列的支持

就 V5R2 而言，IBM 不再支持 IPX 和 SPX 协议。因此，将在 V5R2 中禁用 AF_NS 地址系列。当应用程序试图使用 **socket()** API 打开 AF_NS（IPX/SPX）时将生成 -1 返回码，并且错误号设置为 EAFNOSUPPORT。

© Copyright IBM Corp. 2001, 2002

第 2 章 打印本主题

可查看或下载本文档的“可移植文档格式”（PDF）版本以便查看或打印。必须安装有 Adobe® Acrobat® Reader 才能查看 PDF 文件。可从 http://www.adobe.com/prodindex/acrobat/readstep.html 下载它的副本。

要查看或下载 PDF 版本，选择套接字编程。（444 KB，132 页）

要将 PDF 保存在工作站上以便查看或打印:
1. 在浏览器中打开 PDF（单击以上链接）。
2. 在浏览器的菜单中，单击**文件**。
3. 单击**另存为...**
4. 浏览至想要保存 PDF 的目录。
5. 单击**保存**。

第 3 章 套接字编程的先决条件

在编写套接字应用程序之前，必须先完成下列步骤:

编译器需求

1. 安装 QSYSINC 库。此库提供编译套接字应用程序时需要的必要头文件。
2. 安装 C 编译器许可程序（5722–CX2）。

AF_INET 和 AF_INET6 地址系列的需求

除了编译器需求之外，还必须完成下列步骤:

1. 规划 TCP/IP
2. 安装 TCP/IP
3. 首次配置 TCP/IP
4. 配置用于 IPv6 的 TCP/IP。此步骤是可选的。如果打算编写使用 AF_INET6 地址系列的应用程序，则为 TCP/IP 配置 IPv6 接口。

“安全套接字层”（SSL）和 Global Secure Toolkit（GSKit）API 的需求

除了编译器和 AF_INET 与 AF_INET6 地址需求之外，还必须完成下列任务才能使用安全套接字:

1. 安装和配置“数字证书管理器”许可程序（5722 — SS1 选项 34）。有关详细信息，参见“信息中心”中的数字证书管理。
2. 安装“加密访问提供程序”许可程序（5722 - AC3）。
3. 如果想要将 SSL 与加密硬件配合使用，需要安装和配置 2058 Cryptographic Accelerator for iSeries 或 4758 PCI Cryptographic Coprocessor for iSeries。2058 Cryptographic Accelerator 允许您将 SSL 加密处理从操作系统卸载到卡。有关 2058 Cryptographic Accelerator 及其功能部件的完整描述，参见 2058 Cryptographic Accelerator for iSeries。可使用 4758 Cryptographic Coprocessor 进行 SSL 加密处理；但是，与 2058 不同的是，此卡提供了更多加密功能，如加密和解密密钥。有关此卡的功能部件和配置步骤，参见 4758 PCI Cryptographic Coprocessor for iSeries。

AF_TELEPHONY 地址系列的需求

如果打算设计 AF_TELEPHONY 套接字（它使用电话线），除了一般需求之外，还必须完成下列步骤。

1. 规划 IDSN 服务以确定您对 ISDN 连接的需求。
2. 根据规划信息配置 IDSN 环境。

© Copyright IBM Corp. 2001, 2002

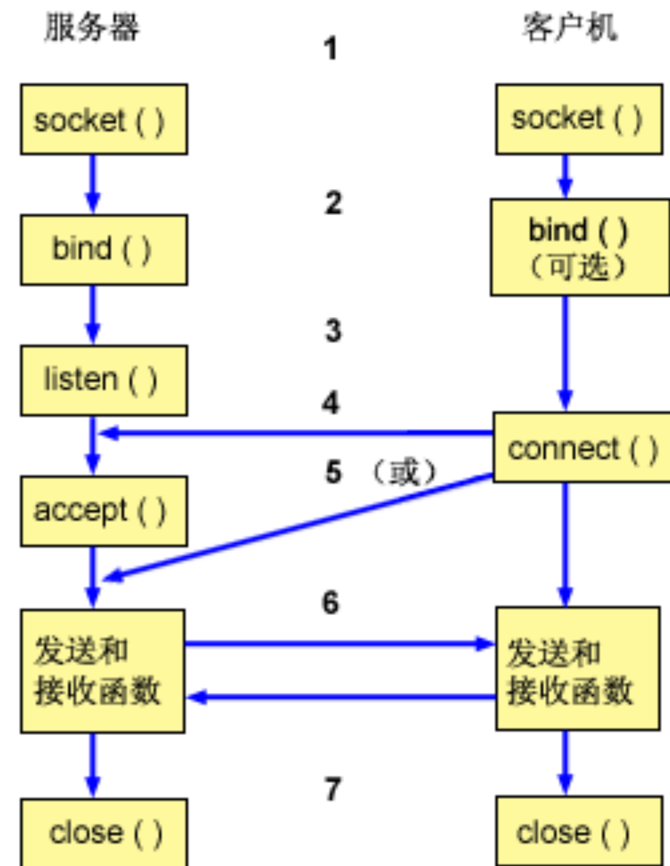
第 4 章 套接字是如何工作的?

套接字通常用于客户机/服务器交互作用。典型系统配置是将服务器放在一台机器上，而将客户机放在其它机器上。客户机与服务器相连接，交换信息，然后断开连接。

套接字有一个典型的事件流。在面向连接的客户机至服务器模型中，服务器进程上的套接字等待来自客户机的请求。为此，服务器先建立（绑定）一个地址，客户机可使用此地址来查找服务器。建立地址之后，服务器等待客户机请求服务。当客户机通过套接字连接至服务器时，就会发生客户机至服务器数据交换。服务器执行客户机的请求并将应答发送回客户机。

注: 目前，IBM 支持大部分套接字 API 的两个版本。缺省 OS/400 套接字使用 Berkeley Socket Distribution (BSD) 4.3 结构和语法。基本 OS/400 套接字与 BSD 4.3 之间的差别在 Berkeley Socket Distribution（BSD）兼容性中作了概述。其它版本的套接字使用与 BSD 4.4 和 UNIX 98 编程接口规范相兼容的语法和结构。程序员可以将 _XOPEN_SOURCE 宏指定为使用 UNIX98 兼容接口。有关这些 API 及结构性差别的描述，参见 UNIX 98 兼容性。

下图显示面向连接的套接字会话的典型事件流（及发出的函数序列）。图的下面是每个事件的说明。



面向连接的套接字的典型事件流:

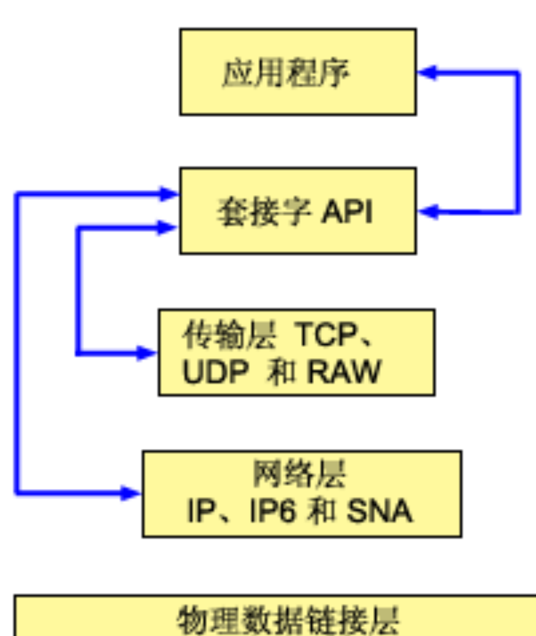
1. **socket()** 函数创建通信端点并返回表示该端点的套接字描述符。
2. 如果应用程序有套接字描述符，它会将唯一名称绑定至该套接字。服务器必须绑定一个名称，才能通过网络访问该服务器。

© Copyright IBM Corp. 2001, 2002

3. **listen()** 函数指示接受客户机连接请求的意愿。在对套接字发出 **listen()** 时，该套接字不能主动地启动连接请求。在使用 **socket()** 函数分配套接字且 **bind()** 函数将名称绑定至该套接字之后，发出 **listen()** API。在发出 **accept()** 函数之前，必须发出 **listen()** 函数。
4. 客户机应用程序对流套接字使用 **connect()** 函数来建立与服务器的连接。
5. 服务器应用程序使用 **accept()** 函数接受客户机连接请求。服务器必须先成功发出 **bind()** 和 **listen()** 函数，才能发出 **accept()**。
6. 在流套接字之间（客户机与服务器之间）建立连接后，可使用任何套接字 API 数据传输函数。客户机和服务器有多种数据传输函数可以选择，如 **send()**、**recv()**、**read()**、**write()** 及其它函数。
7. 当服务器或客户机想要停止操作时，会发出 **close()** 函数以释放套接字获取的所有系统资源。

注:

套接字 API 在通信模型中位于应用层和传输层之间。套接字 API 并非通信模型中的某一层。套接字 API 允许应用程序与典型通信模型的传输层或网络层进行交互。下图中的箭头显示套接字的位置以及套接字提供的通信层。



通常，网络配置不允许安全的内部网络与低安全的外部网络之间的连接。但是，可启用套接字以便与在防火墙（非常安全的主机）外部的系统上运行的服务器程序通信。

套接字也是 IBM 的“多协议传输连网”（MPTN）体系结构的 AnyNet 实现的一部分。MPTN 体系结构提供了通过附加传输网络对传输网络进行操作的能力以及在不同类型的传输网络间连接应用程序的能力。

| “信息中心”允许您以几种方式访问 API 参考信息。主题套接字 API 提供套接字函数和结构的概述。如果想
| 要搜索特定 API 或搜索 API 类别，API 信息提供了交互式 API 查找程序来协助您工作。

第 5 章 套接字特征

套接字共享下列特征:

- 套接字由整数表示。该整数称为**套接字描述符**。
- 只要进程保留指向套接字的打开链接，该套接字就一直存在。
- 可命名套接字并使用它与通信域中的其它套接字进行通信。
- 当服务器接受来自套接字的连接或与套接字交换消息时，套接字会执行通信。
- 可成对创建套接字（仅适用于 AF_UNIX 地址系列中的套接字）。

套接字提供的连接可以是面向连接的或无连接的。**面向连接的**通信暗示已建立连接，接下来会在程序之间建立对话。提供服务（服务器程序）的程序建立可用套接字，将启用该套接字以接受入局连接请求。服务器可随意地对它提供的服务指定名称，这允许客户机标识在何处获取该服务以及如何连接至该服务。服务（客户机程序）的客户机必须请求服务器程序的服务。客户机通过连接至相异名称或与服务器程序指定的相异名称相关联的属性来完成此任务。这与拨电话号码（标识符）并与提供服务的另一方（管道工）建立连接相类似。当呼叫的接收方（服务器，在本示例中为管道工）对电话作出应答时，就建立了连接。管道工验证您是否到达正确的一方，并且只要双方需要，连接会一直保持活动状态。

无连接的通信暗示没有建立连接，也不会在此基础上进行任何对话或数据传输。而是服务器程序指定一个名称来标识获取它的位置（非常类似于邮箱）。如果向信箱发一封信，您不能绝对保证接收方能收到信。您可能需要等待信件的回复。没有活动的实时连接可供数据交换使用。

如何确定套接字特征

当应用程序创建带有 **socket()** 函数的套接字时，它必须通过指定下列参数来标识该套接字:

- 套接字地址系列确定套接字的地址结构的格式。本主题包含每个地址系列的地址结构的示例。有关套接字地址的结构的一般定义，参见套接字地址结构。
- 套接字类型确定套接字的通信的期望格式。
- 支持套接字的协议确定套接字使用的协议。

这些参数或特征定义套接字应用程序以及它与其它套接字应用程序进行交互的方式。根据套接字的地址系列，可以选择不同的套接字类型和协议。下表显示相应的地址系列及其关联套接字类型和协议:

表 1. 套接字特征的总结

地址系列	套接字类型	套接字协议
AF_UNIX	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP 和 ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6 和 ICMP6
AF_TELEPHONY	SOCK_STREAM	N/A
AF_UNIX_CCSID	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A

© Copyright IBM Corp. 2001, 2002

除了这些套接字特征或参数之外，还在随 QSYSINC 库一起交付的网络例程和头文件中定义了常量值。有关头文件的描述，参见“信息中心”中的套接字 API 主题中列示的个别 API。每个 API 在 API 描述的用法部分中列示其相应头文件。

套接字网络例程允许套接字应用程序从 DNS、主机、协议、服务和网络文件获取信息。有关这些例程的描述，参见套接字网络例程。

套接字地址结构

套接字使用 **sockaddr** 地址结构来发送和接收地址。此结构不需要套接字 API 就可以组织寻址格式。目前 OS/400 支持 Berkeley Software Distributions（BSD）4.3 和 X/Open Single Unix 规范（UNIX 98）。基本 OS/400 API 使用 BSD 4.3 结构和语法。可通过将 _XOPEN_SOURCE 宏定义为值 520 或以上来选择 UNIX 98 兼容接口。使用的每个 BSD 4.3 套接字地址结构将会有等效的 UNIX 98 结构。

表 2. BSD 4.3 与 UNIX 98/BSD 4.4 套接字地址结构的比较

BSD 4.3 结构	**BSD 4.4/UNIX 98 兼容结构**

`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. 地址结构

地址结构字段	**定义**
sa_len	此字段包含 UNIX 98 规范的地址长度。 **注：** sa_len 字段仅供 BSD 4.4 兼容性使用。不一定要使用此字段，即使是在使用 BSD 4.4/UNIX 98 兼容性时。该字段在输入地址上会被忽略。
sa_family	此字段定义地址系列。此值是在 **socket()** 调用上对地址系列指定的。
sa_data	此字段包含为保存地址本身而保留的 14 字节。 **注：** 14 字节的 sa_data 长度是地址的占位符。该地址可能超出此长度。该结构是一般结构，原因是它未定义地址的格式。地址的格式是由为其创建套接字的传输类型定义的。每个传输提供程序将在相似的地址结构中定义精确格式以满足其特定寻址需求。传输是由 **socket()** API 上的协议参数值标识的。

表 3. 地址结构 (续)

sockaddr_storage	声明用于任何地址系列地址的存储器。此结构大到足以放下任何特定于协议的结构，并向其看齐。然后可将其强制转型为 sockaddr 结构以便在 API 上使用。sockaddr_storage 的 ss_family 字段将总是向任何特定于协议的结构的系列字段看齐。

套接字地址系列

socket() 上的地址系列参数确定要在套接字函数上使用的地址结构的格式。地址系列协议提供应用程序数据从一个应用程序到另一个应用程序（或从同一台机器上的一个进程到另一个进程）的网络传输。应用程序在套接字的协议参数上指定网络传输提供程序。

socket() 函数上的地址系列参数（address_family）指定在套接字函数上使用的地址结构。下列主题描述每个地址系列、它们的用法、它们的相关协议以及相关结构的示例:

- AF_INET 地址系列
- AF_INET6 地址系列
- AF_UNIX 地址系列
- AF_UNIX_CCSID 地址系列
- AF_TELEPHONY 地址系列

AF_INET 地址系列

此地址系列提供在同一系统或不同系统上运行的进程间的“进程间通信”。AF_INET 套接字的地址包括 IP 地址和端口号。可将 AF_INET 套接字的 IP 地址指定为 IP 地址（如 130.99.128.1）或 32 位格式（如 X’82638001’）。

对于使用网际协议 V4（IPv4）的套接字应用程序，AF_INET 地址系列使用 **sockaddr_in** 地址结构。在使用 _XOPEN_SOURCE 宏时，AF_INET 地址结构更改为与 BSD 4.4/UNIX 98 规范兼容。对于 sockaddr_in 地址结构，这些差别在下表中作了总结:

表 4. BSD 4.3 与 BSD 4.4/UNIX 98 在 sockaddr_in 地址结构上的差别

<table>
<tr><th>BSD 4.3 sockaddr_in 地址结构</th><th>BSD 4.4/UNIX 98 sockaddr_in 地址结构</th></tr>
<tr><td><pre>struct sockaddr_in {
  short   sin_family;
  u_short   sin_port;
  struct in_addr sin_addr;
  char   sin_zero[8];
};</pre></td><td><pre>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];
};</pre></td></tr>
</table>

表 5. AF_INET 地址结构

地址结构字段	定义
sin_len	此字段包含 UNIX 98 规范的地址长度。 **注:** sin_len 字段仅供 BSD 4.4 兼容性使用。不一定要使用此字段，即使是在使用 BSD 4.4/UNIX 98 兼容性时。该字段在输入地址上会被忽略。
sin_family	此字段包含地址系列，当使用 TCP 或 UDP 时，它总是为 AF_INET。

表 5. AF_INET 地址结构 (续)

sin_port	此字段包含端口号。
sin_addr	此字段包含因特网地址。
sin_zero	此字段是保留的。将此字段设置为十六进制零。

有关使用 AF_INET 的信息和使用 AF_INET 地址系列的样本程序的信息，参见使用 AF_INET 地址系列。

AF_INET6 地址系列

此地址系列提供对网际协议 V6（IPv6）的支持。AF_INET6 地址系列使用 128 位（16 字节）地址。这些地址的基本体系结构中的 64 位用作网络号，另外的 64 位用作主机号。可将 AF_INET6 地址指定为 x:x:x:x:x:x:x:x，其中“x”表示地址的 8 个 16 位的十六进制值。例如，有效地址如下所示：FEDC:BA98:7654:3210:FEDC:BA98:7654:3210。

对于使用 TCP、UDP 或 RAW 的套接字应用程序，AF_INET6 地址系列使用 **sockaddr_in6** 地址结构。如果使用 _XOPEN_SOURCE 宏实现 BSD 4.4/UNIX 98 规范，则此地址结构会更改。对于 sockaddr_in6 地址结构，这些差别在下表中作了总结：

表 6. BSD 4.3 与 BSD 4.4/UNIX 98 在 sockaddr_in6 地址结构上的差别

BSD 4.3 sockaddr_in6 地址结构	BSD 4.4/UNIX 98 sockaddr_in6 地址结构

`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 地址结构

地址结构字段	定义
sin6_len	此字段包含 UNIX 98 规范的地址长度。 **注：** sin6_len 字段仅供 BSD 4.4 兼容性使用。不一定要使用此字段，即使是在使用 BSD 4.4/UNIX 98 兼容性时。该字段在输入地址上会被忽略。
sin6_family	此字段指定 AF_INET6 地址系列。
sin6_port	此字段包含传输层端口。
sin6_flowinfo	此字段包含两部分信息：流量类和流标号。 **注：** 此字段目前不受支持，且应设置为零以获取向上兼容性。
sin6_addr	此字段指定 IPv6 地址。
sin6_scope_id	此字段针对 **sin6_addr** 字段中装入的地址作用域相应地标识一组接口。 **注：** 此字段目前不受支持，且应设置为零以获取向上兼容性。

AF_UNIX 地址系列

此地址系列提供在同一系统上使用套接字 API 进行的“进程间通信”。该地址实际上是指向文件系统中某项的路径名。可在根目录或任何开放式文件系统中创建套接字，但诸如 QSYS 或 QDOC 之类的文件系统除外。该程序必须将 AF_UNIX 即 SOCK_DGRAM 套接字绑定至一个名称以接收所有返回的数据报。此外，在关闭套接字时，该程序必须使用 **unlink()** API 显式除去该文件系统对象。

带有地址系列 AF_UNIX 的套接字使用 **sockaddr_un** 地址结构。如果使用 _XOPEN_SOURCE 宏实现 BSD 4.4/UNIX 98 规范，则此地址结构会更改。对于 sockaddr_un 地址结构，这些差别在下表中作了总结:

表 8. BSD 4.3 与 BSD 4.4/UNIX 98 在 sockaddr_un 地址结构上的差别

BSD 4.3 sockaddr_un 地址结构	BSD 4.4/UNIX 98 sockaddr_un 地址结构

`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 地址结构

地址结构字段	定义
sun_len	此字段包含 UNIX 98 规范的地址长度。 **注:** sun_len 字段仅供 BSD 4.4 兼容性使用。不一定要使用此字段，即使是在使用 BSD 4.4/UNIX 98 兼容性时。该字段在输入地址上会被忽略。
sun_family	此字段包含地址系列。
sun_path	此字段包含指向文件系统中某项的路径名。

对于 AF_UNIX 地址系列，协议规范不适用，原因是没有涉及协议标准。两个进程使用的通信机制是特定于机器的。

有关使用 AF_UNIX 的信息，参见使用 AF_UNIX 地址系列，它提供了使用此地址系列的样本程序。

AF_UNIX_CCSID 地址系列

AF_UNIX_CCSID 系列与 AF_UNIX 地址系列兼容，但有一些限制。它们要么都是无连接的，要么都是面向连接的，并且没有任何外部通信函数会连接这两个进程。差别在于带有地址系列 AF_UNIX_CCSID 的套接字使用 **sockaddr_unc** 地址结构。此地址结构类似于 sockaddr_un，但它通过使用 **Qlg_Path_Name_T** 格式允许使用 UNICODE 或任何 CCSID 格式的路径名。参见“信息中心”中的路径名格式。

但是，由于 AF_UNIX 套接字可能从 AF_UNIX 地址结构中的 AF_UNIX_CCSID 套接字返回路径名，所以路径大小是受限制的。AF_UNIX 仅支持使用 126 个字符，所以 AF_UNIX_CCSID 也将限制为使用 126 个字符。

用户可能不会在单个套接字上交换 AF_UNIX 和 AF_UNIX_CCSID 地址。如果在 **socket()** 调用上指定了 AF_UNIX_CCSID，在以后的 API 调用上，所有地址都必须是 **sockaddr_unc**。

```
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 地址结构*

地址结构字段	定义
sunc_family	此字段包含地址系列，它总是为 AF_UNIX_CCSID。
sunc_format	此字段包含路径名格式的两个定义值: • **SO_UNC_DEFAULT** 使用集成文件系统路径名的当前缺省 CCSID 指示宽路径名。忽略 sunc_qlg 字段。 • **SO_UNC_USE_QLG** 指示 **sunc_qlg** 字段定义路径名的格式和 CCSID。
sunc_zero	此字段是保留的。将此字段设置为十六进制零。
sunc_qlg	此字段指定路径名格式。
sunc_path	此字段包含路径名。最多包含 126 个字符，这些字符可以是单字节或双字节的。它可以包含在 sunc_path 字段中，或单独指定并由 sunc_path 指向它。该格式是由 sunc_format 和 sunc_qlg 确定的。

有关 AF_UNIX_CCSID 套接字的更多信息，参见使用 AF_UNIX_CCSID 地址系列，它提供了一个样本程序。

AF_TELEPHONY 地址系列

AF_TELEPHONY 地址系列允许用户通过使用标准套接字 API 的 ISDN 电话网络拨号或应答电话呼叫。在此域中组成连接端点的套接字实际上是电话呼叫的被呼叫方和呼叫方。此地址系列的地址由 40 位电话号码表示。通常会在开发传真支持时使用此地址系列。

系统仅支持 AF_TELEPHONY 套接字作为面向连接的套接字，其套接字类型为 SOCK_STREAM。电话域套接字中的连接并不比基本电话连接可靠。如果想要保证传送，必须使用提供这些服务的应用程序，如使用此地址系列的传真应用程序。

带有 AF_TELEPHONY 地址系列的套接字使用 **sockaddr_tel** 地址结构:

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

电话地址包括 2 字节长度，后跟最多 40 位的电话号码（0 至 9）。

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

表 11. *AF_TELEPHONY 地址结构*

地址结构字段	定义
stel_family	此字段包含地址系列。
stel_addr	此字段包含电话地址。
stel_zero	此字段是保留字段。

有关 AF_TELEPHONY 地址系列的更多信息，参见使用 AF_TELEPHONY 地址系列，它提供有关配置环境以便使用 AF_TELEPHONY 地址系列的步骤。

套接字类型

套接字调用上的第二个参数确定套接字的类型。套接字类型标识将对从一台机器至另一台机器或从一个进程至另一个进程的数据传输启用的连接的类型和特征。以下列表描述 iSeries 支持的套接字类型:

流（SOCK_STREAM）
此类型的套接字是面向连接的。通过使用 **bind()**、**listen()**、**accept()** 和 **connect()** 函数来建立端到端连接。SOCK_STREAM 发送数据而不发送错误或重复内容，并按发送次序接收数据。SOCK_STREAM 构建流量控制以避免数据过速。它并未对数据强制记录边界。SOCK_STREAM 将数据视作字节流。在 iSeries 实现中，可通过“传输控制协议”（TCP）、“系统网络体系结构 ”（SNA）、AF_UNIX、AF_UNIX_CCSID 和 AF TELEPHONY 套接字使用流套接字。还可使用流套接字与安全主机（防火墙）外部的系统通信。

| **数据报（SOCK_DGRAM）**
| 在“网际协议”术语中，数据传输的基本单位是**数据报**。它基本上是一个报头，后跟一些数据。数据报套接字
| 是无连接的。它不会使用传输提供程序（协议）建立任何端到端连接。套接字将数据报作为独立的信息包发
| 送，不对传送作任何保证。您可能会丢失数据或使数据重复。数据报到达时可能会次序混乱。数据报的大小被
| 限制为可在单个事务中发送的数据大小。对于某些传输提供程序，每个数据报都可通过网络使用不同的路由。
| 可对此类型的套接字发出 **connect()** 函数。但是，在 **connect()** 函数上，必须指定程序发送至的目标地址和
| 从其接收的目标地址。在 iSeries 实现中，可通过用户数据报协议（UDP）、SNA 以及 AF_UNIX 和
| AF_UNIX_CCSID 地址系列来使用数据报套接字。

| **原始（SOCK_RAW）**
| 此类型的套接字允许直接访问低层协议，如网际协议（IPv4 或 IPv6）和因特网控制报文协议（ICMP 或
| ICMP6）。SOCK_RAW 需要较多的编程经验，原因是需要管理传输提供程序使用的协议头信息。在这一层中，
| 传输提供程序可能会要求目标地址特定于传输提供程序的数据格式和语义。

套接字协议

协议提供从一台机器至另一台机器（或在同一机器内从一个进程至另一个进程）的应用程序数据的网络传输。应用程序在 **socket()** 函数的**协议**参数上指定传输提供程序。

对于 AF_INET 地址系列，允许使用多个传输提供程序。对于同一套接字，SNA、TCP/IP 或 UDP/IP 协议可以同时处于活动状态。ALWANYNET（允许 ANYNET 支持）网络属性允许客户选择是否可对 AF_INET 套接字应用程序使用不同于 TCP/IP 的传输。此网络属性可以是 ***YES** 或 ***NO**。缺省值为 ***NO**。

例如，如果当前状态（缺省状态）为 *NO，则不能使用基于 SNA 传输的 AF_INET。如果仅使用基于 TCP/IP 传输的 AF_INET 套接字，则 ALWANYNET 状态应设置为 *NO 以改进 CPU 利用率。

注: ALWANYNET 网络属性还会影响基于 TCP/IP 的 APPC。

有关 APPC 配置选项的信息，参见配置 APPC、APPN 和 HPR。

基于 TCP/IP 的 AF_INET 套接字还可指定类型 SOCK_RAW，这表示套接字直接与称为网际协议（IP）的网络层通信。TCP 或 UDP 传输提供程序通常与这一层通信。在使用 SOCK_RAW 套接字时，应用程序指定 0 与 255 之间的任何协议（TCP 和 UDP 协议除外）。当机器通过网络进行通信时，此协议号就会流入 IP 头。实际上，应用程序就是传输提供程序，原因是它必须提供 UDP 或 TCP 传输通常情况下提供的所有传输服务。

对于 AF_UNIX、AF_UNIX_CCSID 和 AF_TELEPHONY 地址系列，协议规范并无实际意义，原因是没有涉及协议标准。同一机器上的两个进程间的通信机制是特定于机器的。

第 6 章 基本套接字设计

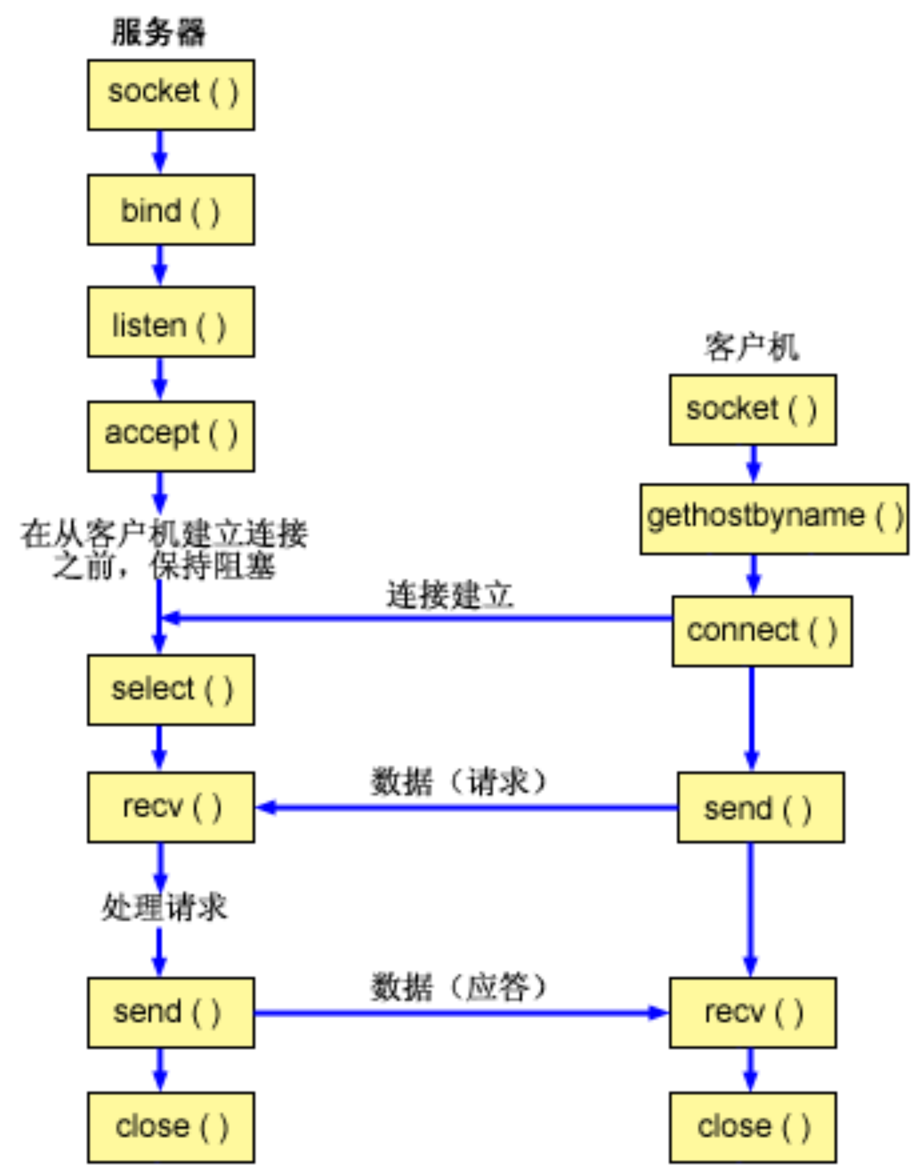
本主题提供使用最基本设计的套接字程序的示例。这些示例为较复杂的套接字设计提供基础。它们实现了上述主题强调的一些基本概念。下列样本程序提供最常见类型的套接字程序的示例。

- 创建面向连接的套接字
- 创建无连接的套接字
- 设计使用地址系列的应用程序

创建面向连接的套接字

| 这些服务器和客户机示例举例说明了为面向连接的协议（如传输控制协议（TCP））编写的套接字 API。

| 下图举例说明了面向连接的协议的套接字 API 的客户机／服务器关系。



套接字事件流：面向连接的服务器

以下套接字调用序列提供图形的描述。它还描述了在面向连接的设计中服务器与客户机应用程序之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：面向连接的服务器使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. **setsockopt()** 函数允许在必需的等待时间到期之前重新启动服务器时重复使用本地地址。
3. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。在此示例中，用户将 s_addr 设置为零，这允许从指定端口 3005 的任何 IPv4 客户机建立连接。
4. **listen()** 允许服务器接受入局客户机连接。在此示例中，储备设置为 10。这表示系统将对 10 个入局连接请求进行排队，然后才开始拒绝入局请求。
5. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
6. **select()** 函数允许进程等待事件发生并在事件发生时唤醒进程。在此示例中，仅当数据可读时，系统才会通知进程。对此选择调用使用 30 秒超时。
7. **recv()** 函数从客户机应用程序接收数据。在此示例中，我们知道客户机将发送超过 250 字节的数据。既然如此，就可以使用 SO_RCVLOWAT 套接字选项指定在所有 250 字节数据都到达之前不要唤醒 **recv()**。
8. **send()** 函数将数据回传至客户机。
9. **close()** 函数关闭所有打开的套接字描述符。

套接字事件流：面向连接的客户机

示例：面向连接的客户机使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. 在客户机示例程序中，如果传递至 **inet_addr()** 函数的服务器字符串不是点分十进制 IP 地址，则假定它是服务器的主机名。在这种情况下，使用 **gethostbyname()** 函数来检索服务器的 IP 地址。
3. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
4. **send()** 函数将 250 字节的数据发送至服务器。
5. **recv()** 函数等待服务器回传 250 字节的数据。在此示例中，我们知道服务器正打算响应刚发送的相同 250 字节。在客户机示例中，250 字节的数据可能分装在不同信息包中到达，所以我们将重复使用 **recv()** 函数直到所有 250 字节数据都到达为止。
6. **close()** 函数关闭所有打开的套接字描述符。

示例：面向连接的服务器

以下代码示例显示如何创建面向连接的服务器。可使用此示例来创建您自己的套接字服务器应用程序。面向连接的服务器设计是套接字应用程序的最常见模型之一。在面向连接的设计中，服务器应用程序创建套接字以接受客户机请求。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例: 面向连接的客户机

以下示例显示在面向连接的设计中如何创建套接字客户机程序以连接面向连接的服务器。服务（客户机程序）的客户机必须请求服务器程序的服务。可使用此代码示例来编写您自己的客户机应用程序。有关使用代码示例的信息，参见代码不保证声明。

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

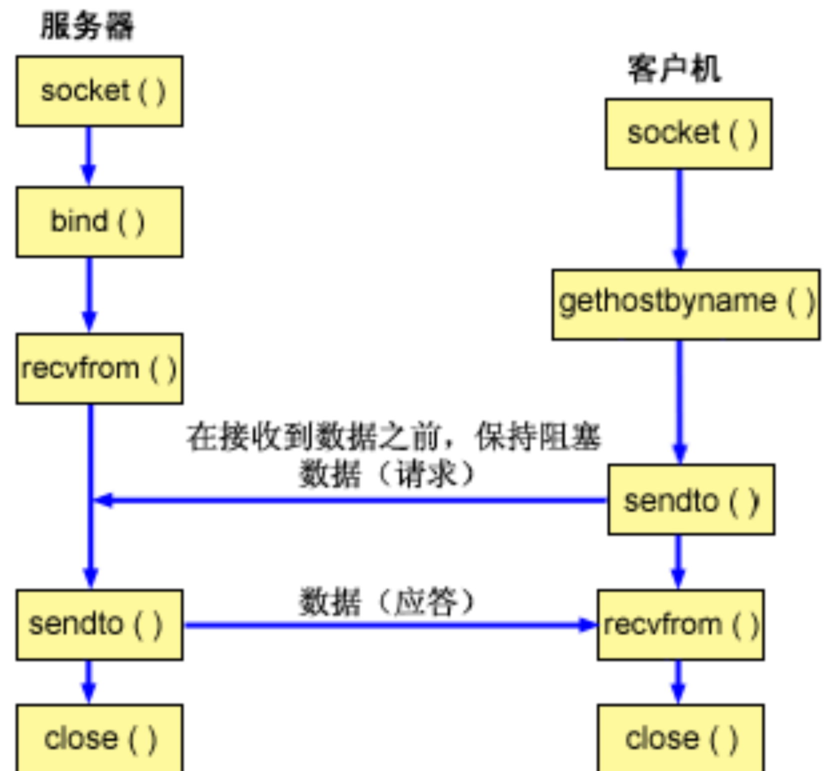
```
      /* server.                                                   */
      /*************************************************************/

      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);
   }
```

创建无连接的套接字

无连接的套接字不会建立连接，也不会在此基础上传输任何数据。而是由服务器应用程序指定其名称，客户机可在该位置发送请求。无连接的套接字使用用户数据报协议（UDP）而不是 TCP/IP。示例：无连接的服务器和示例：无连接的客户机举例说明了为用户数据报协议（UDP）编写的套接字 API。

下图举例说明了无连接的套接字设计的代码示例中使用的套接字 API 中的客户机／服务器关系。



套接字事件流：无连接的服务器

以下套接字调用序列提供图形的描述及下列示例程序。它还描述无连接的设计中的服务器与客户机应用程序之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：无连接的服务器使用以下函数调用序列：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 UDP 传输（SOCK_DGRAM）的 INET（网际协议）地址系列。

2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。在此示例中，用户将 s_addr 设置为零，这表示 UDP 端口 3555 将绑定至系统上的所有 IPv4 地址。
3. 服务器使用 **recvfrom()** 函数来接收该数据。**recvfrom()** 函数无限期等待数据到达。
4. **sendto()** 函数将数据回传至客户机。
5. **close()** 函数结束所有打开的套接字描述符。

套接字事件流：无连接的客户机

示例：无连接的客户机使用以下函数调用序列：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 UDP 传输（SOCK_DGRAM）的 INET（网际协议）地址系列。
2. 在客户机示例程序中，如果传递至 inet_addr() 函数的服务器字符串不是点分十进制 IP 地址，则假定它是服务器的主机名。在这种情况下，使用 **gethostbyname()** 函数来检索服务器的 IP 地址。
3. 使用 **sendto()** 函数将数据发送至服务器。
4. 使用 **recvfrom()** 函数接收从服务器返回的数据。
5. **close()** 函数结束所有打开的套接字描述符。

示例：无连接的服务器

可使用该示例创建您自己的无连接的服务器设计。此示例使用 UDP 创建无连接的套接字服务器程序。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例：无连接的客户机

以下示例显示如何使用 UDP 将无连接的套接字客户机程序连接至服务器。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

设计使用地址系列的应用程序

下列主题提供了举例说明每个套接字地址系列的样本程序:

- 使用 AF_INET 地址系列
- 使用 AF_INET6 地址系列
- 使用 AF_UNIX 地址系列
- 使用 AF_TELEPHONY 地址系列
- 使用 AF_UNIX_CCSID 地址系列

使用 AF_INET 地址系列

AF_INET 地址系列套接字可以是面向连接的（类型 SOCK_STREAM），也可以是无连接的（类型 SOCK_DGRAM）。面向连接的 AF_INET 套接字将 TCP 用作传输协议。无连接的 AF_INET 套接字将 UDP 用作传输协议。在创建 AF_INET 域套接字时，指定 AF_INET 作为套接字程序中的地址系列。AF_INET 套接字还可以使用类型 SOCK_RAW。如果设置了此类型，应用程序会直接连接至 IP 层，而不使用 TCP 或 UDP 传输。

有关将环境设置为使用 AF_INET 地址系列的详细信息，参见套接字编程的先决条件。

有关使用 AF_INET 地址系列的样本程序，参见示例: 面向连接的服务器和示例: 面向连接的客户机。

使用 AF_INET6 地址系列

AF_INET6 套接字提供对网际协议 V6（IPv6）128 位（16 字节）地址结构的支持。程序员可使用 AF_INET6 地址系列编写应用程序以接受 IPv4 或 IPv6 节点的客户机请求或仅接受来自 IPv6 节点的客户机请求。

像 AF_INET 套接字一样，AF_INET6 套接字可以是面向连接的（类型 SOCK_STREAM），也可以是无连接的（类型 SOCK_DGRAM）。面向连接的 AF_INET6 套接字将 TCP 用作传输协议。无连接的 AF_INET6 套接字将 UDP 用作传输协议。在创建 AF_INET6 域套接字时，指定 AF_INET6 作为套接字程序中的地址系列。AF_INET6 套接字还可以使用类型 SOCK_RAW。如果设置了此类型，应用程序会直接连接至 IP 层，而不使用 TCP 或 UDP 传输。有关将环境设置为使用 AF_INET6 地址系列的详细信息，参见套接字编程的先决条件。

IPv6 应用程序与 IPv4 应用程序的兼容性

使用 AF_INET6 地址系列编写的套接字应用程序允许网际协议 V6（IPv6）应用程序使用网际协议 V4（IPv4）应用程序（那些使用 AF_INET 地址系列的应用程序）。此功能部件允许套接字程序员使用映射 IPv4 的 IPv6 地址格式。此地址格式表示以 IPv6 地址形式出现的 IPv4 节点的 IPv4 地址。IPv4 地址被编码到低序 32 位 IPv6 地址中，高序 96 位具有固定前缀 0:0:0:0:0:FFFF。例如，映射 IPv4 的地址可能如下所示:

```
::FFFF:192.1.1.1
```

在指定主机只有 IPv4 地址时，可通过 **getaddrinfo()** 函数自动生成这些地址。

可创建使用 AF_INET6 套接字的应用程序以打开与 IPv4 节点的 TCP 连接。要完成此任务，可将目标位置的 IPv4 地址编码为映射 IPv4 的 IPv6 地址，并在 **connect()** 或 **sendto()** 调用的 sockaddr_in6 结构中传递该地址。当应用程序使用 AF_INET6 套接字接受来自 IPv4 节点的 TCP 连接或接收来自 IPv4 节点的 UDP 信息包时，系统在 **accept()**、**recvfrom()** 或 **getpeername()** 调用中使用以此方式编码的 sockaddr_in6 结构将对等项的地址返回至应用程序。

虽然 **bind()** 函数允许应用程序选择 UDP 信息包和 TCP 连接的源 IP 地址，但应用程序通常希望系统为它们选择源地址。应用程序使用 in6addr_any 的方式类似于为此目的在 IPv4 中使用 INADDR_ANY 宏的方式。以此方式绑定的附加功能允许 AF_INET6 套接字与 IPv4 和 IPv6 节点进行通信。例如，在绑定至 in6addr_any 的侦听套接字上发出 **accept()** 的应用程序将接受来自 IPv4 或 IPv6 节点的连接。此行为可通过使用 IPPROTO_IPV6 层套接字选项 IPV6_V6ONLY 进行修改。几乎所有应用程序都不需要知道它们与哪一类型的节点进行交互。但是，对于确实需要知道这一点的应用程序，提供了在 <netinet/in.h> 中定义的 IN6_IS_ADDR_V4MAPPED() 宏。

如果希望了解 IPv4 与 IPv6 的更为详细的比较，参见“信息中心”中的比较 IPv4 与 IPv6。此信息比较这两种协议之间的特色。

有关 AF_INET6 套接字同时与 IPv4 和 IPv6 节点通信的示例程序和描述，参见创建应用程序以接受 IPv4 和 IPv6 客户机。

IPv6 限制

目前在 V5R2 中，OS/400 对 IPv6 的支持在某些功能上有所限制。下表列示这些限制及其对套接字程序员的影响。

表 12. IPv6 限制和影响

限制	影响
IPv6 不支持分段存储。	AF_INET6（SOCK_DGRAM）不应试图发送大小超过接口的 MTU 减去头大小的数据报。
不支持 IPv6 任意广播。	不能连接至或发送至任意广播地址。
不支持 IPv6 多点广播。	不能发送或接收多点广播数据报。
iSeries 主机表不支持 IPv6 地址。	**getaddrinfo()** 和 **getnameinfo()** API 将无法在主机表中找到 IPv6 地址。这些 API 只能在 DNS 中查找地址。
gethostbyname() 和 **gethostbyaddr()** API 仅支持 IPv4 地址解析。	如果需要 IPv6 地址解析，使用 **getaddrinfo()** 和 **getnameinfo()** API。

使用 AF_UNIX 地址系列

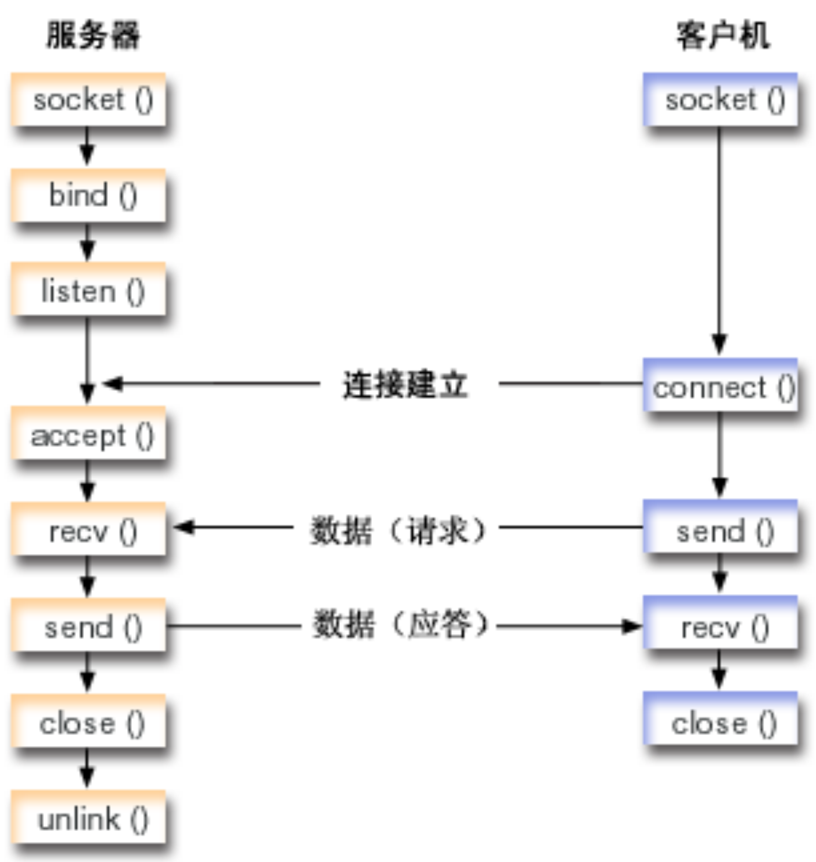
AF_UNIX 地址系列（使用 AF_UNIX 或 AF_UNIX_CCSID 地址系列的套接字）可以是面向连接的（类型 SOCK_STREAM），也可以是无连接的（类型 SOCK_DGRAM）。两种类型都很可靠，原因是没有连接两个进程的外部通信函数。

UNIX 域数据报套接字的运行方式与 UDP 数据报套接字有所不同。借助 UDP 数据报套接字，客户机程序就不必调用 **bind()** 函数，原因是系统会自动指定未使用的端口号。于是服务器可将数据报发送回该端口号。但是，使用 UNIX 域数据报套接字，系统不会自动指定客户机的路径名。因此，使用 UNIX 域数据报的所有客户机程序必须调用 **bind()** 函数。在客户机的 **bind()** 上指定的精确路径名就是传递至服务器的路径名。因此，如果客户机指定相对路径名（即，并非以 / 开头的全限定路径名），除非服务器以同一当前目录运行，否则它不能向客户机发送数据报。

应用程序可能对此地址系列使用的示例路径名就是 /tmp/myserver 或 servers/thatserver。借助 servers/thatserver，可使用并非全限定（未指定 /）的路径名。这表示该项在文件系统层次结构中的位置应根据当前工作目录确定。

注: 文件系统中的路径名是启用了 NLS 的。

下图举例说明了 AF_UNIX 地址系列的客户机／服务器关系。有关将环境设置为使用 AF_UNIX 地址系列的详细信息，参见套接字编程的先决条件。



套接字事件流: 使用 AF_UNIX 地址系列的服务器应用程序

示例: 使用 AF_UNIX 地址系列的服务器应用程序使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有流传输（SOCK_STREAM）的 UNIX 地址系列。该函数返回表示端点的套接字描述符。还可使用 **socketpair()** 函数初始化 UNIX 套接字。

   AF_UNIX 或 AF_UNIX_CCSID 是支持 **socketpair()** 函数的唯一地址系列。**socketpair()** 函数返回未命名的和已连接的套接字描述符。
2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。

UNIX 域套接字的名称空间由路径名组成。当套接字程序调用 **bind()** 函数时，会在文件系统目录中创建一项。如果路径名已存在，则 **bind()** 失败。因此，UNIX 域套接字程序应总是调用 **unlink()** 函数以在结束时除去该目录项。

3. **listen()** 允许服务器接受入局客户机连接。在此示例中，储备设置为 10。这表示系统将对 10 个入局连接请求排队，然后才开始拒绝入局请求。
4. **recv()** 函数从客户机应用程序接收数据。在此示例中，我们知道客户机将发送超过 250 字节的数据。既然如此，就可以使用 SO_RCVLOWAT 套接字选项指定在所有 250 字节数据都到达之前不要唤醒 **recv()**。
5. **send()** 函数将数据回传至客户机。
6. **close()** 函数关闭所有打开的套接字描述符。
7. **unlink()** 函数从文件系统除去 UNIX 路径名。

套接字事件流：使用 AF_UNIX 地址系列的客户机应用程序

示例：使用 AF_UNIX 地址系列的客户机应用程序使用以下函数调用序列：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有流传输（SOCK_STREAM）的 UNIX 地址系列。该函数返回表示端点的套接字描述符。还可使用 **socketpair()** 函数初始化 UNIX 套接字。

   AF_UNIX 或 AF_UNIX_CCSID 是支持 **socketpair()** 函数的唯一地址系列。**socketpair()** 函数返回未命名的和已连接的套接字描述符。
2. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
3. **send()** 函数发送指定的 250 字节数据，该数据是在服务器应用程序中使用 SO_RCVLOWAT 套接字选项指定的。
4. **recv()** 函数一直循环，直到所有 250 字节数据都到达为止。
5. **close()** 函数关闭所有打开的套接字描述符。

示例：使用 AF_UNIX 地址系列的服务器应用程序

此示例为 AF_UNIX 地址系列提供样本服务器。AF_UNIX 地址系列使用的许多套接字调用与其它地址系列一样，但它使用路径名结构来标识服务器应用程序。下列样本程序使用 AF_UNIX 地址系列。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例: 使用 AF_UNIX 地址系列的客户机应用程序

此示例为 AF_UNIX 地址系列提供样本客户机应用程序。AF_UNIX 地址系列使用的许多套接字调用与其它地址系列一样，但它使用路径名结构来标识服务器应用程序。以下样本程序使用 AF_UNIX 地址系列来创建客户机与服务器的连接。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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");
```

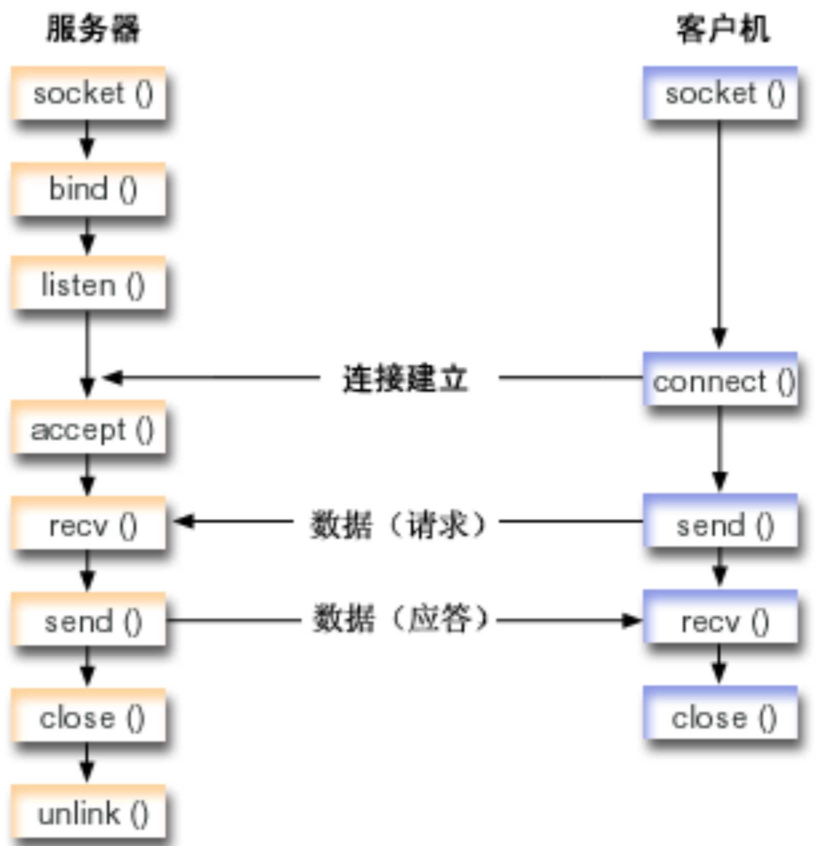
```
         break;
      }

      /********************************************************************/
      /* 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);
}
```

使用 AF_UNIX_CCSID 地址系列

AF_UNIX_CCSID 地址系列套接字使用的规范与 AF_UNIX 地址系列套接字相同。可将它们用于面向连接的或无连接的，并提供同一系统上的通信。有关详细信息，参见使用 AF_UNIX 地址系列。

在使用 AF_UNIX_CCSID 套接字应用程序之前，应了解 **Qlg_Path_Name_T** 结构以确定输出格式。有关详细信息，参见“信息中心”中的“API 参考信息”中的路径名结构。

在使用输出地址结构时（如从 **accept()**、**getsockname()**、**getpeername()**、**recvfrom()** 和 **recvmsg()** 返回的输出地址结构），应用程序必须检查套接字地址结构（sockaddr_unc）以确定其格式。sunc_format 和 sunc_qlg 字段确定路径名的输出格式。但套接字不一定要对输出使用应用程序在输入地址上使用的值。



套接字事件流：使用 AF_UNIX_CCSID 地址系列的服务器应用程序

示例：使用 AF_UNIX_CCSID 地址系列的服务器应用程序使用以下函数调用序列：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有流传输（SOCK_STREAM）的 UNIX_CCSID 地址系列。还可使用 **socketpair()** 函数初始化 UNIX 套接字。

   AF_UNIX 或 AF_UNIX_CCSID 是支持 **socketpair()** 函数的唯一地址系列。**socketpair()** 函数返回未命名的和已连接的两个套接字描述符。
2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。

   | UNIX 域套接字的名称空间由路径名组成。当套接字程序调用 **bind()** 函数时，会在文件系统目录中创建一
   | 项。如果路径名已存在，则 **bind()** 失败。因此，UNIX 域套接字程序应总是调用 **unlink()** 函数以在结束
   | 时除去该目录项。
3. **listen()** 允许服务器接受入局客户机连接。在此示例中，储备设置为 10。这表示系统将对 10 个入局连接请求排队，然后才开始拒绝入局请求。
4. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
5. **recv()** 函数从客户机应用程序接收数据。在此示例中，我们知道客户机将发送超过 250 字节的数据。既然如此，就可以使用 SO_RCVLOWAT 套接字选项指定在所有 250 字节数据都到达之前不要唤醒 **recv()**。
6. **send()** 函数将数据回传至客户机。
7. **close()** 函数关闭所有打开的套接字描述符。

8. **unlink()** 函数从文件系统除去 UNIX 路径名。

套接字事件流：使用 AF_UNIX_CCSID 地址系列的客户机应用程序

示例：使用 AF_UNIX_CCSID 地址系列的客户机应用程序使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有流传输（SOCK_STREAM）的 UNIX 地址系列。该函数返回表示端点的套接字描述符。还可使用 **socketpair()** 函数初始化 UNIX 套接字。

   AF_UNIX 或 AF_UNIX_CCSID 是支持 **socketpair()** 函数的唯一地址系列。**socketpair()** 函数返回未命名的和已连接的套接字描述符。
2. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
3. **send()** 函数发送指定的 250 字节数据，该数据是在服务器应用程序中使用 SO_RCVLOWAT 套接字选项指定的。
4. **recv()** 函数一直循环，直到所有 250 字节数据都到达为止。
5. **close()** 函数关闭所有打开的套接字描述符。

示例：使用 AF_UNIX_CCSID 地址系列的服务器应用程序

下列样本程序使用 AF_UNIX_CCSID 地址系列。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例：使用 AF_UNIX_CCSID 地址系列的客户机应用程序

下列样本程序使用 AF_UNIX_CCSID 地址系列。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

使用 AF_TELEPHONY 地址系列

电话地址系列（使用 AF_TELEPHONY 地址系列的套接字）允许用户通过使用标准套接字 API 的已连接 ISDN 电话网络启动（拨号）和完成（应答）电话呼叫。在此域中组成连接端点的套接字实际上是电话呼叫的被呼叫方（被动端点）和呼叫方（主动端点）。AF_TELEPHONY 地址就是由 sockaddr_tel 地址结构中包含的最多 40 位（0 至 9）组成的电话号码。

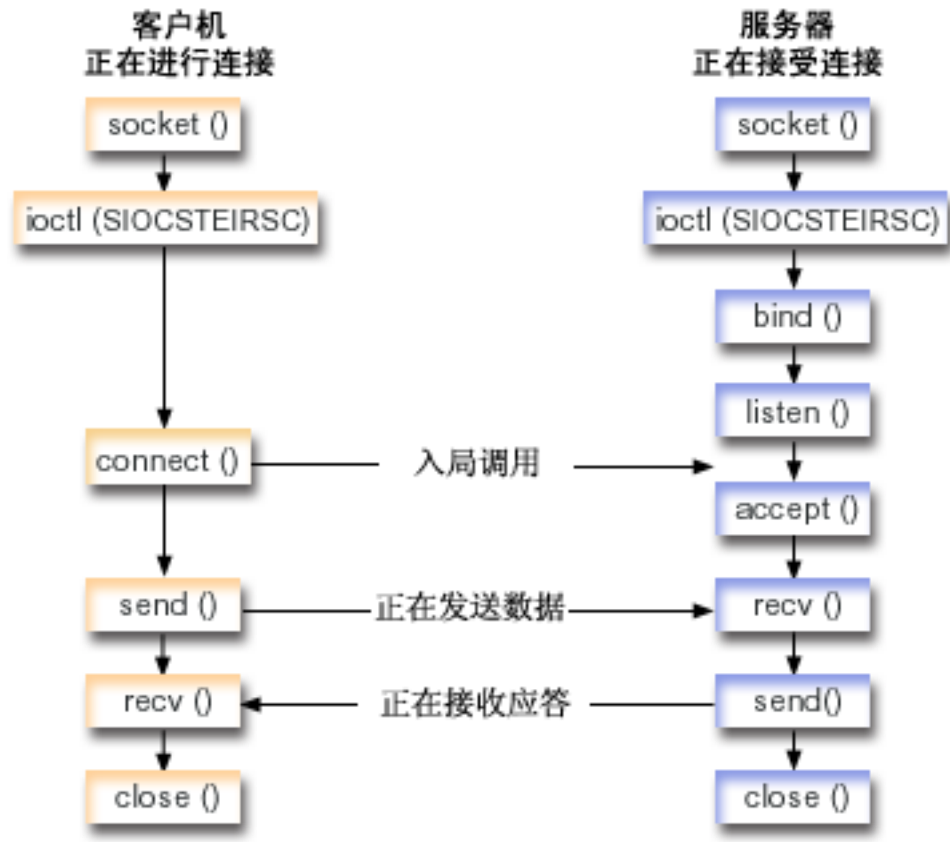
仅支持 AF_TELEPHONY 套接字作为面向连接的（类型 SOCK_STREAM）套接字。这些套接字的语义和函数类似于其它面向连接的协议的语义和函数。主要差别在于电话域中的连接并不比基本电话连接可靠。如果期望有保证的传送，必须在应用程序级别完成，如在使用此系列的 FAX 应用程序中。而且，带外数据的概念在电话地址系列中不受支持。

AF_TELEPHONY 套接字必须先与网络电话设备（逻辑上就是电话）相关联，才能启动或完成连接。使用特殊 **ioctl()** 命令 **SIOCSTELRSC**（设置电话资源）来建立此关联。必须先配置这些设备并使其准备就绪以供使用，然后发出此命令。

在建立 SIOCSTELRSC **ioctl()** 调用之前，应用程序必须解析设备名。设备名必须解析为系统指针，且必须将此指针用作 SIOCSTELRSC 命令的输入。

设备将保留与套接字的关联，直到关闭套接字为止。最后，可使用一个套接字关联多个设备。此多项关联允许应用程序通过单个套接字在多个设备上侦听和应答调用。

下图举例说明了与 AF_TELEPHONY 地址系列配合使用的套接字调用的关系。有关将环境设置为使用 AF_UNIX 地址系列的详细信息，参见套接字编程的先决条件。



| **套接字事件流：使用 AF_TELEPHONY 地址系列的客户机**
| 示例：建立 AF_TELEPHONY 连接使用以下函数调用序列:
| 1. **socket()** 函数返回表示端点的套接字描述符。
| 2. 必须通过将设备名解析为系统指针并针对该请求填充结构以便将套接字与设备相关联。发出 **ioctl()** 函数以
| 关联设备和系统指针。
| 3. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
| 4. **send()** 函数将发送缓冲区的内容发送至客户机。
| 5. **recv()** 函数从客户机应用程序接收数据
| 6. **close()** 函数关闭所有打开的套接字描述符。

| **套接字事件流：使用 AF_TELEPHONY 地址系列的服务器应用程序**
| 示例：接受 AF_TELEPHONY 连接使用以下函数调用序列:
| 1. **socket()** 函数返回表示端点的套接字描述符。
| 2. 必须通过将设备名解析为系统指针并针对该请求填充结构以便将套接字与设备相关联。发出 **ioctl()** 函数以
| 关联设备和系统指针。
| 3. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
| 4. **listen()** 允许服务器接受入局客户机连接。
| 5. 服务器使用 **accept()** 函数接受入局连接请求。
| 6. **recv()** 函数从客户机应用程序接收数据
| 7. **send()** 函数将发送缓冲区的内容发送至客户机。
| 8. **close()** 函数关闭所有打开的套接字描述符。

示例：建立 AF_TELEPHONY 连接

程序可通过电话域套接字相互通信。使用下列代码启用套接字以便与客户机建立连接。有关使用代码示例的信息，参见代码不保证声明。

```
/****************************************************************************/
/* 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);
}
```

示例：接受 AF_TELEPHONY 连接

AF_TELEPHONY 地址系列用于使用电话号码标识套接字的应用程序。此地址系列主要用于传真应用程序。程序可通过电话域套接字相互通信。使用下列代码启用套接字以便接受来自服务器的连接。有关使用代码示例的信息，参见代码不保证声明。

```
/************************************************************************/
/* 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);
}
```

第 7 章 套接字概念

下列主题讨论高级套接字概念，这已经超出什么是套接字以及套接字如何工作的一般讨论。它们提供一些方法来设计套接字应用程序，以适应更大更复杂的网络。下列每个概念都链接至相应的样本程序。

- 异步 I/O
- 安全套接字
- 客户机 SOCKS 支持
- 线程安全
- 非阻塞 I/O
- 信号
- IP 多点广播
- 文件数据传输 — send_file() 和 accept_and_recv()
- 带外数据
- I/O 多路复用 — select()
- 套接字网络函数
- 域名系统（DNS）支持
- BSD 兼容性
- 在进程间传递描述符 — sendmsg() 和 recvmsg()

异步 I/O

异步 I/O API 提供了一种方法，以便线程式客户机服务器模型执行高并行的且存储有效的 I/O。在先前的线程式客户机／服务器模型中，通常流行使用两种 I/O 模型。第一种模型要求每个客户机连接一个线程。第一种模型消耗了过多的线程，可能会招致高额的休眠和唤醒成本。第二种模型通过对一大组客户机连接发出 **select()** API，并将准备好的客户机连接或请求委托给线程以将线程数目减至最少。在第二种模型中，必须选择或对每个后续选择作标记，这可能导致大量冗余工作。

异步 I/O 和重叠式 I/O 解决了这些窘境，方法是在控制权返回给用户应用程序之后将数据传递至用户缓冲区或从用户缓冲区传递数据。异步 I/O 在数据可读或可使用连接传送数据时通知这些工作程序线程。

异步 I/O 的优点

- 更有效的使用系统资源。
  在用户缓冲区中发出的和收到的数据副本对启动请求的应用程序是异步的。这一重叠式处理有效地利用了多个处理器，并且在许多情况下，它还会改进调页速率，原因是在数据到达时会释放系统缓冲区以供重复使用。
- 最小化进程／线程等待时间。
- 为客户机请求提供即时服务。
- 使休眠和唤醒成本趋于平均水平。
- 有效处理“串传输应用程序”。
- 提供更好的可伸缩性。
- 提供处理大量数据传输的最有效方法。
  QsoStartRecv() API 上的 fillBuffer 标志通知操作系统要获取大量数据才能完成异步 I/O。也可以使用一个异步操作发送大量数据。

© Copyright IBM Corp. 2001, 2002

- 最小化需要的线程数。
- 可选择使用定时器指定允许此操作以异步方式完成的最大时间。如果客户机连接已空闲一段时间，服务器将关闭该连接。异步定时器允许服务器强制使用此时间限制。
- 使用 **gsk_secure_soc_startInit()** API 以异步方式启动安全会话。

表 13. 异步 I/O API

函数	描述
gsk_secure_soc_startInit()	使用为 SSL 环境和安全会话设置的属性启动安全会话的异步协商。 **注：** 此 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。
gsk_secure_soc_startRecv()	对安全会话启动异步接收操作。 **注：** 此 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。
gsk_secure_soc_startSend()	对安全会话启动异步发送操作。 **注：** 此 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。
QsoCreateIOCompletionPort()	为已完成的异步重叠式 I/O 操作创建公共等待点。**QsoCreateIOCompletionPort()** 函数返回表示等待点的端口句柄。此句柄是在 **QsoStartRecv()**、**QsoStartSend()**、**QsoStartAccept()**、**gsk_secure_soc_startRecv()** 或 **gsk_secure_soc_startSend()** 函数上指定的，用来启动异步重叠式 I/O 操作。此句柄还可与 **QsoPostIOCompletion()** 配合使用以对关联的 I/O 完成端口公布事件。
QsoDestroyIOCompletionPort()	破坏 I/O 完成端口。
QsoWaitForIOCompletionPort()	等待完成的重叠式 I/O 操作。I/O 完成端口表示此等待点。
QsoStartAccept()	启动异步接受操作。 **注：** 此 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。
QsoStartRecv()	启动异步接收操作。 **注：** 此 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。
QsoStartSend()	启动异步发送操作。 **注：** 此 API 仅支持带有 AF_INET 或 AF_INET6 地址系列且使用 SOCK_STREAM 套接字类型的套接字。
QsoPostIOCompletion()	允许应用程序通知完成端口发生了某些函数或活动。

异步 I/O 是如何工作的

应用程序将使用 **QsoCreateIOCompletionPort()** API 创建 I/O 完成端口。此 API 将返回一个句柄，可使用该句柄安排和等待异步 I/O 请求的完成。该应用程序将启动输入或输出函数，指定 I/O 完成端口句柄。当 I/O 完成时，会对指定 I/O 完成端口公布状态信息和应用程序定义的句柄。对 I/O 完成端口的公布将准确地唤醒正在等待的可能的多个线程的其中一个线程。应用程序接收：

- 在原始请求上提供的缓冲区

- 送至该缓冲区或发自该缓冲区的已处理数据的长度
- 指示完成的 I/O 操作的类型
- 以及在初始 I/O 请求上传递的应用程序定义的句柄

此应用程序句柄可能只是标识客户机连接的套接字描述符，或是指向包含有关客户机连接的状态的大量信息的存储器的指针。自从操作完成且传递了应用程序句柄，工作程序线程就确定了完成客户机连接的下一个步骤。处理这些完成的异步操作的工作程序线程可以处理多个不同的客户机请求，而不是只处理一个客户机请求。由于复制至用户缓冲区与从用户缓冲区复制对服务器进程是异步进行的，所以客户机请求的等待时间就减少了。这对有多个处理器的系统非常有利。

有关简单服务器模型的示例，参见示例：使用异步 I/O。

安全套接字

目前，OS/400 支持使用两种方法在 iSeries 上创建安全套接字应用程序。SSL_ API 和 Global Secure Toolkit（GSKit）通过开放式通信网络（大部分情况是因特网）提供通信保密性。这些 API 允许客户机／服务器应用程序使用指定方式通信以防止窃听、篡改和消息伪造。两者都支持服务器和客户机认证，并允许应用程序使用“安全套接字层”（SSL）协议。但是，GSKit API 在所有 IBM @server 平台上都是受支持的，而 SSL_ API 是 OS/400 操作系统本地的。为确保平台间的互操作性，建议在开发用于安全套接字连接的应用程序时使用 GSKit API。

有关每个 API 的描述，参见下列主题：

| • Global Secure Toolkit（GSKit）API
| • SSL_ API

| 安全套接字 API 错误代码消息主题提供使用安全套接字 API 可能出现的常见错误代码消息列表。

安全套接字概述

“安全套接字层”（SSL）协议一开始是由 Netscape 开发的，它是一种分层协议，在可靠的传输（如“传输控制协议”（TCP））上使用，以便为应用程序提供安全通信。一些需要安全通信的应用程序包括 HTTP、FTP、SMTP 和 TELNET。

启用 SSL 的应用程序通常需要使用不同于未启用 SSL 的应用程序的端口。例如，启用 SSL 的浏览器使用以“HTTPs”而不是“HTTP”开头的全球资源定位器（URL）访问启用 SSL 的超文本传输协议（HTTP）服务器。在大部分情况下，URL“HTTPs”将试图打开与服务器系统的端口 443 而不是标准 HTTP 服务器使用的端口 80 的连接。

定义了多个版本的 SSL 协议。最新版本“传输层安全性”（TLS）V1.0 提供源自 SSL V3.0 的改良升级。iSeries 本地的 SSL_ API 和 GSKit API 都支持 TLS V1.0，带有 SSL V3.0 兼容性的 TLS V1.0、SSL V3.0、SSL V2.0 和带有 2.0 兼容性的 SSL V3.0。有关 TLS V1.0 的更多详细信息，参见 Internet Engineering Task Force (IETF) RFC 2246 ″Transport Layer Security″。

| Global Secure ToolKit（GSKit）API

| Global Secure ToolKit（GSKit）是一组可编程接口，允许应用程序启用 SSL。就象 SSL_ API 一样，GSKit API
| 允许从套接字应用程序访问 SSL 和 TLS 功能。但是，GSKit API 在 IBM @server 平台上是受支持的，同
| 先前的 SSL_ API 相比，在 GSKit API 中更容易编程。此外，已经添加了新的 GSKit API 来创建安全套接
| 字会话的异步实例。此 API 提供安全连接以便控制多台客户机，要不然入局请求的数目会很高且需要多个作
| 业。不过此 API 是 OS/400 本地的，不能移植至其它 @server 平台。

注: 这些 API 仅支持带有地址系列 AF_INET 或 AF_INET6 且类型为 SOCK_STREAM 的套接字。

下表描述 GSKit API:

表 14. Global secure toolkit API

函数	描述
gsk_attribute_get_buffer()	获取关于安全会话或 SSL 环境的特定字符串信息，如证书存储库文件、证书存储库密码、应用程序标识和密码。
gsk_attribute_get_cert_info()	获取关于安全会话或 SSL 环境的服务器或客户机证书的特定信息。
gsk_attribute_get_enum_value()	获取关于安全会话或 SSL 环境的特定枚举数据的值。
gsk_attribute_get_numeric_value()	获取关于安全会话或 SSL 环境的特定数字信息。
gsk_attribute_set_buffer()	将指定缓冲区属性设置为指定安全会话或 SSL 环境中的值。
gsk_attribute_set_enum()	将指定枚举类型属性设置为安全会话或 SSL 环境中的枚举值。
gsk_attribute_set_numeric_value()	设置关于安全会话或 SSL 环境的特定数字信息。
gsk_environment_close()	关闭 SSL 环境并释放与该环境相关联的所有存储器。
gsk_environment_init()	在设置完所有必需的属性后初始化 SSL 环境。
gsk_environment_open()	返回必须保存以便在后续 gsk 调用上使用的 SSL 环境句柄。
gsk_secure_soc_close()	关闭安全会话并释放该安全会话的所有关联资源。
gsk_secure_soc_init()	使用为 SSL 环境和安全会话设置的属性协商安全会话。
gsk_secure_soc_misc()	对安全会话执行其它函数。
gsk_secure_soc_open()	获取安全会话的存储器，设置属性的缺省值，并返回必须保存以便在与安全会话相关的函数调用上使用的句柄。
gsk_secure_soc_read()	从安全会话接收数据。
gsk_secure_soc_startInit()	使用为 SSL 环境和安全会话设置的属性启动安全会话的异步协商。
gsk_secure_soc_write()	写下有关安全会话的数据。
gsk_secure_soc_startRecv()	对安全会话启动异步接收操作。
gsk_secure_soc_startSend()	对安全会话启动异步发送操作。
gsk_strerror()	检索错误消息和相关联的文本字符串，它描述通过调用 GSK API 返回的返回值。

使用套接字和 GSKit API 的应用程序包含下列元素:

1. 对 **socket()** 的调用，用来获取套接字描述符。
2. 对 **gsk_environment_open()** 的调用，用来获取 SSL 环境的句柄。
3. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置 SSL 环境的属性。至少是对 **gsk_attribute_set_buffer()** 的调用，用来设置 GSK_OS400_APPLICATION_ID 值或设置 GSK_KEYRING_FILE 值。仅应设置其中之一。最好使用 GSK_OS400_APPLICATION_ID 值。还应确保使用 **gsk_attribute_set_enum()** 设置应用程序（客户机或服务器）的类型 GSK_SESSION_TYPE。
4. 对 **gsk_environment_init()** 的调用，用来初始化此环境以便进行 SSL 处理和建立将使用此环境运行的所有 SSL 会话的 SSL 安全性信息。
5. 套接字调用，用来激活连接。它调用 **connect()** 以激活客户机程序的连接，或调用 **bind()**、**listen()** 和 **accept()** 允许服务器以接受入局连接请求。

6. 对 **gsk_secure_soc_open()** 的调用，用来获取安全会话的句柄。
7. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置安全会话的属性。至少是对 **gsk_attribute_set_numeric_value()** 的调用，用来将特定套接字与此安全会话相关联。
8. 对 **gsk_secure_soc_init()** 的调用，用来启动加密参数的 SSL 握手协商。

   注: 通常，服务器程序必须提供证书，SSL 握手才会成功。服务器还必须具有对与服务器证书相关联的专用密钥以及存储该证书的密钥数据库文件的访问权。在某些情况下，客户机还必须在 SSL 握手处理期间提供证书。如果客户机正连接至的服务器已启用客户机认证，就需要这样做。**gsk_attribute_set_buffer**（GSK_OS400_APPLICATION_ID）或 **gsk_attribute_set_buffer**（GSK_KEYRING_FILE）API 调用标识（尽管以不同方式）在获取握手期间从中使用证书和专用密钥的密钥数据库文件。
9. 对 **gsk_secure_soc_read()** 和 **gsk_secure_soc_write()** 的调用，用来接收和发送数据。
10. 对 **gsk_secure_soc_close()** 的调用，用来结束安全会话。
11. 对 **gsk_environment_close()** 的调用，用来关闭 SSL 环境。
12. 对 **close()** 的调用，用来破坏连接的套接字。

有关使用 GSKit API 的样本程序，参见下列示例:
- 示例：使用异步数据接收建立安全服务器
- 示例：使用异步握手建立安全服务器
- 示例：使用 Global Secure ToolKit（GSKit）API 建立安全客户机

SSL_ API

SSL_ API 允许程序员在 iSeries 上创建安全套接字应用程序。与 GSKit API 不同，SSL_ API 仅对于 OS/400 系统才是本地的。下表描述在 OS/400 实现中受支持的 9 个 SSL_ API。使用这些链接来了解有关“信息中心”中的 API 信息中列示的个别 API 的详细信息。

表 15. SSL_ API

函数	描述
SSL_Create()	对指定套接字描述符启用 SSL 支持。
SSL_Destroy()	对指定 SSL 会话和套接字结束 SSL 支持。
SSL_Handshake()	启动 SSL 握手协议。
SSL_Init()	初始化 SSL 的当前作业并建立当前作业的 SSL 安全性信息。 **注:** 必须先在进程中执行 **SSL_Init()** 或 **SSL_Init_Application()** API 才能使用 SSL。
SSL_Init_Application()	初始化 SSL 的当前作业并建立当前作业的 SSL 安全性信息。 **注:** 必须先在进程中执行 **SSL_Init()** 或 **SSL_Init_Application()** API 才能使用 SSL。
SSL_Read()	从启用 SSL 的套接字描述符接收数据。
SSL_Write()	将数据写至启用 SSL 的套接字描述符。
SSL_Strerror()	检索 SSL 运行时错误消息。
SSL_Perror()	打印 SSL 错误消息。

使用套接字和 SSL_ API 的应用程序包含下列元素:

- 对 **socket()** 的调用，用来获取套接字描述符。
- 调用 **SSL_Init()** 或 **SSL_Init_Application()**，以初始化 SSL 处理的作业环境和建立将在当前作业中运行的所有 SSL 会话的 SSL 安全性信息。仅应使用其中一个 API。最好使用 **SSL_Init_Application()** API。
- 套接字调用，用来激活连接。它调用 **connect()** 以激活客户机程序的连接，或调用 **bind()**、**listen()** 和 **accept()** 以允许服务器接受入局连接请求。
- 对 **SSL_Create()** 的调用，用来对连接的套接字启用 SSL 支持。
- 对 **SSL_Handshake()** 的调用，用来启动加密参数的 SSL 握手协商。

  注: 通常，服务器程序必须提供证书，SSL 握手才会成功。服务器还必须具有对与服务器证书相关联的专用密钥以及存储该证书的密钥数据库文件的访问权。在某些情况下，客户机还必须在 SSL 握手处理期间提供证书。如果客户机正连接至的服务器已启用客户机认证，就需要这样做。**SSL_Init()** 或 **SSL_Init_Application()** API 标识（尽管以不同方式）在获取握手期间从中使用证书和专用密钥的密钥数据库文件。
- 对 **SSL_Read()** 和 **SSL_Write()** 的调用，用来接收和发送数据。
- 对 **SSL_Destroy()** 的调用，用来对套接字禁用 SSL 支持。
- 对 **close()** 的调用，用来破坏连接的套接字。

有关使用这些 SSL_ API 的样本程序，参见下列样本程序:
- 示例: 使用 SSL_ API 建立安全服务器
- 示例: 使用 SSL_ API 建立安全客户机

安全套接字 API 错误代码消息

要访问有关下列安全套接字错误代码消息的信息，完成下列步骤:
1. 在命令行上输入
   ```
   DSPMSGD RANGE(XXXXXXX)
   ```
   其中 XXXXXXX 是返回码的消息标识。例如，如果返回码为 3，则应输入
   ```
   DSPMSGD RANGE(CPDBCB9)
   ```
2. 选择 **1** 以显示消息文本。

表 16. 安全套接字 API 错误代码消息

返回码	消息标识	常量名
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

表 16. 安全套接字 API 错误代码消息 (续)

403	CPDBC82	GSK_ERROR_NO_CERTIFICATE
404	CPDBC84	GSK_ERROR_BAD_CERTIFICATE
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

表 16. 安全套接字 API 错误代码消息 (续)

6018	CPDBC87	GSK_OS400_ERROR_CRL_INVALID
6019	CPCBC88	GSK_OS400_ASYNCHRONOUS_SOC_INIT
0	CPCBC80	成功返回
-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（映射为 -4）
-16	CPDBC8B	SSL_ERROR_BAD_PEER（映射为 -11）
-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

客户机 SOCKS 支持

iSeries 使用 SOCKS V4 以允许使用 AF_INET 地址系列且套接字类型为 SOCK_STREAM 的程序与在防火墙外部的系统上运行的服务器程序通信。防火墙是网络管理员放置在安全的内部网络与低安全的外部网络之间的非常安全的主机。通常这种网络配置不允许源自安全主机的通信在低安全网络上进行路由选择，反之亦然。防火墙上的代理服务器帮助管理在安全主机与低安全网络之间进行的必需的访问。

在安全内部网络中的主机上运行的应用程序必须将它们的请求发送至防火墙代理服务器才能对防火墙进行导航。于是代理服务器将这些请求转发至低安全网络上的实际服务器，然后将应答中继回起始主机上的应用程序。代理服务器的常见示例是 HTTP 代理服务器。代理服务器对 HTTP 客户机执行一些任务:

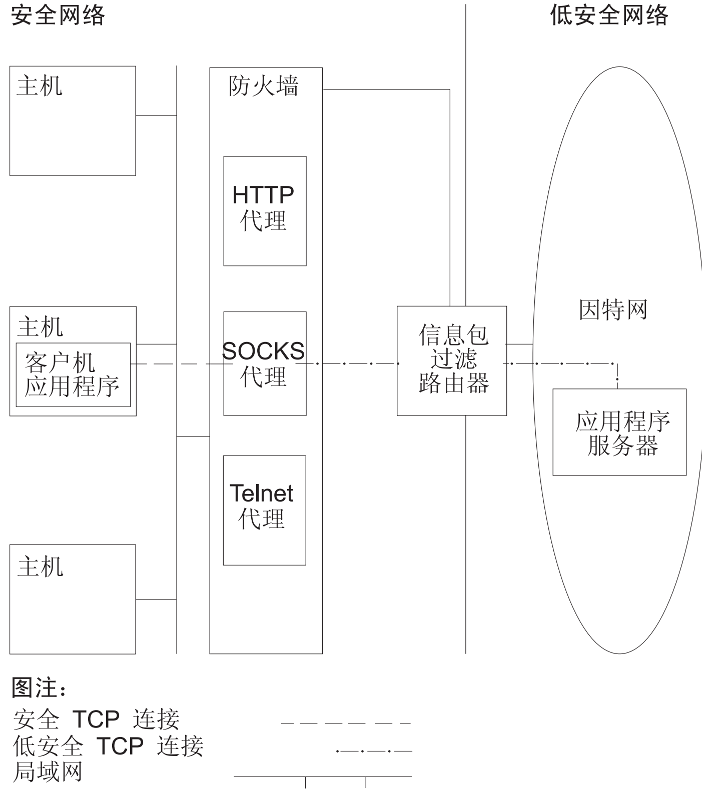
- 它们对外部系统隐藏内部网络。

- 它们保护主机免受外部系统的直接访问。
- 如果经过正确设计和配置，它们可以过滤来自外部的数据。

HTTP 代理服务器仅控制 HTTP 客户机。

在防火墙上运行多个代理服务器的常见备用方法是运行称为 SOCKS 服务器的健壮代理服务器。SOCKS 服务器可充当使用套接字 API 建立的任何 TCP 客户机连接的代理。iSeries 客户机 SOCKS 支持的关键优点在于它允许客户机应用程序透明地访问 SOCKS 服务器而不更改任何客户机代码。

下图显示常见防火墙布置，防火墙上有 HTTP 代理、Telnet 代理和 SOCKS 代理。注意用于正访问因特网上的服务器的安全客户机的两个独立 TCP 连接。一个连接从安全主机指向 SOCKS 服务器，另一个连接从低安全网络指向 SOCKS 服务器。



要使用 SOCKS 服务器，需要对安全客户机主机执行两个操作:

1. 配置 SOCKS 服务器。2000 年 2 月 15 日，IBM 声明 IBM Firewall for iSeries 产品（5769-FW1）的当前 V4R4 的功能是最强的，它提供 SOCKS 服务器支持。
2. 在安全客户机系统上，定义在客户机系统上要引导至 SOCKS 服务器的所有出站客户机 TCP 连接。可通过使用 iSeries Access 95 或 Windows NT® 的 “iSeries 导航器” 功能下的 SOCKS 选项卡来定义安全客户机 SOCKS 配置项。SOCKS 选项卡有很多关于为安全客户机系统配置客户机 SOCKS 支持的帮助。

   要配置客户机 SOCKS 支持，执行下列操作:

   a. 在 “iSeries 导航器” 上，展开 **iSeries 服务器 --> 网络 --> TCP/IP 配置**。
   b. 右键单击 **TCP/IP 配置**。
   c. 单击**属性**。
   d. 单击 **SOCKS** 选项卡。
   e. 在 SOCKS 页面上输入连接信息。

注: 安全客户机 SOCKS 配置数据保存在安全客户机主机系统上的库 QUSRSYS 中的文件 QASOSCFG 中。

一旦作了配置，系统就会自动将特定出站连接引导至在 SOCKS 页面上指定的 SOCKS 服务器。不必对安全客户机应用程序作任何更改。在接收请求时，SOCKS 服务器会建立与低安全网络中的服务器的另一外部 TCP/IP 连接。然后 SOCKS 服务器会在内部和外部 TCP/IP 连接之间中继数据。

注: 低安全网络上的远程主机直接连接至 SOCKS 服务器。它没有对安全客户机的直接访问权。
到了这时，我们已经解决了源自安全客户机的“出站”TCP 连接。客户机 SOCKS 支持还允许您通知 SOCKS 服务器允许跨防火墙的入站连接请求。来自安全客户机系统的 **Rbind()** 调用允许这一通信。要让 **Rbind()** 起作用，安全客户机先前必须已经发出了 **connect()** 调用，且该调用必须已经通过 SOCKS 服务器生成出站连接。**Rbind()** 入站连接必须源自 **connect()** 建立的出站连接寻址得到的 IP 地址。

下图详细概述套接字函数如何与对应用程序透明的 SOCKS 服务进行交互。在该示例中，FTP 客户机调用 **Rbind()** 函数而不是 **bind()** 函数[1]。它通过使用 __Rbind 预处理器 #define（将 **bind()** 定义为 **Rbind()**）重新编译 FTP 客户机代码来建立此调用。或者，应用程序可在相关源代码中显式编写 **Rbind()**。如果应用程序不需要通过 SOCKS 服务器的入站连接，则不应使用 **Rbind()**。

注:

1. FTP 客户机使用 **Rbind()**，原因是 FTP 协议允许 FTP 服务器因为来自 FTP 客户机的请求而建立数据连接以发送文件或数据。
2. SOCKS 服务器建立与 FTP 客户机的数据连接并在 FTP 客户机与 FTP 服务器之间中继数据。许多 SOCKS 服务器允许服务器在一定时间内连接至安全客户机。如果服务器在此时间内未进行连接，将在 **accept()** 上遇到错误号 ECONNABORTED。
3. FTP 客户机通过 SOCKS 服务器启动与低安全网络的出站 TCP 连接。FTP 客户机对连接指定的目标地址是位于低安全网络上的 FTP 服务器的 IP 地址和端口。安全主机系统已通过 SOCKS 页面配置为通过 SOCK 服务器引导此连接。一旦作出配置，系统就会自动将连接引导至通过 SOCKS 页面指定的 SOCKS 服务器。
4. 打开套接字并调用 **Rbind()** 以建立入站 TCP 连接。建立之后，此入站连接就是来自上面指定的相同目标出站 IP 地址的连接。对于特定线程，基于 SOCKS 服务器的出站和入站连接必须是成对的。换句话说，所有 **Rbind()** 入站连接应紧跟在基于 SOCKS 服务器的出站连接之后，且在 **Rbind()** 运行之前，不会尝试插入任何与此线程相关的非 SOCKS 连接。
5. **getsockname()** 返回 SOCKS 服务器地址。套接字在逻辑上绑定至与通过 SOCKS 服务器选择的端口配对的 SOCKS 服务器 IP 地址。在此示例中，地址是通过“控制连接”套接字 CTLed 发送至位于低安全网络上的 FTP 服务器的。这就是 FTP 服务器连接至的地址。FTP 服务器连接至 SOCKS 服务器而不直接连接至安全主机。
6. SOCKS 服务器建立与 FTP 客户机的数据连接并在 FTP 客户机与 FTP 服务器之间中继数据。许多 SOCKS 服务器允许服务器在一定时间内连接至安全客户机。如果服务器在此时间内未进行连接，将在 **accept()** 上遇到错误号 ECONNABORTED。

线程安全

如果可以在同一进程内的多个线程内同时启动一个函数，则认为该函数是线程安全的。当且仅当某个函数调用的所有函数都是线程安全的，该函数才是线程安全的。套接字 API 由系统和网络函数组成，它们都是线程安全的。

名称以“_r”结尾的所有网络函数的语义相似，而且都是线程安全的。有关使用线程安全套接字 API 的样本程序，参见示例：对线程安全网络例程使用 gethostbyaddr_r()。

其它解析程序例程相互是线程安全的，但它们使用 _res 数据结构。此数据结构在进程内的所有线程间是共享的，在解析程序调用期间可通过应用程序进行更改。有关使用解析程序例程的样本程序，参见示例：更新和查询 DNS。

非阻塞 I/O

当应用程序发出套接字输入函数之一且没有数据可读时，函数会被阻塞，在有数据可读之前不会返回。相似地，如果不能即时发送数据，应用程序会在套接字输出函数上被阻塞。最后，在等待与伙伴的程序建立连接时，**connect()** 和 **accept()** 也会被阻塞。

套接字提供了一个方法，以允许应用程序发出被阻塞的函数以便该函数返回而不延迟。这是通过调用 **fcntl()** 以打开 **O_NONBLOCK** 标志或调用 **ioctl()** 以打开 **FIONBIO** 标志完成的。一旦在此非阻塞方式下运行，如果函数不能在不阻塞的情况下完成，它会立即返回。**connect()** 可能返回并带有 **[EINPROGRESS]**，表示该连接启动已启动。然后可使用 **select()** 来确定何时完成连接。对于由于以非阻塞方式运行而受到影响的所有其它函数，错误代码 **[EWOULDBLOCK]** 指示调用不成功。

可将非阻塞与下列套接字函数配合使用：

- **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()**

有关使用非阻塞 I/O 的样本程序，参见示例：非阻塞 I/O 和 select()。

信号

当应用程序对 occurs 发生兴趣时，可请求以异步方式得到通知（请求系统发送**信号**）。套接字会向应用程序发送两种异步信号。

1. **SIGURG** 是一种信号，在支持带外（OOB）数据的概念的套接字上接收到 OOB 数据时将发送该信号。例如，带有地址系列 AF_INET 且类型为 SOCK_STREAM 的套接字可调节为发出 SIGURG 信号。

2. **SIGIO** 是一种信号，在任何类型的套接字上出现普通数据、OOB 数据、错误状态或差不多任何内容时将发送该信号。

应用程序应确保能够控制在请求系统发送信号之前先接收信号。这是通过设置**信号处理程序**完成的。设置信号处理程序的一种方法是发出 **sigaction()** 调用。

应用程序使用下列方法之一请求系统发送 **SIGURG** 信号:

- 发出 **fcntl()** 调用并使用 F_SETOWN 命令指定进程标识或进程组标识。
- 发出 **ioctl()** 调用并指定 FIOSETOWN 或 SIOCSPGRP 命令（请求）。

应用程序在请求系统发送 **SIGIO** 信号时分为两个步骤。首先它必须按上述要求为 **SIGURG** 信号设置进程标识或进程组标识。这会让系统知道应用程序想要在什么位置传送信号。接着，应用程序必须执行下列任一操作:

- 发出 **fcntl()** 调用并指定带有 FASYNC 标志的 F_SETFL 命令。
- 发出 **ioctl()** 调用并指定 FIOASYNC 命令。

此步骤请求系统生成 SIGIO 信号。注意，可以任何次序完成这些步骤。还应注意，如果应用程序对侦听套接字发出这些请求，这些请求设置的值将由从 **accept()** 函数返回至应用程序的所有套接字继承。也就是说，就发送 SIGIO 信号而言，新接受的套接字也具有相同的进程标识或进程组标识以及相同的信息。

套接字还会生成有关错误情况的同步信号。每当应用程序接收到 [EPIPE] 时，就会将套接字函数上的**错误号** SIGPIPE 信号传送至发出接收**错误号**值的操作的进程。在 BSD 实现中，缺省情况下 SIGPIPE 信号会结束接收到**错误号**值的进程。为保留与 OS/400 实现的先前发行版的兼容性，OS/400 实现对 SIGPIPE 信号使用缺省行为即忽略。这确保了现有应用程序不会受添加信号函数的负面影响。

在将信号传送至在套接字函数上被阻塞的进程时，从等待状态返回的函数带有 **[EINTR] 错误号**值，这允许运行应用程序的信号处理程序。会发生这种情况的函数包括:

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

一定要注意，这些信号不会向应用程序提供套接字描述符，该描述符标识被通知的情况实际存在的位置。所以，如果应用程序在使用多个套接字描述符，它必须轮询描述符或使用 **select()** 调用来确定接收到信号的原因。

有关使用信号的样本程序，参见示例：将信号与阻塞套接字 API 配合使用。

IP 多点广播

IP 多点广播允许应用程序发送网络中的一组主机可以接收到的单个 IP 数据报。该组中的主机可能驻留在单个子网中，也可能在可使用多点广播的路由器连接的不同子网中。主机可以随时加入或离开组。对主机组中的成员位置或数目没有任何限制。范围在 224.0.0.1 到 239.255.255.255 之间的 D 类因特网地址标识主机组。

目前仅可将 IP 多点广播与 AF_INET 地址系列配合使用。

应用程序可使用套接字 API 和无连接的 SOCK_DGRAM 类型套接字发送或接收多点广播数据报。多点广播是一种一对多的传送方法。类型为 SOCK_STREAM 的面向连接的套接字不能用于多点广播。在创建类型为 SOCK_DGRAM 的套接字后，应用程序可使用 **setsockopt()** 函数来控制与该套接字相关联的多点广播特征。**setsockopt()** 函数接受下列 IPPROTO_IP 级别标志:

- IP_ADD_MEMBERSHIP: 加入指定的多点广播组
- IP_DROP_MEMBERSHIP: 离开指定的多点广播组
- IP_MULTICAST_IF: 设置通过其发送出局多点广播数据报的接口
- IP_MULTICAST_TTL: 在 IP 头中设置出局多点广播数据报的“有效时间”（TTL）
- IP_MULTICAST_LOOP: 指定当发送主机是多点广播组的成员时，出局多点广播数据报的副本是否应传送至该主机

有关 IP 多点广播的示例，参见下列示例示例: 使用多点广播。

文件数据传输 — send_file() 和 accept_and_recv()

OS/400 套接字提供 **send_file()** 和 **accept_and_recv()** API，使得基于连接的套接字的文件传送更快更容易。对于文件服务应用程序（如超文本传输协议（HTTP）服务器），这两个 API 特别有用。

send_file() 允许使用单个 API 调用通过连接的套接字从文件系统直接发送文件数据。

accept_and_recv() 是以下三个套接字函数的组合: **accept()**、**getsockname()** 和 **recv()**。

有关 **send_file()** 和 **accept_and_recv()** API 的样本程序，参见示例: 使用 send_file() 和 accept_and_recv() 传输文件数据。

带外数据

带外（00B）数据是特定于用户的数据，仅对面向连接的（流）套接字有意义。流数据通常是按发送次序接收的。OOB 数据的接收与它在流中的位置无关（与发送它的次序无关）。这是有可能的，原因是数据是按以下方式标记的，在将数据从程序 A 发送至程序 B 时，会通知程序 B 数据到达。

| OOB 数据仅在 AF_INET（SOCK_STREAM）和 AF_INET6（SOCK_STREAM）上受支持。

通过在 **send()**、**sendto()** 和 **sendmsg()** 函数上指定 MSG_OOB 标志来发送 OOB 数据。

传送 OOB 数据与传送常规数据一样。它是在所有缓冲数据之后发送的。换句话说，OOB 数据的优先级别没有可能缓冲的任何数据的优先级别高；数据是按其发送次序传送的。

在接收端，事情有一点复杂:

- 套接字 API 通过使用 OOB 标记程序了解在系统上接收到的 OOB 数据。OOB 标记程序指向发送的 OOB 数据中的最后一个字节。

注: 指示 OOB 标记程序指向哪个字节的值是在系统基础上设置的（所有应用程序都使用该值）。此值在 TCP 连接的本地和远程端必须一致。使用此值的套接字应用程序在使用它时必须在客户机和服务器应用程序之间保持一致。要更改 OOB 标记程序指向的字节，参见“信息中心”中的更改 TCP 属性（CHGTCPA）命令。

SIOCATMARK ioctl() 请求确定读指针是否正指向最后一个 OOB 字节。

注: 如果发送多次出现的 OOB 数据，则 OOB 标记程序指向最后一次 OOB 数据出现的最后一个 OOB 字节。

- 无论 OOB 数据是否以直接插入方式接收，输入操作会一直处理数据直至遇到 OOB 标记程序（如果发送了 OOB 数据的话）。
- **recv()**、**recvmsg()** 或 **recvfrom()** 函数（设置有 MSG_OOB 标志）用于接收 OOB 数据。如果其中一个接收函数完成并发生下列情况之一，会返回错误 [EINVAL]。
  - 未设置套接字选项 SO_OOBINLINE，也没有 OOB 数据可接收。
  - 设置了套接字选项 SO_OOBINLINE。

  如果未设置套接字选项 SO_OOBINLINE，且发送程序发送的 OOB 数据的大小超过 1 字节，则除最后一个字节之外的所有字节都被视作普通数据。（普通数据表示接收程序可接收数据而不指定 MSG_OOB 标志。）发送的 OOB 数据的最后一个字节未存储在普通数据流中。只能发出 **recv()**、**recvmsg()** 或 **recvfrom()** 函数（设置有 MSG_OOB 标志）来检索此字节。如果未设置 MSG_OOB 标志而发出接收，将检索普通数据，OOB 字节将被删除。而且，如果发送多次出现的 OOB 数据，则先前出现的 OOB 数据将会丢失，仅记住最后一次 OOB 数据出现的 OOB 数据位置。

  如果设置了套接字选项 SO_OOBINLINE，则发送的所有 OOB 数据都存储在普通数据流中。可通过发出下列三个接收函数之一而不指定 MSG_OOB 标志（如果指定它的话，将返回错误 [EINVAL]）来检索数据。如果发送多次出现的 OOB 数据，OOB 数据不会丢失。
- 如果未设置 SO_OOBINLINE 且已接收到 OOB 数据，则不会废弃 OOB 数据，用户会将 SO_OOBINLINE 设置为开。初始 OOB 字节被视作普通数据。
- 如果未设置 SO_OOBINLINE 且已发送 OOB 数据，同时接收程序发出了输入函数以接收 OOB 数据，则 OOB 标记程序仍然有效。接收程序仍然可以检查读指针是否在 OOB 标记程序上，即使接收到 OOB 字节。

I/O 多路复用 — select()

因为异步 I/O 提供了更有效的方法以最大限度地利用应用程序资源，所以建议使用异步 I/O API 而不是 **select()** API。但是，特定应用程序设计可能允许使用 **select()**。与异步 I/O 一样，**select()** 创建公共点以同时等待多种情况。而且 **select()** 允许应用程序指定描述符集合以执行下列操作:

- 查看是否有要读取的数据。
- 查看是否可写入数据。
- 查看是否有异常情况。

可在每一个集合中指定的描述符可以是套接字描述符、文件描述符或由描述符表示的任何其它对象。

select() 函数还允许应用程序指定是否想要等待数据变得可用。应用程序可指定要等待多长时间。有关样本程序，参见示例: 非阻塞 I/O 和 select()。

套接字网络函数

套接字网络函数允许应用程序从主机、协议、服务和网络文件获取信息。该信息可按名称或地址或按文件的顺序存取法访问。在跨网络运行的程序之间设置通信时需要这些网络函数（或例程），因此 AF_UNIX 套接字不会使用它们。有关个别网络函数例程的简要总结，参见“信息中心”中的 API 参考主题中的套接字网络函数（例程）。

这些例程执行下列操作:

- 将主机名映射为网络地址。
- 将网络名映射为网络号。
- 将协议名映射为协议号。
- 将服务名映射为端口号。
- 转换因特网网络地址的字节顺序。
- 转换因特网地址和点分十进制表示。

网络例程中包括一组称为解析程序例程的例程。这些例程为因特网域中的名称服务器建立、发送和解释信息包，同时还用于名称解析。通常 **gethostbyname()**、**gethostbyaddr()**、**getnameinfo()** 和 **getaddrinfo()** 会调用解析程序例程，而且可以直接调用。有关使用这些解析程序例程的示例，参见示例：对线程安全网络例程使用 gethostbyaddr_r()。解析程序例程主要用于通过套接字应用程序访问域名系统（DNS）。有关如何将套接字与 DNS 配合使用的详细信息，参见域名服务支持。

域名系统（DNS）支持

iSeries 允许应用程序通过解析程序函数访问域名系统（DNS）。DNS 有下列三个主要组件:

- **域名空间和资源记录**
  树结构的名称空间以及与这些名称相关联的数据的说明。
- **名称服务器**
  保存有关域树结构的信息和设置信息的服务器程序。有关名称服务器的更多信息，参见“信息中心”中的 DNS 主题。
- **解析程序**
  从名称服务器中抽取信息以响应客户机请求的程序。

在 OS/400 实现中提供的解析程序就是提供与名称服务器的通信的套接字函数。这些例程用来建立、发送、更新和解释信息包以及执行名称高速缓存以获取高性能。它们还为提供用于 ASCII 至 EBCDIC 和 EBCDIC 至 ASCII 转换的函数。解析程序可选择使用事务签名（TSIG）与 DNS 安全地通信。有关个别解析程序例程的简要总结，参见“信息中心”中的 API 参考主题中的套接字网络函数（例程）。此链接还提供了有关 _res 结构的信息。_res 结构包含这些例程使用的全局信息。

有关域名的更多信息，参见以下 RFC，可在 RFC 搜索页面中找到它。

- 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″

| 有关将 DNS 与套接字应用程序配合使用的其它方式的更多信息，参见下列主题:

- 环境变量
  本主题描述可用于名称解析的环境变量。
- 数据高速缓存
  本主题提供有关使用套接字高速缓存对 DNS 查询的响应以减少网络流量的详细信息。示例：更新和查询 DNS 提供了一个样本程序，以显示如何使用套接字 API 查询和更新 DNS 记录。

环境变量

可使用环境变量来覆盖解析程序函数的缺省初始设置。仅在成功调用 **res_init()** 或 **res_ninit()** 后才会检查环境变量。所以，如果已手工初始化该结构，将忽略环境变量。还应注意，该结构只初始化一次，所以以后对环境变量作出的更改将被忽略。

注意: 环境变量的名称必须大写。字符串值可以是混合大小写的。使用 CCSID 290 的日语系统在环境变量名称和值中只能使用大写字符和数字。该列表包含可与 **res_init()** 和 **res_ninit()** API 配合使用的环境变量的描述。

LOCALDOMAIN
将此环境变量设置为最多 6 个搜索域，总计 256 个字符（包括空格）的由空格隔开的列表。这将覆盖已配置的搜索列表（struct state.defdname 和 struct state.dnsrch）。如果指定搜索列表，则不会对查询使用缺省本地域。

RES_OPTIONS
允许修改某个内部解析程序变量。可对环境变量设置下列由空格隔开的选项中的一个或多个:

- **NDOTS:n**
  对在建立初始绝对查询之前指定给 **res_query()** 的名称中必须出现的点的数目设置阈值。n 的缺省值为“1”，表示如果名称中有任何点，在向其追加任何搜索列表元素之前，应首先尝试将该名称作为绝对名称。
- **TIMEOUT:n**
  设置在放弃并重试查询之前解析程序等待来自远程名称服务器的响应的时间量（以秒计）。
- **ATTEMPTS:n**
  设置在放弃并尝试下一个列示名称服务器之前解析程序将发送至给定名称服务器的查询数。
- **ROTATE**
  在 _res.options 中设置 RES_ROTATE，它将旋转列示名称服务器中的名称服务器选择。这相当于展开所有列示服务器中的查询装入，而不是让所有客户机每次都尝试使用第一个列示服务器。
- **NO-CHECK-NAMES**
  在 _res.options 中设置 RES_NOCHECKNAME，这将禁用针对无效字符（如下划线（_）、非 ASCII 或控制字符）作出的入局主机名和邮件名的现代 BIND 检查。

QIBM_BIND_RESOLVER_FLAGS
将此环境变量设置为解析程序选项标志的由空格隔开的列表。这将覆盖 RES_DEFAULT 选项（struct state.options）和系统配置值（更改 TCP/IP 域 — CHGTCPDMN）。state.options 结构将使用 RES_DEFAULT、OPTIONS 环境值和 CHGTCPDMN 配置值以普通方式初始化。然后使用此环境变量来覆盖那些缺省值。在此环境变量中命名的标志必须使用“+”、“-”或“NOT_”作为前缀来设置（“+”）或重新设置（“-”和“NOT_”）值。

例如，要打开 RES_NOCHECKNAME 和关闭 RES_ROTATE，从基于字符的界面使用以下命令:

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

| 或

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

QIBM_BIND_RESOLVER_SORTLIST
将此环境变量设置为使用点分十进制格式（9.5.9.0/255.255.255.0），最多 10 个 IP 地址／掩码对的由空格隔开的列表，以创建排序列表（struct state.sort_list）。

数据高速缓存

高速缓存对 DNS 查询的响应是由 OS/400 套接字完成的，目的是为了减少网络流量。根据需要添加和更新高速缓存。

如果在 _res.options 中设置了 RES_AAONLY（仅适用于权威应答），将总是通过网络发送查询。在此情况下，不会检查应答的高速缓存。如果未设置 RES_AAONLY，则先检查查询的应答的高速缓存，然后才尝试通过网络发送查询。如果发现应答且有效时间未到期，则会将该应答作为查询的应答返回给用户。如果有效时间已到期，则除去该项，并通过网络发送查询。而且，如果在高速缓存中找不到应答，则通过网络发送该查询。

如果响应是权威的，则高速缓存来自网络的应答。不会高速缓存非权威应答。而且，不会高速缓存作为逆向查询的结果接收的响应。可通过使用 CHGTCPDMN（CFGTCP 选项 12）更新 DNS 配置或通过“iSeries 导航器”清除此高速缓存。

有关使用数据高速缓存的示例程序，参见示例：更新和查询 DNS。

Berkeley Software Distributions（BSD）兼容性

套接字是一个 Berkeley Software Distributions（BSD）接口。语义，如应用程序接收的返回码和受支持函数上可用的自变量，都是 BSD 语义。但是，某些 BSD 语义在 OS/400 实现中不可用，而且可能需要更改典型 BSD 套接字应用程序，它才能在系统上运行。

以下列表总结 OS/400 实现和 BSD 实现之间的差别。

/etc/hosts、/etc/services、/etc/networks 和 /etc/protocols
对于这些文件，OS/400 实现提供下列数据库文件，这些文件分别处理相同函数。

QUSRSYS 文件	内容
QATOCHOST	主机名和相应 IP 地址的列表。
QATOCPN	网络和相应 IP 地址的列表。
QATOCPP	在因特网中使用的协议的列表。
QATOCPS	服务以及服务使用的特定端口和协议的列表。

/etc/resolv.conf
OS/400 实现需要使用“iSeries 导航器”中的 TCP/IP 属性页面配置此信息。要访问 TCP/IP 属性页面，完成下列步骤：

1. 在“iSeries 导航器”上，展开 **iSeries 服务器 --> 网络 --> TCP/IP 配置**。
2. 右键单击 **TCP/IP 配置**。
3. 单击**属性**。

bind() 在 BSD 系统上，客户机可使用 socket() 创建 AF_UNIX 套接字，使用 connect() 连接至服务器并使用 bind() 将名称绑定至其套接字。OS/400 实现不支持此方案（bind() 失败）。

close()

OS/400 实现支持对 close() 使用拖延定时器，但不支持对基于 SNA 的 AF_INET 套接字使用该定时器。某些 BSD 实现不支持对 close() 使用拖延定时器。

connect()

在 BSD 系统上，如果对先前已连接至某个地址且正在使用无连接的传输服务的套接字发出 connect()，且使用了无效地址或无效地址长度，则该套接字不再处于已连接状态。OS/400 实现不支持此方案（connect() 失败且套接字仍处于已连接状态）。

已对其发出 connect() 的无连接的传输套接字可通过将 address_length 参数设置为零并发出另一 connect() 断开连接。

accept()、getsockname()、getpeername()、recvfrom() 和 recvmsg()

| 在使用 AF_UNIX 或 AF_UNIX_CCSID 地址系列且套接字未绑定时，缺省 OS/400 实现可能会返回地址长度零和未指定的地址结构。OS/400 BSD 4.4/UNIX 98 及其它实现可能返回一个很小的只指定了地址系列的地址结构。

ioctl()

- 在 BSD 系统上，对于类型为 SOCK_DGRAM 的套接字，FIONREAD 请求返回数据长度和地址长度。在 OS/400 实现上，FIONREAD 仅返回数据长度。
- 并非所有在 **ioctl()** 的大部分 BSD 实现上可用的请求在 **ioctl()** 的 OS/400 实现上也是可用的。

listen()

在 BSD 系统上，发出 **listen()** 并将储备参数设置为小于零的值不会产生错误。此外，在某些情况下，BSD 实现不使用储备参数或使用算法来得出储备值的最终结果。如果储备值小于零，OS/400 实现将返回错误。如果将储备设置为有效值，则会将该值用作储备。但是，如果将储备设置为大于 {SOMAXCONN} 的值，储备将缺省为在 {SOMAXCONN} 中设置的值。

带外（OOB）数据

在 OS/400 实现中，如果未设置 SO_OOBINLINE 且已接收到 OOB 数据，将不会废弃 OOB 数据，并且用户会将 SO_OOBINLINE 设置为开。初始 OOB 字节被视作普通数据。

socket() 的协议参数

作为提供附加安全性的手段，不允许任何用户创建指定协议 IPPROTO_TCP 或 IPPROTO_UDP 的 SOCK_RAW 套接字。

res_xlate() 和 res_close()

这些函数包括在 OS/400 实现的解析程序例程中。**res_xlate()** 对 DNS 信息包进行从 EBCDIC 至 ASCII 和从 ASCII 至 EBCDIC 的转换。**res_close()** 用于关闭由 **res_send()** 使用且设置了 RES_STAYOPEN 选项的套接字。它还会复位 _res 结构。

sendmsg() 和 recvmsg()

sendmsg() 和 **recvmsg()** 的 OS/400 实现允许使用最多（包括）{MSG_MAXIOVLEN} 个 I/O 个向量。BSD 实现允许使用 {MSG_MAXIOVLEN - 1} 个 I/O向量。

shutdown()

在 OS/400 实现上，当前在套接字描述符上被阻塞的输出函数在 **shutdown()** 之后仍然是阻塞的。在 BSD 实现上，阻塞的输出函数会结束，并带有 [EPIPE] 错误号值。相似的，当输入操作阻塞且从另一进程或线程发出 **shutdown()** 时，BSD 实现结束停止阻塞的输入操作，并带有零输出值。OS/400 实现只会使任何后续输入函数失效（带有零输出值），但阻塞的输入函数仍然被阻塞，直到接收到数据或采取一些其它操作取消其等待状态。

信号 在信号支持方面有一些差别:

- 每次接收到对输出操作发送了数据的确认信息时，BSD 实现会发出 SIGIO 信号。OS/400 套接字实现不会生成与出站数据有关的信号。

- SIGPIPE 信号的缺省操作是在 BSD 实现中结束（终止）该进程。为保留与 OS/400 的先前发行版的向下兼容性，OS/400 实现对 SIGPIPE 信号使用缺省操作即忽略。

SO_REUSEADDR 选项

在 BSD 系统上，使用地址系列 AF_INET 且类型为 SOCK_DGRAM 的套接字上的 **connect()** 将导致系统将套接字绑定至的地址更改为用于到达在 **connect()** 上指定的地址的接口的地址。例如，如果将类型为 SOCK_DGRAM 的套接字绑定至地址 INADDR_ANY，然后将其连接至地址 a.b.c.d，则系统会更改该套接字以便使其绑定至选择为将信息包路由选择至地址 a.b.c.d 的接口的 IP 地址。此外，例如，如果套接字绑定至的这一 IP 地址为 a.b.c.e，地址 a.b.c.e 将显示在 **getsockname()** 上而不是显示在 INADDR_ANY 上，而且必须使用 SO_REUSEADDR 选项将所有其它套接字绑定至地址为 a.b.c.e 的同一端口号。

比较起来，在此示例中，OS/400 实现不会将本地地址从 INADDR_ANY 更改为 a.b.c.e。在执行连接后，**getsockname()** 仍然返回 INADDR_ANY。

SO_SNDBUF 和 SO_RCVBUF 选项

在 BSD 系统上为 SO_SNDBUF 和 SO_RCVBUF 设置的值提供的控制级别比 OS/400 实现要高。在 OS/400 实现上，这些值被视作建议值。

UNIX 98 兼容性

Open Group 的一组开发人员和供应商创建的 UNIX 98 改进了在合并已为人所知的 UNIX 的大部分与因特网有关的函数时 UNIX 操作系统的不可操作性。OS/400 套接字是 V5R2 的新增内容，它使得程序员能够编写与 UNIX 98 操作环境相兼容的套接字应用程序。目前，IBM 支持大多数套接字 API 的两个版本。基本 OS/400 API 使用 BSD 4.3 结构和语法。其它 API 使用与 BSD 4.4 和 UNIX 98 编程接口规范相兼容的语法和结构。可通过将 _XOPEN_SOURCE 宏定义为值 520 或以上来选择 UNIX 98 兼容接口。

UNIX 98 兼容应用程序在地址结构上的差别

在指定 _XOPEN_OPEN 宏时，可使用在缺省 OS/400 实现中使用的相同地址系列编写 UNIX 98 兼容应用程序；但是，在 **sockaddr** 地址结构上有一些差别。下表对 BSD 4.3 **sockaddr** 地址结构与 UNIX 98 兼容地址结构进行比较:

表 17. BSD 4.3 与 UNIX 98/BSD 4.4 套接字地址结构的比较

BSD 4.3 结构	BSD 4.4/UNIX 98 兼容结构
sockaddr 地址结构	

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 地址结构	

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 地址结构	

表 17. BSD 4.3 与 UNIX 98/BSD 4.4 套接字地址结构的比较 (续)

<table>
<tr><td><pre>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;
};</pre></td><td><pre>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;
};</pre></td></tr>
<tr><td colspan="2">sockaddr_un 地址结构</td></tr>
<tr><td><pre>struct sockaddr_un {
 short   sun_family;
 char   sun_path[126];
};</pre></td><td><pre>struct sockaddr_un {
 uint8_t   sun_len;
 sa_family_t sun_family;
 char   sun_path[126]
};</pre></td></tr>
</table>

API 差别

在使用基于 ILE 的语言进行开发且使用 _XOPEN_SOURCE 宏编译应用程序时，某些套接字 API 会映射为内部名称。这些内部名称提供的函数与原始 API 相同。下表列示这些受影响的 API。如果在使用一些其它基于 C 的语言编写套接字应用程序，可直接写至这些 API 的内部名称。使用指向原始 API 的链接查看有关两个版本的 API 的使用注意事项和详细信息。

表 18. API 和 UNIX 98 等效名称

API 名称	内部名称
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()
getprotobynumber()	qso_getprotobynumber98()

表 18. API 和 UNIX 98 等效名称 (续)

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()

在进程间传递描述符 — sendmsg() 和 recvmsg()

在作业间传递打开的描述符的能力产生了一种设计客户机/服务器应用程序的新方法。在作业之间传递打开的描述符允许一个进程（通常是服务器）执行获取该描述所需的所有操作（打开文件，建立连接，等待 **accept()** API 完成）并允许另一进程（通常是工作程序）处理描述符打开后的所有数据传输操作。这一设计使得服务器和工作程序作业的逻辑更为简单。此设计还使得对不同类型的工作程序作业的支持更加容易。服务器可进行一次简单的检查，以确定哪些类型的工作程序应接收该描述符。

套接字提供可在服务器作业间传递描述符的三组 API:

- **spawn()**

  注: spawn() 不是套接字 API。它是作为与 OS/400 进程有关的 API 的一部分提供的。
- **givedescriptor()** 和 **takedescriptor()**
- **sendmsg()** 和 **recvmsg()**

spawn() API 启动新的服务器作业（通常称为“子作业”）并对该子作业指定特定描述符。如果子作业已经是活动的，则需要使用 **givedescriptor()** 和 **takedescriptor()** 或 **sendmsg()** 和 **recvmsg()** API。

但是，同 **spawn()** 和 **givedescriptor()** 和 **takedescriptor()** 相比，**sendmsg()** 和 **recvmsg()** API 有许多优点:

可移植性

givedescriptor() 和 **takedescriptor()** API 对 iSeries 是非标准和唯一的。如果 iSeries 与 UNIX 之间的应用程序可移植性是个问题，您可能想要转而使用 **sendmsg()** 和 **recvmsg()** API。

控制信息的通信

当工作程序作业接收描述符时，它通常需要知道类似如下的附加信息:

- 它是什么类型的描述符？
- 工作程序作业应对其作何处理？

sendmsg() 和 **recvmsg()** API 允许您传输数据，可能是控制信息及描述符；**givedescriptor()** 和 **takedescriptor()** API 则不允许您这样做。

性能

同使用 **givedescriptor()** 和 **takedescriptor()** API 的应用程序相比，使用 **sendmsg()** 和 **recvmsg()** API 执行的应用程序在三个方面略有优势:

- 所用时间
- CPU 利用率
- 可伸缩性

应用程序的性能改进程度取决于应用程序传递描述符的范围。

工作程序作业池

您可能想要设置工作程序作业池，以便服务器能够传递描述符且池中只有一个作业变成活动的并接收它。可使用 **sendmsg()** 和 **recvmsg()** API 完成此操作，方法是让所有工作程序作业等待共享描述符。当服务器调用 **sendmsg()** 时，只有一个工作程序作业接收该描述符。

未知工作程序作业标识

givedescriptor() API 要求服务器作业知道工作程序作业的作业标识符。通常工作程序作业会获取作业标识符并使用数据队列并将其传送至服务器作业。**sendmsg()** 和 **recvmsg()** 不需要额外开销就可以创建和管理此数据队列。

自适应服务器设计

在使用 **givedescriptor()** 和 **takedescriptor()** 设计服务器时，通常使用数据队列将作业标识符从工作程序作业传送至服务器。然后服务器执行 **socket()**、**bind()**、**listen()** 和 **accept()**。当 **accept()** API 完成时，服务器从数据队列推出下一个可用的作业标识。然后将入站连接传递至该工作程序作业。如果许多入局连接同时出现且没有足够的工作程序作业可用，就会发生问题。如果包含工作程序作业标识符的数据队列是空的，则服务器会阻塞，等待工作程序作业变成活动的，或者服务器会创建附加工作程序作业。在许多环境中，这些备用方法都不理想，原因是附加入局请求可能会填充侦听储备。

使用 **sendmsg()** 和 **recvmsg()** API 传递描述符的服务器在繁重活动期间仍然正常运行，原因是它们不需要知道哪个工作程序作业处理每个入局连接。当服务器调用 **sendmsg()** 时，入局连接的描述符和所有控制数据都会放入 AF_UNIX 套接字的内部队列。当工作程序作业可用时，它将调用 **recvmsg()** 并接收队列中的第一个描述符和控制数据。

非活动工作程序作业

givedescriptor() API 要求工作程序作业是活动的，虽然 **sendmsg()** API 不是活动的。调用 **sendmsg()** 的作业不需要关于工作程序作业的任何信息。**sendmsg()** API 只要求设置了 AF_UNIX 套接字连接。

以下是如何使用 **sendmsg()** API 将描述符传递至不存在的作业的示例:

服务器可使用 **socketpair()** API 创建一对 AF_UNIX 套接字，使用 **sendmsg()** API 通过 **socketpair()** 创建的 AF_UNIX 套接字之一发送描述符，然后调用 **spawn()** 来创建继承套接字对另一端的子作业。该子作业调用 **recvmsg()** 以接收服务器传递的描述符。当服务器调用 **sendmsg()** 时，子作业是不活动的。

一次传递多个描述符

givedescriptor() 和 **takedescriptor()** API 仅允许一次传递一个描述符。可使用 **sendmsg()** 和 **recvmsg()** API 传递一组描述符。

有关使用 **sendmsg()** 和 **recvmsg()** API 的样本程序，参见示例：在进程间传递描述符。

第 8 章 套接字方案：创建应用程序以接受 IPv4 和 IPv6 客户机

情况

假定您是一个套接字程序员，为一家专攻用于 iSeries 的套接字应用程序的应用程序开发公司工作。为超过它们的竞争者，您已经决定开发一套应用程序，它们使用 AF_INET6 地址系列，接受来自 IPv4 和 IPv6 的连接。您想要创建应用程序以处理来自 IPv4 和 IPv6 节点的请求。您知道 OS/400 支持 AF_INET6 地址系列套接字，并提供与 AF_INET 地址系列套接字的互操作性。您还知道可通过使用映射 IPv4 的 IPv6 地址格式来完成此任务。有关使用 IPv6 和 IPv4 应用程序之间的互操作性的更多详细信息，参见 IPv6 应用程序与 IPv4 应用程序的兼容性。

方案目标

此方案有下列目的：

1. 创建用来接受和处理来自 IPv6 和 IPv4 客户机的请求的服务器应用程序
2. 创建请求 IPv4 或 IPv6 服务器应用程序中的数据的客户机应用程序

必备步骤

在开发满足这些目的的应用程序之前，必须完成下列任务：

1. 安装 QSYSINC 库。此库提供编译套接字应用程序时需要的必要头文件。
2. 安装 C 编译器许可程序（5722–CX2）。
3. 安装和配置 2838 以太网卡。有关以太网选项的信息，参见“信息中心”中的以太网主题。
4.

   设置 TCP/IP 和 IPv6 网络。

方案详细信息

下图描述将为其创建用来处理来自 IPv6 和 IPv4 客户机的请求的应用程序的 IPv6 网络。iSeries 包含将用来侦听和处理来自这些客户机的请求的程序。网络由两个不同的域构成，一个只包含 IPv4 客户机，另一远程网络只包含 IPv6 客户机。iSeries 的域名为 myserver.myco.com。服务器应用程序会使用 AF_INET6 地址系列以处理在 **bind()** 函数调用上指定了 in6addr_any 的入局请求。

© Copyright IBM Corp. 2001, 2002

参见此方案中使用的下列示例程序:

- 示例: 接受来自 IPv6 和 IPv4 客户机的连接
- 示例: IPv4 或 IPv6 客户机

示例: 接受来自 IPv6 和 IPv4 客户机的连接

使用此样本程序创建服务器／客户机模型，以接受来自 IPv4（使用 AF_INET 地址系列的套接字应用程序）和 IPv6（使用 AF_INET6 地址系列的应用程序）的请求。目前套接字应用程序可能只使用 AF_INET 地址系列，它允许使用 TCP 和 UDP 协议；但是，这种情况可能会随着使用 IPv6 地址的情况的增加而有所改变。可使用此样本程序创建您自己的应用程序以同时容纳两个地址系列。

下图显示此示例程序如何工作:

套接字事件流：接受来自 IPv4 和 IPv6 客户机的请求的服务器应用程序

此流描述每个函数调用以及它们在接受 IPv4 和 IPv6 客户机的请求的套接字应用程序中执行的操作。

1. **socket()** API 指定创建端点的套接字描述符。它还指定支持 IPv6 的 AF_INET6 地址系列，且 TCP 传输（SOCK_STREAM）将用于此套接字。
2. **setsockopt()** 函数允许在必需的等待时间到期之前让应用程序在服务器重新启动时重复使用本地地址。

3. **bind()** 函数为套接字提供唯一名称。在此示例中，程序员将地址设置为 in6addr_any，这（缺省情况下）允许从指定端口 3005 的任何 IPv4 或 IPv6 客户机建立连接。（即，同时对 IPv4 和 IPv6 端口空间执行绑定）。

   注： 如果服务器只需要处理 IPv6 客户机，可使用 IPV6_ONLY 套接字选项。
4. **listen()** 函数允许服务器接受入局客户机连接。在此示例中，程序员将储备设置为零，这允许系统在开始拒绝入局请求之前对 10 个连接请求进行排队。
5. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待来自 IPv4 或 IPv6 客户机的入局连接到达。
6. **getpeername()** 函数将客户机的地址返回给应用程序。如果客户机是 IPv4 客户机，该地址将显示为映射 IPv4 的 IPv6 地址。
7. **recv()** 函数从客户机接收 250 字节的数据。在此示例中，程序员知道客户机将发送 250 字节的数据。既然如此，程序员会使用 SO_RCVLOWAT 套接字选项并指定在所有 250 字节数据都到达之前不要唤醒 **recv()** 。
8. **send()** 函数将数据回传至客户机。
9. **close()** 函数关闭所有打开的套接字描述符。

套接字事件流：来自 IPv4 或 IPv6 客户机的请求

注： 此客户机示例可与希望接受 IPv4 或 IPv6 节点的请求的其它服务器应用程序设计配合使用。象示例：面向连接的设计中描述的那些其它服务器设计都可与此客户机示例配合使用。

1. **inet_pton()** 调用将地址的文本格式转换为二进制格式。在此示例中，将发出其中两个调用。第一个调用确定服务器是否为有效 AF_INET 地址。第二个 **inet_pton()** 调用确定服务器是否有 AF_INET6 地址。如果是数字的，则需要阻止 **getaddrinfo()** 执行任何名称解析。否则在发出 **getaddrinfo()** 调用时，需要解析提供的主机名。
2. **getaddrinfo()** 调用检索后续 **socket()** 和 **connect()** 调用所需的地址信息。
3. **socket()** 函数返回表示端点的套接字描述符。该语句还使用从 **getaddrinfo()** 返回的信息标识地址系列、套接字类型和协议。
4. **connect()** 函数建立与服务器的连接而不考虑服务器是 IPv4 还是 IPv6。
5. **send()** 函数将数据请求发送至服务器。
6. **recv()** 函数从服务器应用程序接收数据。
7. **close()** 函数关闭所有打开的套接字描述符。

以下样本代码显示此方案的服务器应用程序。有关客户机应用程序，参见示例：IPv4 或 IPv6 客户机。有关使用此代码的信息，参见代码不保证声明信息。

```
/**************************************************************************/
/* 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);
}
```

示例: IPv4 或 IPv6 客户机

该样本程序可与接受来自 IPv4 或 IPv6 客户机的服务器应用程序配合使用。

```
/**************************************************************************/
/* 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);
}
```

第 9 章 套接字应用程序设计建议

在使用套接字应用程序之前，先评估套接字应用程序的功能需求、目标和需要。还应考虑应用程序的性能需求和系统资源影响。以下建议列表将帮助您解决套接字应用程序的其中一些问题，并指出使用套接字和设计套接字应用程序的较好方法：

表 19. 套接字设计应用程序

建议	原因	最好用于
使用异步 I/O	在线程式服务器模型中使用的异步 I/O 比经常使用的 select() 模型更为可取。有关使用异步 I/O 的优点的更多信息，参见异步 I/O。有关使用异步 I/O API 的样本程序，参见示例：使用异步 I/O。	将处理大量并行客户机的套接字服务器应用程序。
在使用异步 I/O 时，将进程中的线程数调整为要处理的客户机数目的最优数目。	如果定义的线程数太少，一些客户机可能未作处理就超时了。如果定义的线程数太多，一些系统资源可能得不到有效利用。 **注：** 线程数过多比线程数过少要好一些。	使用异步 I/O 的套接字应用程序。
设计套接字应用程序以避免对异步 I/O 的所有启动操作使用 postflag。	避免在操作同步完成的情况下传送至完成端口的性能开销。	使用异步 I/O 的套接字应用程序。
使用 **send()** 和 **recv()** 而不使用 **read()** 和 **write()**。	**send()** 和 **recv()** API 在 **read()** 和 **write()** 的基础上对性能和可服务性有所改进。	知道使用套接字描述符而不是文件描述符的任何套接字程序。
使用接收低水位（SO_RCVLOWAT）套接字选项以避免在所有数据到达之前形成接收操作循环。	允许应用程序在完成阻塞的接收操作之前等待要在套接字上接收的最小量的数据。	接收数据的任何套接字应用程序
使用 MSG_WAITALL 标志以避免在所有数据到达之前形成接收操作循环。	允许应用程序在完成阻塞的接收操作之前等待接收在接收操作上提供的整个缓冲区。	接收数据并且预先知道期望到达的数据量的任何套接字应用程序。
使用 **sendmsg()** 和 **recvmsg()** 而不使用 **givedescriptor()** 和 **takedescriptor()**。	有关使用它们的优点，参见在进程间传递描述符 — sendmsg() 和 recvmsg()。有关使用 **sendmsg()** 和 **recvmsg()** 的样本程序，参见示例：在进程间传递描述符。	在进程间传递套接字或文件描述符的任何套接字应用程序。
在使用 **select()** 时，尝试避免在读写或异常集合中使用大量描述符。 **注：** 如果对 **select()** 处理使用了大量描述符，则参见上述异步 I/O 建议。	如果读写或异常集合中有大量描述符，每次调用 **select()** 时就会产生大量冗余工作。一旦完成 **select()**，还必须执行实际的套接字函数，也就是说还必须执行读写或接受操作。异步 I/O API 将套接字上发生的情况的通知与实际 I/O 操作组合起来。	用于 **select()** 的活动描述符数目很大（超过 50）的应用程序。
在使用 **select()** 之前保存读写和异常集合的副本，以避免每次必须重新发出 **select()** 时重新构建这些集合。	这样可以在每次计划发出 **select()** 时节省重新构建读写或异常集合的开销。有关使用 **select()** 的样本程序，参见示例：非阻塞 I/O 和 select()。	在其中将 **select()** 与对读写或异常处理启用的大量套接字描述符配合使用的任何套接字应用程序。

© Copyright IBM Corp. 2001, 2002

表 19. 套接字设计应用程序 (续)

不要在将 **select()** 用作定时器。而是使用 **sleep()**。 **注**: 如果 **sleep()** 定时器的粒度不足，可能必须将 **select()** 用作定时器。在这种情况下，将最大描述符设置为零，将读写和异常集合设置为 NULL。	定时器响应较好，系统开销较少。	在其中仅将 **select()** 用作计时器的任何套接字应用程序。
如果套接字应用程序使用 DosSetRelMaxFH() 增加了每个进程允许使用的文件和套接字描述符的最大数目，且您正在同一应用程序中使用 **select()**，则应注意这一新的最大值对用于 **select()** 处理的读写和异常集合的大小的影响。	如果在 FD_SETSIZE 指定的读写或异常集合范围外分配描述符，可能会覆盖和破坏存储器。确保集合大小至少大到足以处理为进程设置的最大数目的描述符和在 **select()** API 上指定的最大描述符值。	在其中使用 DosSetRelMaxFH() 和 **select()** 的任何应用程序或进程。
将读或写集合中的所有套接字描述符设置为非阻塞。对读或写启用描述符时，循环和消耗或发送所有数据直到返回 EWOULDBLOCK。有关使用 **select()** 的样本程序，参见示例：非阻塞 I/O 和 select()。	这将允许您在仍然可对描述符处理或读取数据时将 **select()** 调用的数目减至最少。	在其中使用 **select()** 的任何套接字应用程序。
仅指定需要用于 **select()** 处理的集合。	大部分应用程序不需要指定异常集合或写集合。	在其中使用 **select()** 的任何套接字应用程序。
使用 GSKit API 而不是 SSL API。	Global Secure Toolkit（GSKit）和 OS/400 SSL_ API 允许开发安全 AF_INET 或 AF_INET6、SOCK_STREAM 套接字应用程序。因为 GSKit API 在 IBM@server 平台上是受支持的，所以它是保护应用程序的一组最佳 API。SSL_ API 仅对于 OS/400 系统才是本地的。	需要对 SSL/TLS 处理启用的任何套接字应用程序。
避免使用信号。	信号的性能开销（在所有平台上，而不仅仅是 iSeries）是非常昂贵的。最好将套接字应用程序设计为使用异步 I/O 或 **select()** API。	考虑在其套接字应用程序中使用信号的任何程序员。
尽量使用独立于例程的协议，如 **inet_ntop()**、**inet_pton()**、**getaddrinfo()** 和 **getnameinfo()**。	即使还不准备支持 IPv6，还是使用这些 AIP（而不是 **inet_ntoa()**、**inet_addr()**、**gethostbyname()** 和 **gethostbyaddr()**）以便易于迁移。	使用网络例程的任何 AF_INET 或 AF_INET6 应用程序。
使用 sockaddr_storage 声明用于任何地址系列地址的存储器。	简化编写可跨多个地址系列和平台移植的代码的过程。声明足够的存储量以容纳最大的地址系列并确保正确的边界对齐。	存储地址的任何套接字应用程序。

第 10 章 示例：套接字应用程序设计

下列示例提供了许多样本程序，以举例说明较高级的套接字概念。可使用这些样本程序创建您自己的应用程序来完成类似的任务。除了这些示例之外，还有一些图形和举例说明每个程序的的事件流的调用列表。可以交互方式使用 Xsocket 工具在这些程序中尝试其中一些 API，或者可针对特定环境作出特定更改。

- 示例：面向连接的设计
- 示例：建立安全连接
- 示例：对线程安全网络例程使用 gethostbyaddr_r()
- 示例：非阻塞 I/O 和 select()
- 示例：将信号与阻塞套接字 API 配合使用
- 示例：将多点广播与 AF_INET 地址系列配合使用
- 示例：查询和更新 DNS
- 示例：使用 send_file() 和 accept_and_recv() API 传输文件数据

示例：面向连接的设计

有几种方法可用来在 iSeries 上设计面向连接的套接字服务器。下面的示例程序可用来创建您自己的面向连接的设计。虽然可使用附加套接字服务器设计，但下列示例中提供的设计是最常用的：

迭代服务器

在迭代服务器示例中，单个服务器作业处理所有入局连接和带有客户机作业的所有数据流。当 **accept()** API 完成时，服务器处理整个事务。这是要开发的最简单的服务器，但还是有一些问题。当服务器在处理来自给定客户机的请求时，附加客户机可能会尝试连接服务器。这些请求会填充 **listen()** 储备，而且其中某些请求最终会被拒绝。

所有余下示例都是并行服务器设计。在这些设计中，系统使用多个作业和线程来处理入局连接请求。有了并行服务器，通常会有多台客户机同时连接至服务器。

对于网络中的多台并行客户机，建议使用异步 I/O 套接字 API。这些 API 在具有多台并行客户机的网络中提供最佳性能。异步 I/O 描述这些 API 执行了哪些操作以及它们如何工作。有关使用这些 API 的示例程序，参见示例：使用异步 I/O。

spawn() 服务器和 spawn() 工作程序

spawn() 服务器和 spawn() 工作程序示例使用 spawn() API 来创建新作业以处理每个入局请求。spawn() 完成后，服务器可以在 **accept()** API 上等待要接收的下一个入局连接。

此服务器设计的唯一问题就是每次接收连接时创建新作业的性能开销。可通过使用预先启动的作业避免 spawn() 服务器示例的性能开销。对已经活动的作业指定入局连接，而不是每次接收连接时创建新作业。本主题中的所有余下示例使用预先启动的作业。

sendmsg() 服务器和 recvmsg() 工作程序

sendmsg() 服务器和 recvmsg() 工作程序示例将入局连接传递至工作程序（客户机）作业。在服务器作业第一次启动时，服务器会预先启动所有工作程序作业。

多 accept() 服务器和多 accept() 工作程序

在先前的示例中，在服务器接收入局连接请求之前并未涉及工作程序作业。系统的多 accept() 服务器和多 accept() 工作程序示例将每个工作程序作业转移到迭代服务器中。服务器作业仍然调用 **socket()**、

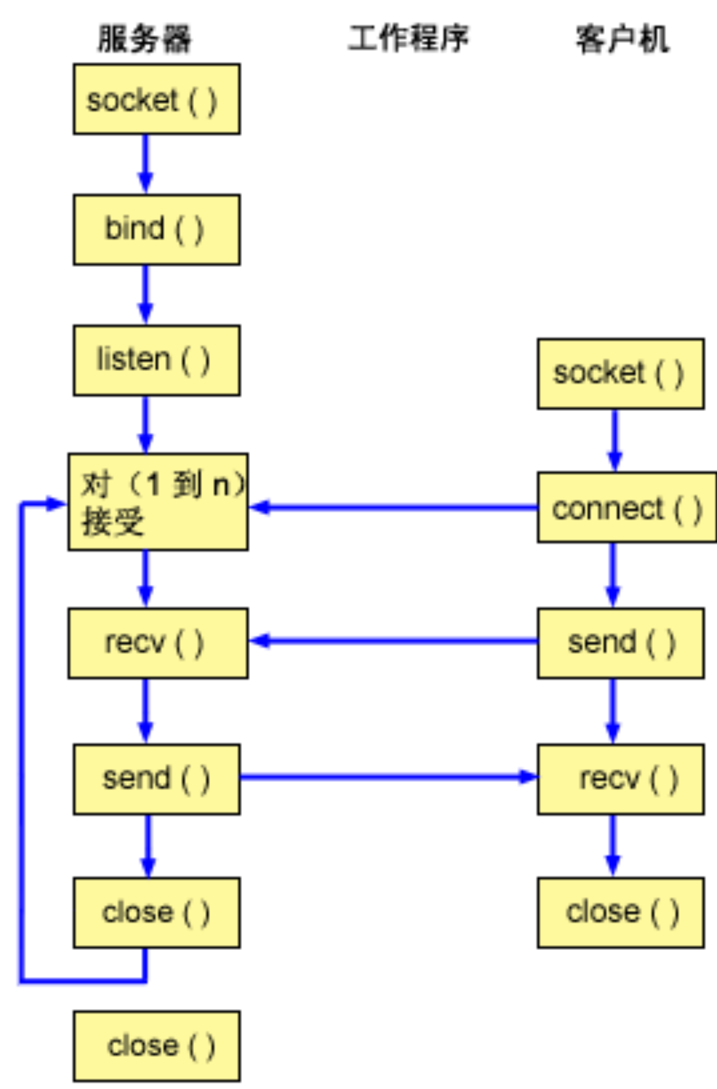
© Copyright IBM Corp. 2001, 2002

bind() 和 **listen()** API。**listen()** 调用完成时，服务器会创建每个工作程序作业并对其指定侦听套接字。于是所有工作程序作业调用 **accept()** API。当客户机试图连接至服务器时，只完成一个 **accept()** 调用，并由该工作程序处理连接。

所有这些示例都使用客户机连接的基本设计。有关详细信息，参见示例：一般客户机。

示例：编写迭代服务器程序

使用此示例来创建处理所有入局连接的单个服务器作业。当 **accept()** API 完成时，服务器处理整个事务。该图举例说明了系统使用迭代服务器设计时服务器和客户机作业如何进行交互。有关包含可与此示例配合使用的常见客户机作业的代码的示例，参见示例：一般客户机。



套接字事件流：迭代服务器

以下套接字调用序列提供图形的描述。它还描述服务器与工作程序应用程序之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。有关此流的客户机部分的描述，参见示例：一般客户机。以下序列显示迭代服务器应用程序的这一样本程序的函数调用：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。
3. **listen()** 允许服务器接受入局客户机连接。
4. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
5. **recv()** 函数从客户机应用程序接收数据。
6. **send()** 函数将数据回传至客户机。
7. **close()** 函数关闭所有打开的套接字描述符。

```
/**************************************************************************/
/* 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);
```

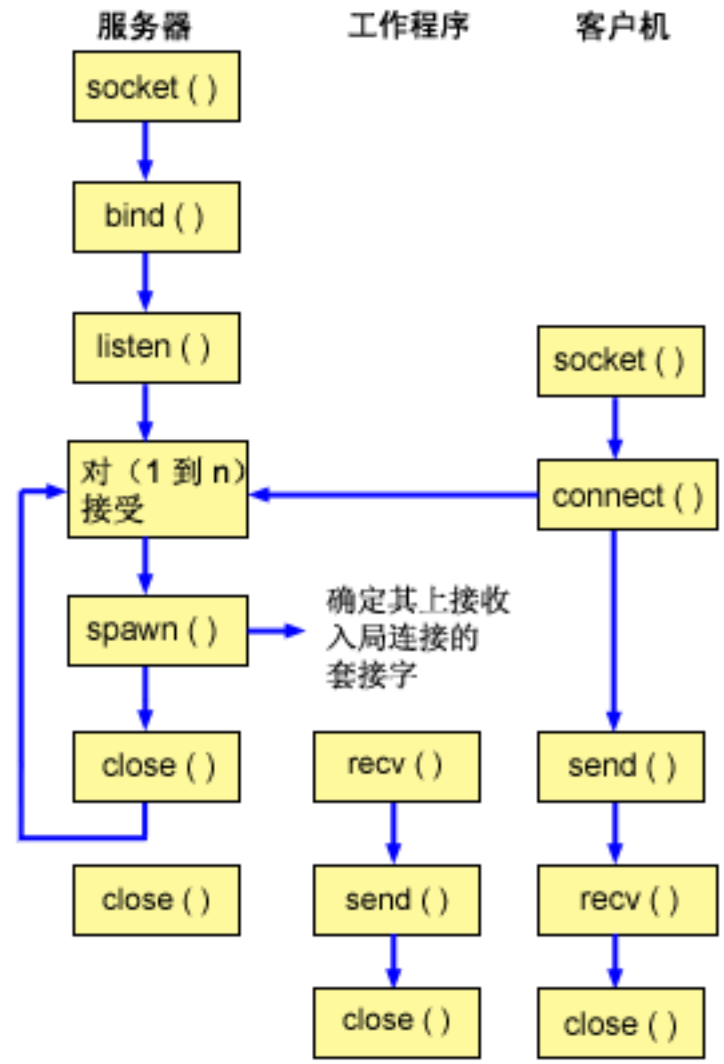
```
      if (rc <= 0)
      {
         perror("send() failed");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }

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

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

示例：使用 spawn() API 创建子进程

此示例显示服务器程序如何使用 **spawn()** API 创建从父代继承套接字描述符的子进程。服务器作业等待入局连接，然后调用 **spawn()** 以创建子作业以处理入局连接。子进程使用 **spawn()** 函数继承下列属性:

- 套接字和文件描述符
- 信号掩码
- 信号操作向量
- 环境变量

| 下图举例说明了使用 **spawn()** 服务器设计时，服务器、工作程序和客户机作业如何进行交互。
|



|

| **套接字事件流：使用 spawn() 接受和处理请求的服务器**

| 以下套接字调用序列提供图形的描述。它还描述服务器与工作程序示例之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：创建使用 spawn() 的服务器使用下列套接字调用创建带有 **spawn()** 函数调用的子进程：

| 1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
| 2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。
| 3. **listen()** 允许服务器接受入局客户机连接。
| 4. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
| 5. **spawn()** 函数初始化工作作业的参数以处理入局请求。在此示例中，新连接的套接字描述符在子程序中映射为描述符零。
| 6. 在此示例中，第一个 **close()** 函数关闭侦听套接字描述符。第二个 **close ()** 调用结束接受的套接字。

| **套接字事件流：spawn() 创建的工作程序作业**

| 示例：允许工作程序作业接收数据缓冲区使用以下函数调用序列：

1. 在服务器上调用 **spawn()** 函数之后，**recv()** 函数从入局连接接收数据。
2. **send()** 函数将数据回传至客户机。
3. **close()** 函数结束生成的工作程序作业。

示例：创建使用 spawn() 的服务器

此示例显示如何使用 **spawn()** API 创建从父代继承套接字描述符的子进程。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

有关使用套接字描述符完成进程的的样本程序，参见示例：允许工作程序作业接收数据缓冲区。

示例：允许工作程序作业接收数据缓冲区

此示例包含允许工作程序作业从客户机作业接收数据缓冲区并将其回传的代码。有关使用代码示例的信息，参见代码不保证声明。

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

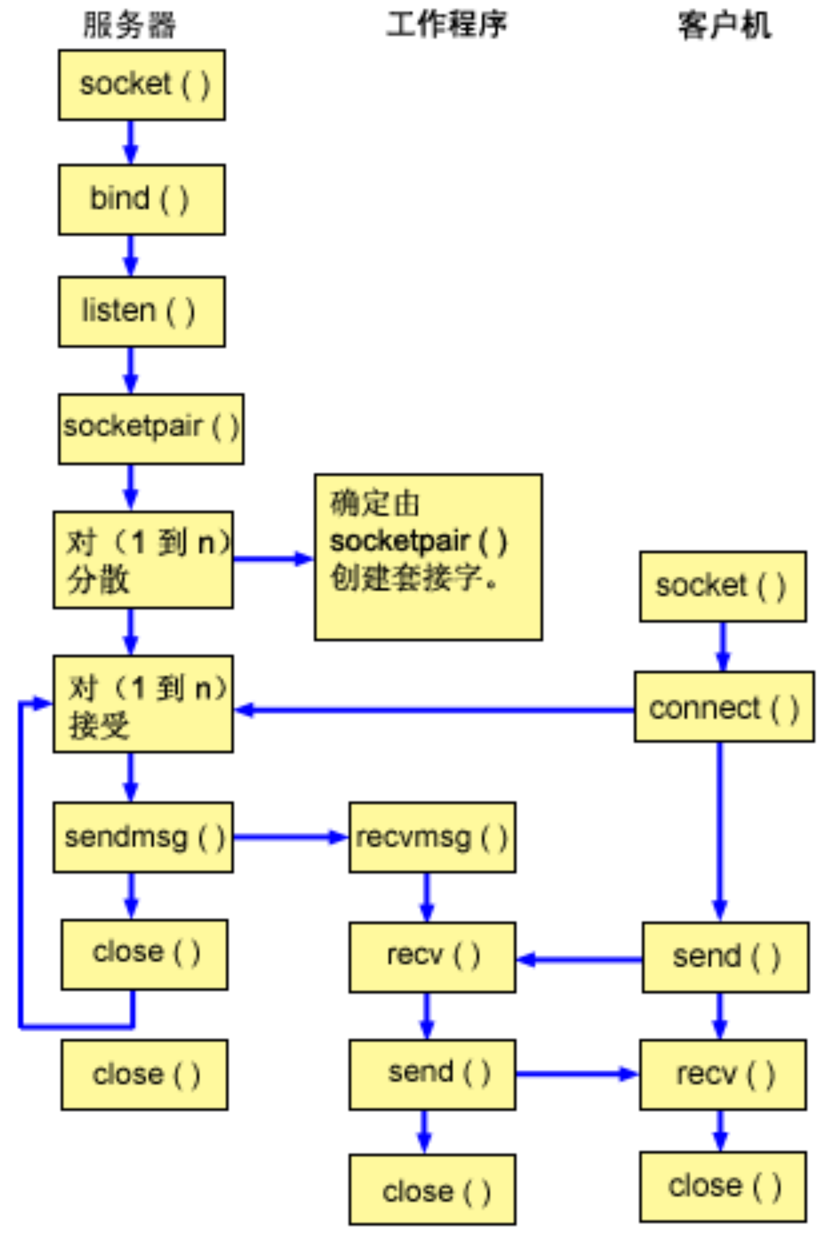
```
   printf("Echo it back\n");
   len = rc;
   rc = send(sockfd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("send() failed");
      close(sockfd);
      exit(-1);
   }

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

}
```

示例：在进程间传递描述符

sendmsg() 和 **recvmsg()** 示例显示如何设计服务器程序以使用这些 API 处理入局连接。当服务器启动时，它会创建工作程序作业池。这些预先分配的（生成的）工作程序作业将等待直到需要为止。当客户机作业连接至服务器时，服务器将入局连接指定给其中一个工作程序。

下图举例说明了当系统使用 **sendmsg()** 和 **recvmsg()** 服务器设计时，服务器、工作程序和客户机作业如何进行交互。

| **注:** 有关此图的客户机部分的描述，参见示例：一般客户机。
|



|

| **套接字事件流：使用 sendmsg() 和 recvmsg() 函数的服务器**
| 以下套接字调用序列提供图形的描述。它还描述服务器与工作程序示例之间的关系。每一组流包含指向有关特
| 定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：用于
| sendmsg() 和 recvmsg() 的服务器程序使用下列套接字调用来创建带有 **sendmsg()** 和 **recvmsg()** 函数调用的
| 子进程：

| 1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输
| （SOCK_STREAM）的 INET（网际协议）地址系列。
| 2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。
| 3. **listen()** 允许服务器接受入局客户机连接。
| 4. **socketpair()** 函数创建一对 UNIX 数据报套接字。服务器可使用 **socketpair()** API 来创建一对 AF_UNIX
| 套接字。

5. **spawn()** 函数初始化工作作业的参数以处理入局请求。在此示例中，创建的子作业继承 **socketpair()** 创建的套接字描述符。
6. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
7. **sendmsg()** 函数将入局连接发送至其中一个工作程序作业。子进程使用 **recvmsg()** 函数接受该连接。当服务器调用 **sendmsg()** 时，子作业不是活动的。
8. 在此示例中，第一个 **close()** 函数关闭接受的套接字。第二个 **close ()** 调用结束侦听套接字。

套接字事件流：使用 recvmsg() 的工作程序作业

示例：用于 sendmsg () 和 recvmsg () 的工作程序使用以下函数调用序列:

1. 服务器接受连接并将其套接字描述符传递至工作程序作业后，**recvmsg()** 函数接收该描述符。在此示例中，**recvmsg()** 函数将等待直到服务器发送描述符为止。
2. **recv()** 函数从客户机接收消息。
3. **send()** 函数将数据回传至客户机。
4. **close()** 函数结束工作程序作业。

示例：用于 sendmsg() 和 recvmsg() 的服务器程序

此示例显示如何使用 **sendmsg()** API 创建工作程序作业池。有关包含可与此示例配合使用的常见客户机作业的代码的示例，参见示例：一般客户机。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例: 用于 sendmsg () 和 recvmsg () 的工作程序

此示例显示如何使用 **recvmsg()** API 客户机作业来接收工作程序作业。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

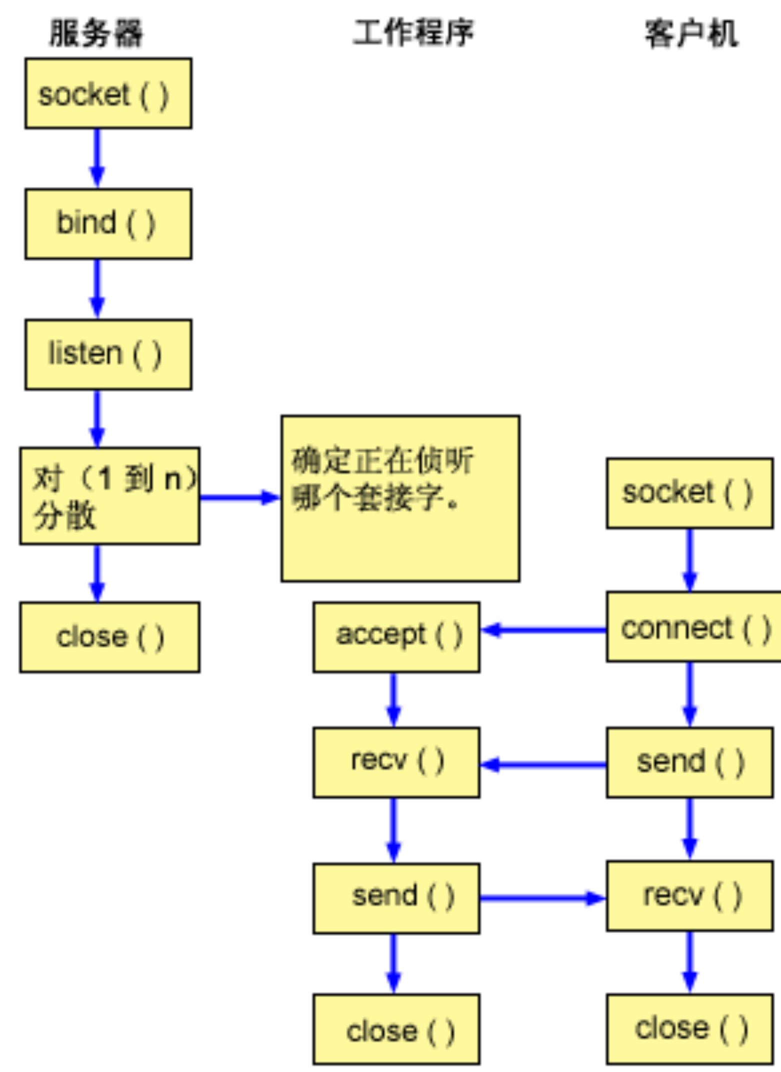
```
    /*************************************************/
    /* Close down the descriptors                    */
    /*************************************************/
    close(worker_sd);
    close(pass_sd);
}
```

示例: 使用多 accept() API 处理入局请求

这些示例显示如何设计服务器程序以使用多 **accept()** 模型来处理入局连接请求。当多 **accept()** 服务器启动时，它通常会执行 **socket()**、**bind()** 和 **listen()**。然后创建工作程序作业池并为每个工作程序作业指定侦听套接字。于是每个多 **accept()** 工作程序调用 **accept()**。

下图举例说明当系统使用多 **accept()** 服务器设计时，服务器、工作程序和客户机作业如何进行交互。

注: 有关此图的客户机部分的描述，参见示例: 一般客户机。



套接字事件流：创建多 accept() 工作程序作业池的服务器

以下套接字调用序列提供图形的描述。它还描述服务器与工作程序示例之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：用来创建多 accept() 工作程序作业池的服务器程序使用下列套接字调用创建子进程:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。
3. **listen()** 允许服务器接受入局客户机连接。
4. **spawn()** 函数创建每个工作程序作业。
5. 在此示例中，第一个 **close()** 函数关闭侦听套接字。

套接字事件流：多 accept() 的工作程序作业

示例：多 accept() 的工作程序作业使用以下函数调用序列:

1. 服务器生成工作程序作业后，会将侦听套接字描述符作为命令行参数传递至此工作程序作业。**accept()** 函数等待入局客户机连接。
2. **recv()** 函数从客户机接收消息。
3. **send()** 函数将数据回传至客户机。
4. **close()** 函数结束工作程序作业。

示例：用来创建多 accept() 工作程序作业池的服务器程序

此示例显示如何使用多 **accept()** 模型创建工作程序作业池。有关包含可与此示例配合使用的常见客户机作业的代码的示例，参见示例：一般客户机。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例：多 accept() 的工作程序作业

此示例显示多 **accept()** API 如何接收工作程序作业和调用 **accept()** 服务器。有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
}
```

示例：一般客户机

下列代码示例包含一般客户机作业的代码。客户机作业执行 **socket()**、**connect()**、**send()**、**recv()** 和 **close()**。客户机作业不知道它发送和接收的数据缓冲区是针对工作程序作业而不是针对服务器的。如果打算创建的客户机应用程序在服务器是 AF_INET 地址系列或 AF_INET6 地址系列时起作用，则使用示例：IPv4 或 IPv6 客户机。

此客户机作业将使用每一个常用的面向连接的服务器设计：

- 迭代服务器。有关样本程序，参见示例：编写迭代服务器程序。
- 生成的服务器和工作程序。有关样本程序，参见示例：使用 spawn() API 创建子进程。
- sendmsg() 服务器和 rcvmsg() 工作程序。有关样本程序，参见示例：用于 sendmsg() 和 recvmsg() 的服务器程序。
- 多 accept() 设计。有关样本程序，参见示例：用来创建多 accept() 工作程序作业池的服务器程序。
- 非阻塞 I/O 和 select() 设计。有关样本程序，参见示例：非阻塞 I/O 和 select()。
- 接受来自 IPv4 或 IPv6 客户机的连接的服务器。有关样本程序，参见示例：接受来自 IPv6 和 IPv4 客户机的连接。

套接字事件流：一般客户机

以下样本程序使用以下函数调用序列：

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. 接收到套接字描述符后，使用 **connect()** 函数来建立与服务器的连接。
3. **send()** 函数将数据缓冲区发送至工作程序作业。
4. **recv()** 从工作程序作业接收数据缓冲区。
5. **close()** 函数关闭所有打开的套接字描述符。

有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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);
```

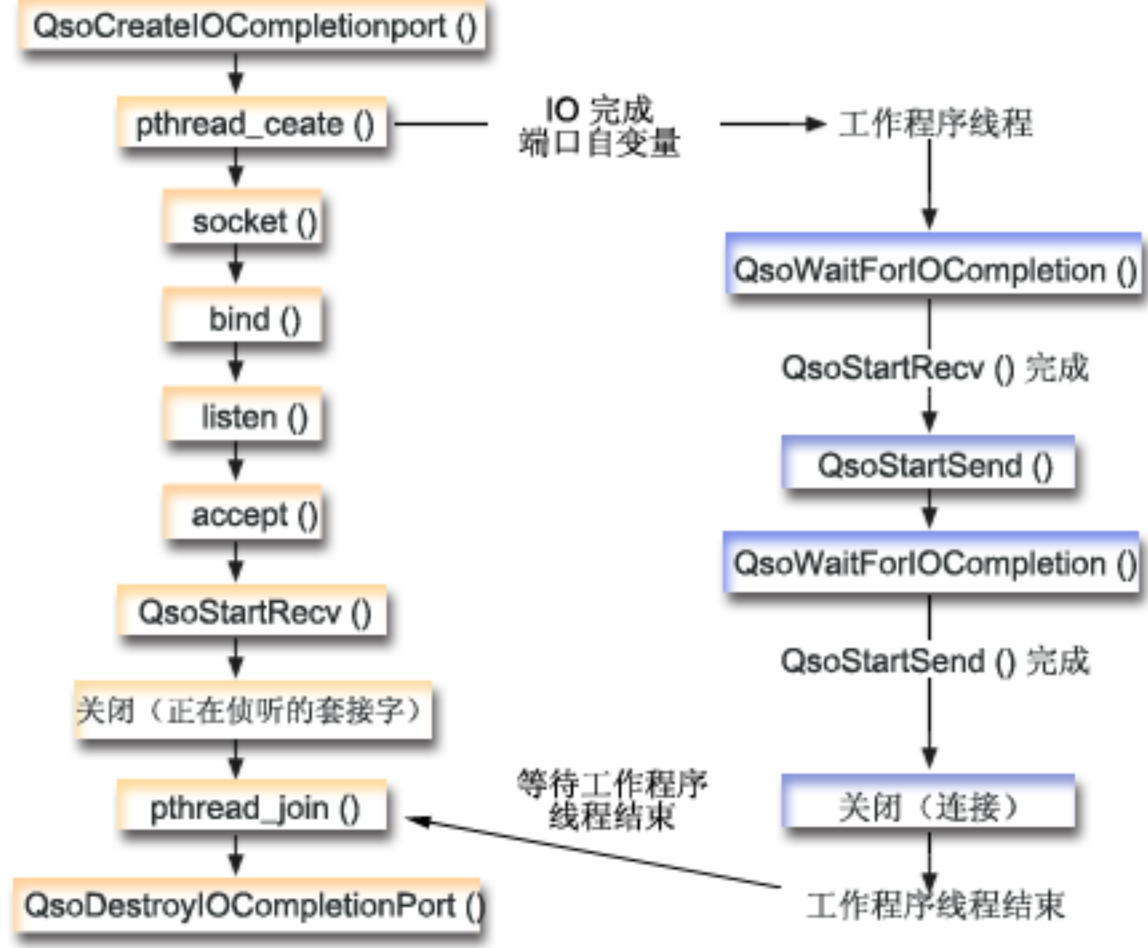
```
   /**************************************************/
   /* Receive data buffer from the worker job       */
   /**************************************************/
   len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
   if (len != strlen(send_buf) + 1)
   {
      perror("recv");
      close(sockfd);
      exit(-1);
   }
   printf("%d bytes received\n", len);

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

示例：使用异步 I/O

应用程序将使用 **QsoCreateIOCompletionPort()** API 创建 I/O 完成端口。此 API 将返回一个句柄，可使用该句柄安排和等待异步 I/O 请求的完成。该应用程序将启动输入或输出函数，指定 I/O 完成端口句柄。当 I/O 完成时，会对指定 I/O 完成端口公布状态信息和应用程序定义的句柄。对 I/O 完成端口的公布将准确地唤醒正在等待的可能的多个线程的其中一个线程。应用程序接收：

- 在原始请求上提供的缓冲区
- 送至该缓冲区或发自该缓冲区的已处理数据的长度
- 指示完成的 I/O 操作的类型
- 以及在初始 I/O 请求上传递的应用程序定义的句柄

此应用程序句柄可能只是标识客户机连接的套接字描述符，或是指向包含有关客户机连接的状态的大量信息的存储器的指针。自从操作完成且传递了应用程序句柄，工作程序线程就确定了完成客户机连接的下一个步骤。处理这些完成的异步操作的工作程序线程可以处理多个不同的客户机请求，而不是只处理一个客户机请求。由于复制至用户缓冲区与从用户缓冲区复制对服务器进程是异步进行的，所以客户机请求的等待时间就减少了。这对有多个处理器的系统非常有利。



套接字事件流：异步 I/O 服务器

以下套接字调用序列提供图形的描述。它还描述服务器与工作程序示例之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。此流描述以下样本应用程序中的套接字调用。将此服务器示例与一般客户机示例配合使用。

1. 主线程通过调用 **QsoCreateIOCompletionPort()** 创建 I/O 完成端口
2. 主线程使用 **pthread_create** 函数创建工作程序线程池以处理所有 I/O 完成端口请求。
3. 工作程序线程调用 **QsoWaitForIOCompletionPort()**，它将等待要处理的客户机请求。
4. 主线程接受客户机连接，然后发出 **QsoStartRecv()**，它将指定工作程序线程等待的 I/O 完成端口。

   注： 还可通过使用 **QsoStartAccept()** 来使用异步接受。
5. 有时客户机请求异步到达服务器进程。套接字操作系统装入提供的用户缓冲区，并将完成的 **QsoStartRecv()** 请求发送至指定 I/O 完成端口。一个工作程序线程被唤醒并继续处理此请求。
6. 工作程序线程通过执行 **QsoStartSend()** 操作从应用程序定义的句柄抽取客户机套接字描述符并继续将接收到的数据回传至客户机。
7. 如果可立即发送数据，则 **QsoStartSend()** 返回实际情况的指示信息，否则套接字操作系统会尽快发送数据并对指定 I/O 完成端口公布实际情况的指示信息。工作程序线程获取发送数据的指示信息并在 I/O 完成端口上等待另一请求或终止（如果指示这样做的话）。主线程可使用 **QsoPostIOCompletion()** 来公布工作程序线程终止事件。

8. 主线程等待工作程序线程完成，然后通过调用 **QsoDestroyIOCompletionPort()** 破坏 I/O 完成端口。

注: 此服务器示例使用在示例：一般客户机中描述的常用客户机代码。

此示例显示服务器程序如何使用异步 API。有关使用代码示例的信息，参见代码不保证声明。

```
      #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 */
```

示例: 建立安全连接

可使用 Global Secure ToolKit（GSKit）API 或 SSL_ API 创建安全服务器和客户机。GSKit API 是首选方法，原因是它们提供了跨 IBM @server 平台的安全连接。SSL_API 仅对 OS/400 才是本地的。每一组安全套接字 API 都有返回码，这将帮助您在建立安全套接字连接时标识错误。有关访问有关这些错误消息的信息的详细信息，参见安全套接字 API 错误代码消息。

下列示例说明如何使用每种方法建立安全服务器和客户机。

- 示例: 使用异步数据接收的 GSKit 安全服务器
- 示例: 使用异步握手的 GSKit 安全服务器
- 示例: 使用 GSKit API 建立安全客户机

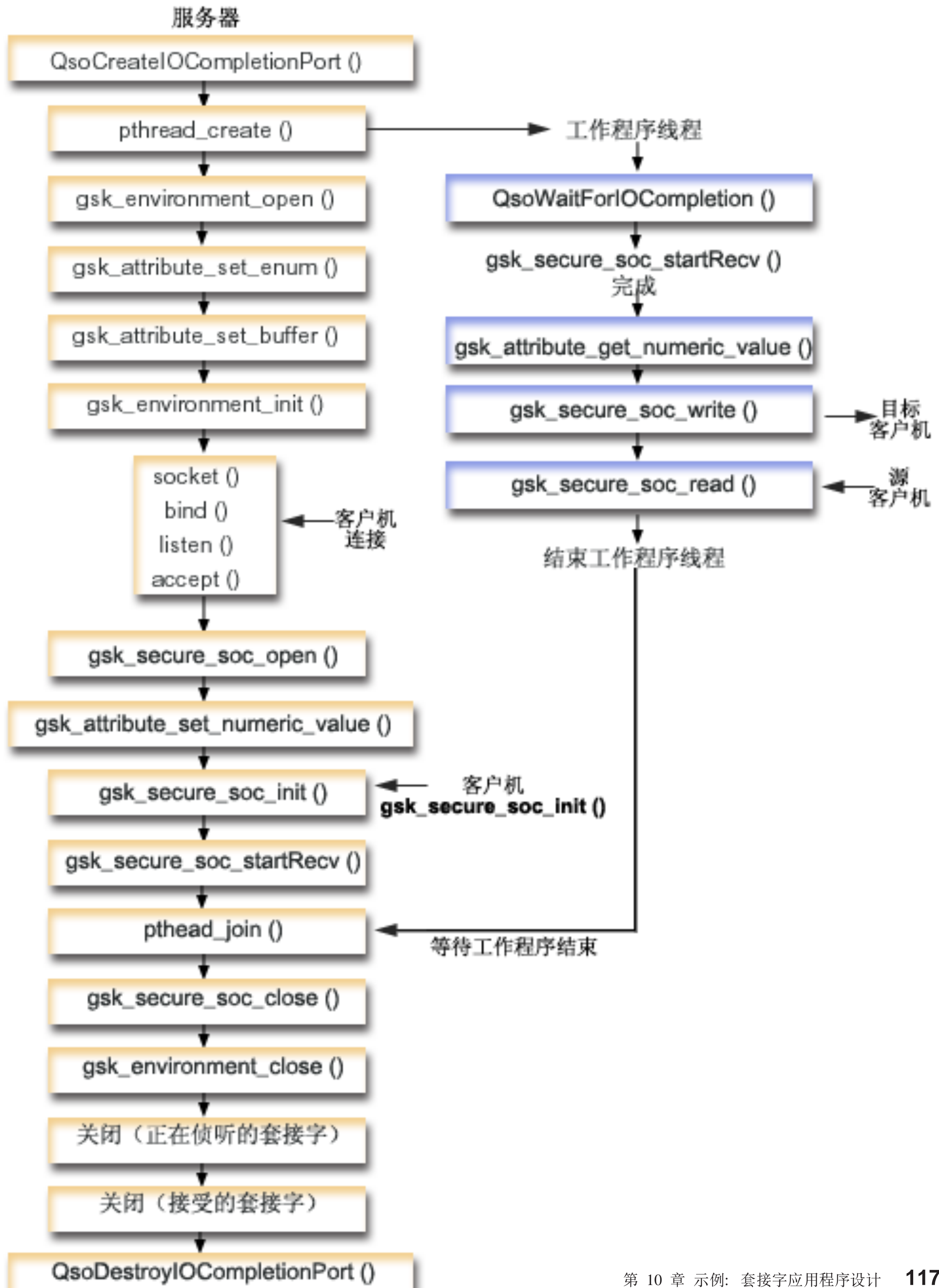
- 示例：使用 SSL API 建立安全服务器
- 示例：使用 SSL API 建立安全客户机

示例：使用异步数据接收的 GSKit 安全服务器

以下代码示例可用来使用 Global Secure Toolkit（GSKit）API 建立安全服务器。服务器打开套接字，准备安全环境，接受并处理连接请求，与客户机交换数据并结束会话。客户机还会打开套接字，设置安全环境，调用服务器并请求安全连接，与服务器交换数据并关闭会话。下图和描述显示服务器/客户机事件流。

注： 下列示例程序使用 AF_INET 地址系列，但它们也可修改为使用 AF_INET6 地址系列。

| 套接字事件流：使用异步数据接收的安全服务器
|



|

要查看此图的客户机部分，参见安全 GSKit 客户机图。

以下套接字调用序列提供图形的描述。它还描述服务器与客户机示例之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。此流描述以下样本应用程序中的套接字调用。.

1. **QsoCreateIOCompletionPort()** 函数创建 I/O 完成端口。
2. **pthread_create** 函数创建工作程序线程以接收数据并将其回传至客户机。工作程序线程将等待客户机请求到达刚创建的 I/O 完成端口。
3. 对 **gsk_environment_open()** 的调用，用来获取 SSL 环境的句柄。
4. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置 SSL 环境的属性。至少是对 **gsk_attribute_set_buffer()** 的调用，用来设置 GSK_OS400_APPLICATION_ID 值或设置 GSK_KEYRING_FILE 值。仅应设置其中之一。最好使用 GSK_OS400_APPLICATION_ID 值。还应确保使用 **gsk_attribute_set_enum()** 设置应用程序（客户机或服务器）的类型 GSK_SESSION_TYPE。
5. 对 **gsk_environment_init()** 的调用，用来初始化此环境以便进行 SSL 处理和建立将使用此环境运行的所有 SSL 会话的 SSL 安全性信息。
6. **socket** 函数创建套接字描述符。于是服务器发出一组标准套接字调用：**bind()**、**listen()** 和 **accept()** 以允许服务器接受入局连接请求。
7. **gsk_secure_soc_open()** 函数获取安全会话的存储器，设置属性的缺省值，并返回必须保存以便在与安全会话相关的函数调用上使用的句柄。
8. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置安全会话的属性。至少是对 **gsk_attribute_set_numeric_value()** 的调用，用来将特定套接字与此安全会话相关联。
9. 对 **gsk_secure_soc_init()** 的调用，用来启动加密参数的 SSL 握手协商。

   注： 通常，服务器程序必须提供证书，SSL 握手才会成功。服务器还必须具有对与服务器证书相关联的专用密钥以及存储该证书的密钥数据库文件的访问权。在某些情况下，客户机还必须在 SSL 握手处理期间提供证书。如果客户机正连接至的服务器已启用客户机认证，就需要这样做。**gsk_attribute_set_buffer**（GSK_OS400_APPLICATION_ID）或 **gsk_attribute_set_buffer**（GSK_KEYRING_FILE）API 调用标识（尽管以不同方式）在获取握手期间从中使用证书和专用密钥的密钥数据库文件。
10. **gsk_secure_soc_startRecv()** 函数对安全会话启动异步接收操作。
11. **pthread_join** 使服务器与工作程序同步。此函数等待线程终止，拆离该线程，然后将线程退出状态返回给服务器。
12. **gsk_secure_soc_close()** 函数结束安全会话。
13. **gsk_environment_close()** 函数关闭 SSL 环境。
14. **close()** 函数结束侦听套接字。
15. **close()** 结束接受的（客户机连接）套接字。
16. **QsoDestroyIOCompletionPort()** 函数破坏完成端口。

套接字事件流：使用 GSKit API 的工作程序线程

1. 服务器应用程序创建工作程序线程后，它会等待服务器向其发送入局客户机请求以便使用 **gsk_secure_soc_startRecv()** 调用处理客户机数据。**QsoWaitForIOCompletionPort()** 函数将等待提供的 IO 完成端口，这是由服务器指定的。

2. 一旦接收到客户机请求，**gsk_attribute_get_numeric_value()** 函数会获取与安全会话相关联的套接字描述符。
3. **gsk_secure_soc_write()** 函数将该消息发送至使用安全会话的客户机。

有关使用代码示例的信息，参见代码不保证声明。

```
/* 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);

```

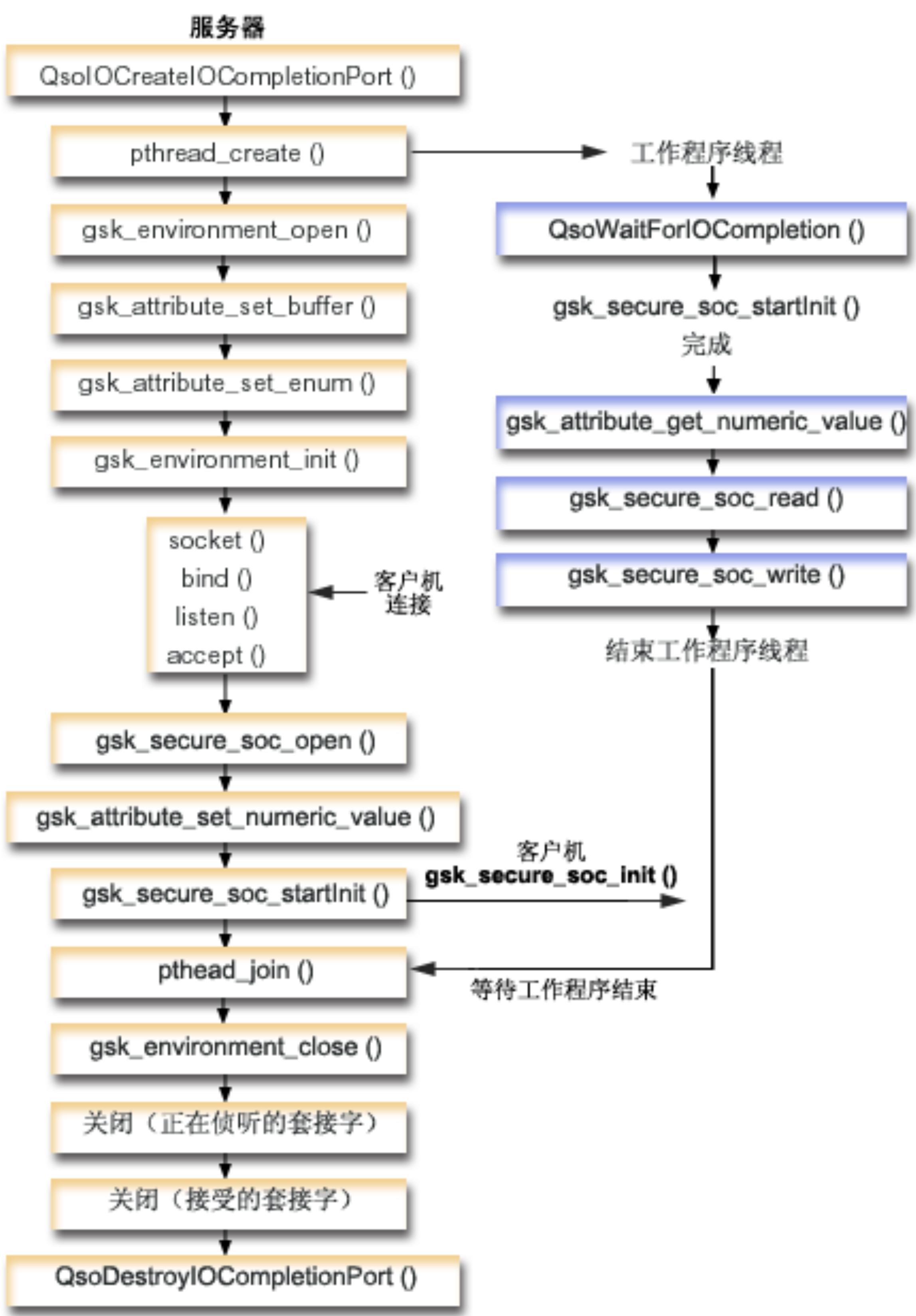
```
  /* disable the SSL environment */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* 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 */
```

示例: 使用异步握手的 GSKit 安全服务器

OS/400 套接字是 V5R2 的新增内容，它们引入了 **gsk_secure_soc_startInit()** API。此 API 允许您创建可异步处理请求的安全服务器应用程序。以下代码样本提供如何使用此 API 的示例。它类似于使用异步数据接收的 GSKit 安全服务器示例，但它使用这一新的 API 启动安全会话。

下图显示用于在安全服务器上协商异步握手的函数调用:



要查看此图的客户机部分，参见 GSKit 客户机。

套接字事件流：使用异步握手的 GSKit 安全服务器

此流描述以下样本应用程序中的套接字调用。

1. **QsoCreateIOCompletionPort()** 函数创建 I/O 完成端口。
2. **pthread_create** 函数创建工作程序线程以处理所有客户机请求。工作程序线程将等待客户机请求到达刚创建的 I/O 完成端口。
3. 对 **gsk_environment_open()** 的调用，用来获取 SSL 环境的句柄。
4. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置 SSL 环境的属性。至少是对 **gsk_attribute_set_buffer()** 的调用，用来设置 GSK_OS400_APPLICATION_ID 值或设置 GSK_KEYRING_FILE 值。仅应设置其中之一。最好使用 GSK_OS400_APPLICATION_ID 值。还应确保使用 **gsk_attribute_set_enum()** 设置应用程序（客户机或服务器）的类型 GSK_SESSION_TYPE。
5. 对 **gsk_environment_init()** 的调用，用来初始化此环境以便进行 SSL 处理和建立将使用此环境运行的所有 SSL 会话的 SSL 安全性信息。
6. **socket** 函数创建套接字描述符。于是服务器发出一组标准套接字调用：**bind()**、**listen()** 和 **accept()** 以允许服务器接受入局连接请求。
7. **gsk_secure_soc_open()** 函数获取安全会话的存储器，设置属性的缺省值，并返回必须保存以便在与安全会话相关的函数调用上使用的句柄。
8. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置安全会话的属性。至少是对 **gsk_attribute_set_numeric_value()** 的调用，用来将特定套接字与此安全会话相关联。
9. **gsk_secure_soc_startInit()** 函数使用为 SSL 环境和安全会话设置的属性启动安全会话的异步协商。控制权将返回给程序。握手处理完成时，会向完成端口公布结果。线程可继续进行其它处理；但是，为简单起见，我们选择在此处等待工作程序完成。

   注： 通常，服务器程序必须提供证书，SSL 握手才会成功。服务器还必须具有对与服务器证书相关联的专用密钥以及存储该证书的密钥数据库文件的访问权。在某些情况下，客户机还必须在 SSL 握手处理期间提供证书。如果客户机正连接至的服务器已启用客户机认证，就需要这样做。**gsk_attribute_set_buffer**（GSK_OS400_APPLICATION_ID）或 **gsk_attribute_set_buffer**（GSK_KEYRING_FILE）API 调用标识（尽管以不同方式）在获取握手期间从中使用证书和专用密钥的密钥数据库文件。
10. **pthread_join** 使服务器与工作程序同步。此函数等待线程终止，拆离该线程，然后将线程退出状态返回给服务器。
11. **gsk_secure_soc_close()** 函数结束安全会话。
12. **gsk_environment_close()** 函数关闭 SSL 环境。
13. **close()** 函数结束侦听套接字。
14. **close()** 结束接受的（客户机连接）套接字。
15. **QsoDestroyIOCompletionPort()** 函数破坏完成端口。

套接字事件流：处理安全异步请求的工作程序线程

1. 服务器应用程序创建工作程序线程后，它会等待服务器向其发送要处理的入局客户机请求。**QsoWaitForIOCompletionPort()** 函数将等待提供的 IO 完成端口，这是由服务器指定的。此调用一直等待，直到 **gsk_secure_soc_startInit()** 完成。

2. 一旦接收到客户机请求，**gsk_attribute_get_numeric_value()** 函数会获取与安全会话相关联的套接字描述符。
3. **gsk_secure_soc_read()** 函数接收来自使用安全会话的客户机的消息。
4. **gsk_secure_soc_write()** 函数将该消息发送至使用安全会话的客户机。

有关使用代码示例的信息，参见代码不保证声明。

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

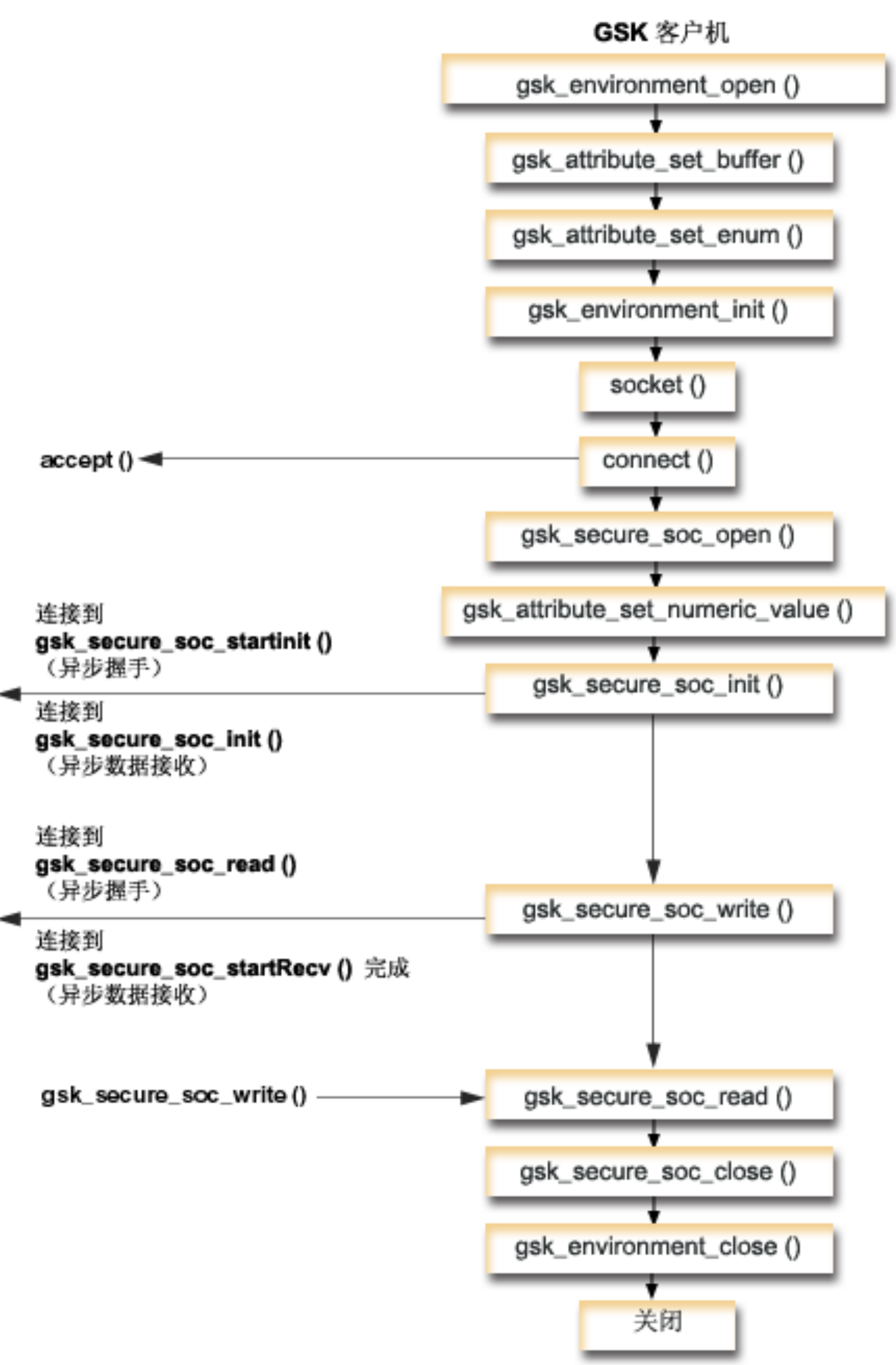
```
/* Descriptive Name: Process client connection.                    */
/*                                                                 */
/* Note:  To make the sample more straight forward the main routine */
/*        handles all of the clean up although this function could  */
/*        be made 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 */
```

示例：使用 Global Secure ToolKit（GSKit）API 建立安全客户机

以下代码样本提供使用 GSKit API 的客户机的示例。有关使用代码示例的信息，参见代码不保证声明。

下图显示使用 GSKit API 的安全客户机上的函数调用:


GSK 客户机
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 ()
连接到
gsk_secure_soc_startinit ()
（异步握手）
gsk_secure_soc_init ()
连接到
gsk_secure_soc_init ()
（异步数据接收）
连接到
gsk_secure_soc_read ()
（异步握手）
gsk_secure_soc_write ()
连接到
gsk_secure_soc_startRecv () 完成
（异步数据接收）
gsk_secure_soc_write ()
gsk_secure_soc_read ()
gsk_secure_soc_close ()
gsk_environment_close ()
关闭

套接字事件流：GSKit 客户机

此流描述以下样本应用程序中的套接字调用。将此客户机示例与 GSKit 服务器示例以及示例：使用异步握手的 GSKit 安全服务器配合使用。

1. **gsk_environment_open()** 函数获取 SSL 环境的句柄。
2. 对 gsk_attribute_set_xxxxx() 的一次或多次调用，用来设置 SSL 环境的属性。至少是对 **gsk_attribute_set_buffer()** 的调用，用来设置 GSK_OS400_APPLICATION_ID 值或设置 GSK_KEYRING_FILE 值。仅应设置其中之一。最好使用 GSK_OS400_APPLICATION_ID 值。还应确保使用 **gsk_attribute_set_enum()** 设置应用程序（客户机或服务器）的类型 GSK_SESSION_TYPE。
3. 对 **gsk_environment_init()** 的调用，用来初始化此环境以便进行 SSL 处理和建立将使用此环境运行的所有 SSL 会话的 SSL 安全性信息。
4. **socket** 函数创建套接字描述符。于是客户机发出 **connect()** 以连接至服务器应用程序。
5. **gsk_secure_soc_open()** 函数获取安全会话的存储器，设置属性的缺省值，并返回必须保存以便在与安全会话相关的函数调用上使用的句柄。
6. **gsk_attribute_set_numeric_value()** 函数将特定套接字与此安全会话相关联。
7. **gsk_secure_soc_init()** 函数使用为 SSL 环境和安全会话设置的属性启动安全会话的异步协商。
8. **gsk_secure_soc_write()** 函数将安全会话上的数据写至工作程序线程。

   注： 对于 GSKit 服务器示例，此函数将数据写至 **gsk_secure_soc_startRecv()** 函数完成的工作程序线程。在异步示例中，它将写至完成的 **gsk_secure_soc_startInit()** 。
9. **gsk_secure_soc_read()** 函数接受来自使用安全会话的工作程序线程的消息。
10. **gsk_secure_soc_close()** 函数结束安全会话。
11. **gsk_environment_close()** 函数关闭 SSL 环境。
12. **close()** 函数结束连接。

```
/* 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     */
```

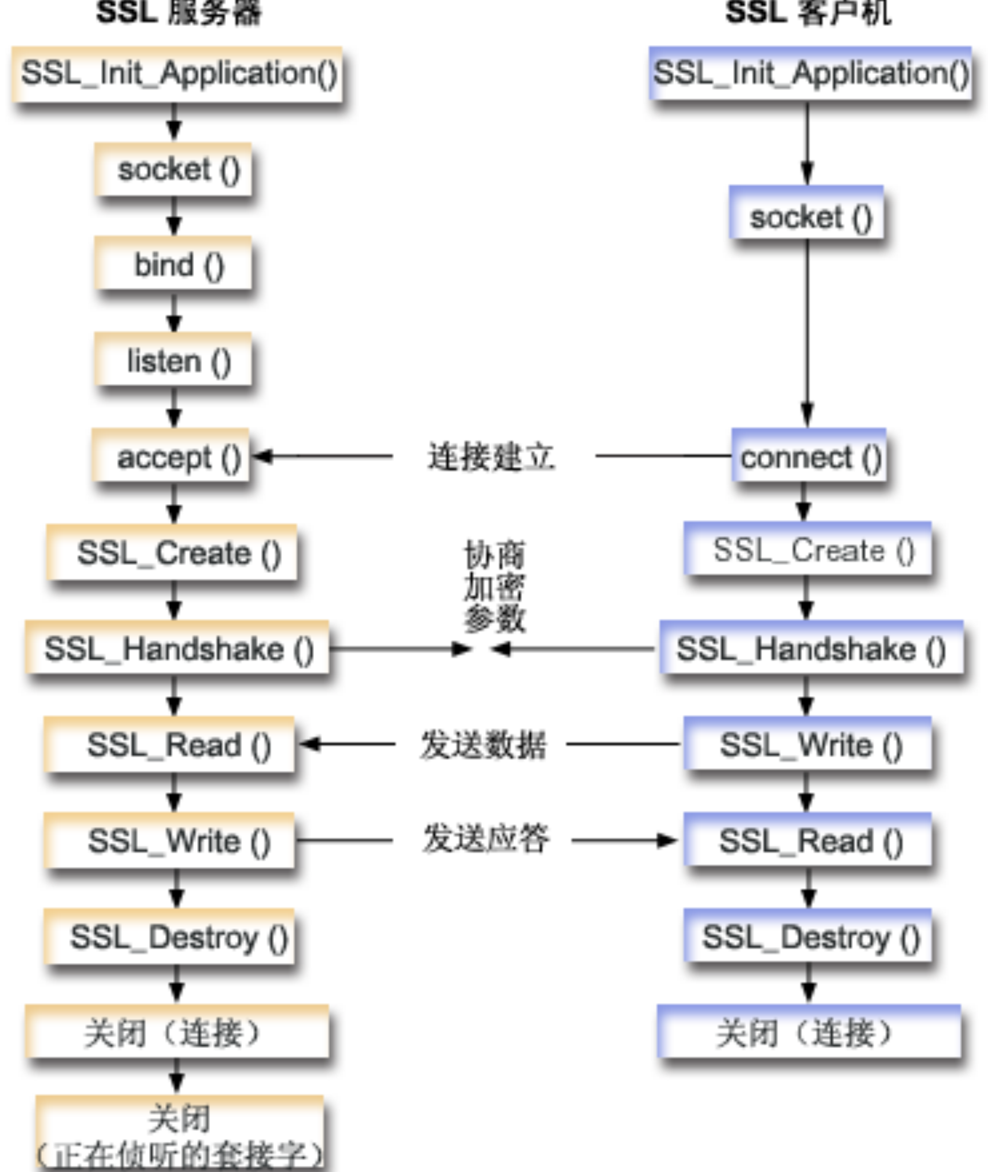
```
    /* of ciphers is enabled.  For this example we will just use one */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                  GSK_V3_CIPHER_SPECS,
                                  "05", /* SSL_RSA_WITH_RC4_128_SHA */
                                  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;
}
```

示例：使用 SSL_ API 建立安全服务器

除使用 GSKit API 创建安全应用程序之外，还可使用 SSL_ API。这些 API 是 iSeries 操作系统本地的。就 GSKit API 而言，服务器必须提供有效证书才能安全地交换数据。下图显示用于创建安全服务器的套接字和 SSL_ API。如果正在编写跨 IBM @server 平台的安全应用程序，使用 GSKit API。



套接字事件流：使用 SSL_ API 的安全服务器

以下描述显示允许 SSL 服务器执行并与 SSL 客户机的通信的 API 之间的关系。

1. 调用 **SSL_Init()** 或 **SSL_Init_Application()**，以初始化 SSL 处理的作业环境和建立将在当前作业中运行的所有 SSL 会话的 SSL 安全性信息。仅应使用其中一个 API。最好使用 **SSL_Init_Application()** API。

   注： 以下示例程序使用 **SSL_Init_Application** API。
2. 服务器调用 **socket()** 以获取套接字描述符。
3. 服务器调用 **bind()**、**listen()** 和 **accept()** 以激活服务器程序的连接。
4. 服务器调用 **SSL_Create()** 以便对连接的套接字启用 SSL 支持。
5. 服务器调用 **SSL_Handshake()** 以启动加密参数的 SSL 握手协商。

6. 服务器调用 **SSL_Write()** 和 **SSL_Read()** 以发送和接收数据。
7. 服务器调用 **SSL_Destroy()** 以便对套接字禁用 SSL 支持。
8. 服务器调用 **close()** 以破坏连接的套接字。

套接字事件流：使用 SSL_ API 的安全客户机

1. 调用 **SSL_Init()** 或 **SSL_Init_Application()**，以初始化 SSL 处理的作业环境和建立将在当前作业中运行的所有 SSL 会话的 SSL 安全性信息。仅应使用其中一个 API。最好使用 **SSL_Init_Application** API。

   注：以下示例程序使用 **SSL_Init_Application** API。
2. 客户机调用 **socket()** 以获取套接字描述符。
3. 客户机调用 **connect()** 以激活客户机程序的连接。
4. 客户机调用 **SSL_Create()** 以便对连接的套接字启用 SSL 支持。
5. 客户机调用 **SSL_Handshake()** 以启动加密参数的 SSL 握手协商。
6. 客户机调用 **SSL_Read()** 和 **SSL_Write()** 以接收和发送数据。
7. 客户机调用 **SSL_Destroy()** 以便对套接字禁用 SSL 支持。
8. 客户机调用 **close()** 以破坏连接的套接字。

注：样本使用 AF_INET 地址系列；但是它可以修改为使用 AF_INET6 地址系列。
有关使用代码示例的信息，参见代码不保证声明。

```
/* 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;
}
```

示例: 使用 SSL_ API 建立安全客户机

除 GSKit API 之外，OS/400 套接字还支持传统 SSL_ API。这些 API 在 iSeries 本地服务器与客户机应用程序之间建立安全连接。有关描述此程序及其相应服务器应用程序的套接字事件流的图形，参见示例：使用 SSL_ API 建立安全服务器。以下示例允许使用 SSL_ API 的客户机应用程序与使用 SSL_API 的服务器应用程序通信:

有关使用代码示例的信息，参见代码不保证声明。

```
/* 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;
}
```

示例：对线程安全网络例程使用 gethostbyaddr_r()

以下是使用 **gethostbyaddr_r()** 的程序的示例。名称以“_r”结尾的所有其它例程具有相似语义，而且也是线程安全的。此示例程序采用点分十进制表示的 IP 地址，并打印主机名。

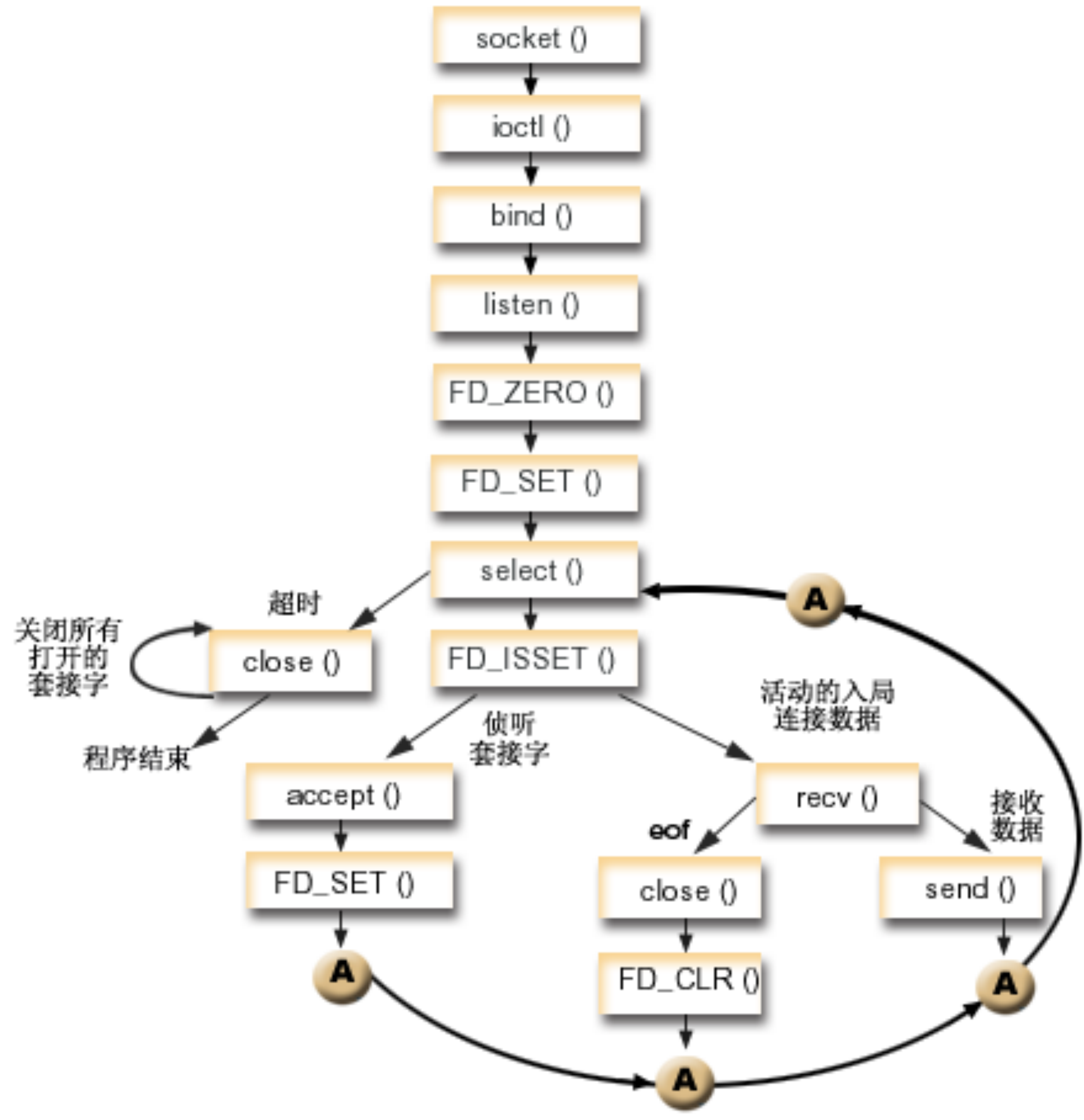
有关使用代码示例的信息，参见代码不保证声明。

```
/********************************************************/
/* 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);
     }
```

示例：非阻塞 I/O 和 select()

以下样本程序使用非阻塞 I/O 和 select() API。有关包含可与此示例配合使用的常见客户机作业的代码的示例，参见示例：一般客户机。



套接字事件流：使用非阻塞 I/O 和 select() 的服务器

示例中使用了下列调用:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_STREAM）的 INET（网际协议）地址系列。
2. **ioctl()** 函数允许在必需的等待时间到期之前在重新启动服务器时重用本地地址。在此示例中，它将套接字设置为非阻塞的。入局连接的所有套接字也是非阻塞的，原因是它们将从侦听套接字继承该状态。
3. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。
4. **listen()** 允许服务器接受入局客户机连接。
5. 服务器使用 **accept()** 函数接受入局连接请求。**accept()** 调用将无限期阻塞，等待入局连接的到来。
6. **select()** 函数允许进程等待事件发生并在事件发生时唤醒进程。在此示例中，**select ()** 函数返回一个数字，表示已准备好进行处理的套接字描述符。

| **0** 指示进程将超时。在此示例中超时设置为 30 秒。

| **–1** 指示进程已失败。

| **1** 指示只准备处理一个描述符。在此示例中，返回 1 时 FD_ISSET 和后续套接字调用只完成一次。

| **n** 指示有多个描述符等待处理。在此示例中，返回 n 时 FD_ISSET 和后续代码循环执行并按服务器
| 接收请求的次序来完成这些请求。

7. 返回 EWOULDBLOCK 时，将完成 **accept()** 和 **recv()** 函数。
8. **send()** 函数将数据回传至客户机。
9. **close()** 函数关闭所有打开的套接字描述符。

有关使用代码示例的信息，参见代码不保证声明。

```
#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;
            }
```

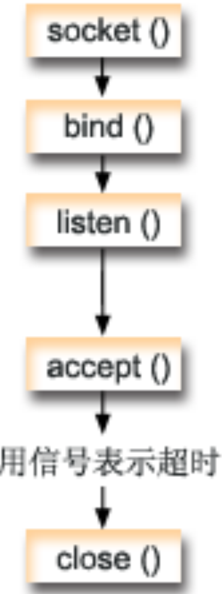
```
            /*************************************************/
            /* Add the new incoming connection to the     */
            /* master read set                            */
            /*************************************************/
            printf("  New incoming connection - %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);
   }
}
```

示例: 将信号与阻塞套接字 API 配合使用

信号允许在进程或应用程序被阻塞时让您得到通知。信号提供阻塞进程的时间限制。在此示例中，在 **accept()** 调用上会在 5 秒之后发出信号。此调用通常无限期阻塞，但因为设置了警报，所以它将仅阻塞 5 秒。因为阻塞的程序可能会对应用程序或服务器的性能有影响，所以可使用它们来消除此影响。以下示例显示如何将信号与阻塞套接字 API 配合使用。

注: 在线程式服务器模型中使用的异步 I/O 比经常使用的模型更为可取。有关使用异步 I/O 的优点的更多信息，参见异步 I/O。有关使用异步 I/O API 的样本程序，参见示例: 使用异步 I/O。



套接字事件流：将信号与阻塞套接字配合使用

以下函数调用序列显示当套接字处于不活动状态时如何使用信号警告应用程序:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 UDP 传输（SOCK_DGRAM）的 INET（网际协议）地址系列。
2. 在创建套接字描述符之后，**bind()** 函数获取套接字的唯一名称。在此示例中，未指定端口号，原因是客户机应用程序不会连接至此套接字。可在使用阻塞 API 的其它服务器程序中使用此代码片断，如 **accept()**。
3. **listen()** 函数指示接受客户机连接请求的意愿。发出 **listen()** 函数后，将警报设置为在 5 秒之后发出。当 **accept()** 调用停止时，此警报或信号将提醒您。
4. **accept()** 函数接受客户机连接请求。此调用通常无限期阻塞，但因为设置了警报，所以它将仅阻塞 5 秒。当警报发出时，接受调用将会完成并带有值 -1，错误号值为 EINTR。
5. **close()** 函数结束所有打开的套接字描述符。

有关使用代码示例的信息，参见代码不保证声明。

```
/******************************************************************/
/* 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);
}
```

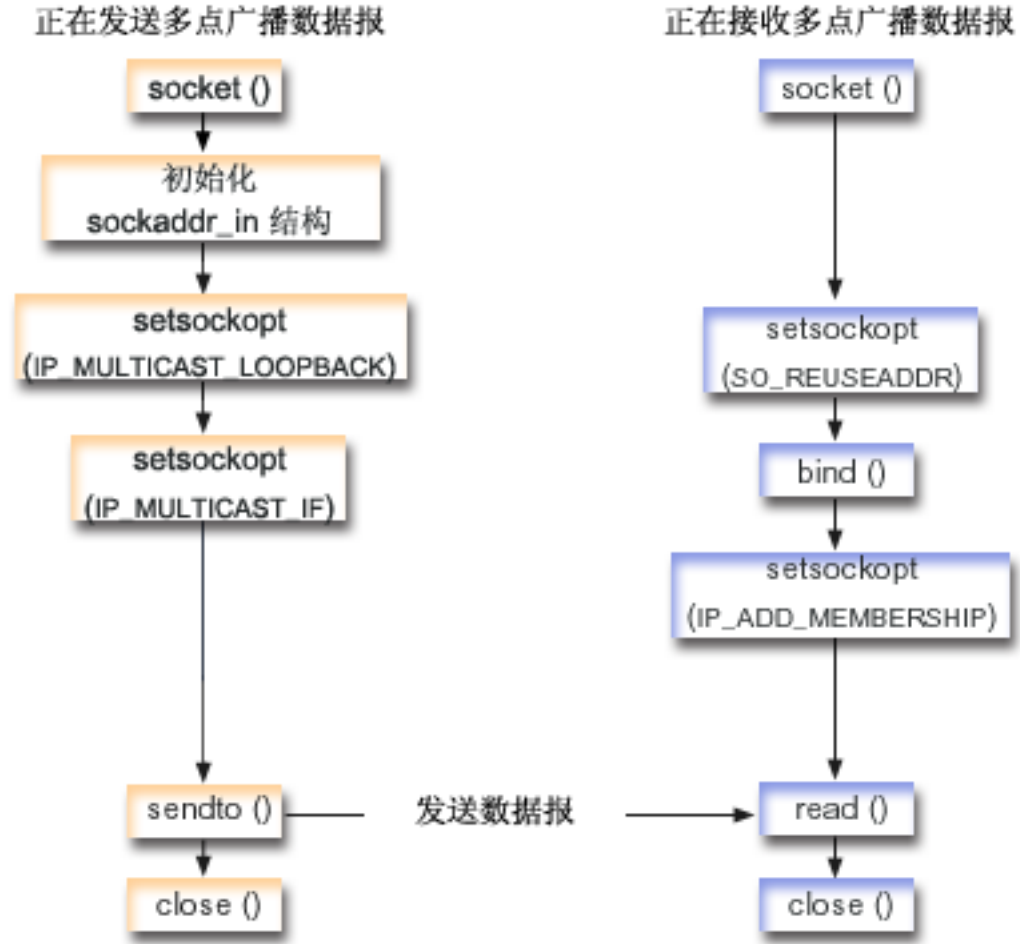
示例: 使用多点广播

IP 多点广播允许应用程序发送网络中的一组主机可以接收到的单个 IP 数据报。该组中的主机可能驻留在单个子网中，也可能驻留在连接可使用多点广播的路由器的不同子网中。主机可以随时加入或离开组。对主机组中的成员位置或数目没有任何限制。范围在 224.0.0.1 到 239.255.255.255 之间的 D 类因特网地址标识主机组。

应用程序可使用 **socket()** API 和无连接的 SOCK_DGRAM 类型套接字发送或接收多点广播数据报。多点广播是一种一对多的传送方法。不能使用类型为 SOCK_STREAM 的面向连接的套接字进行多点广播。在创建类型为 SOCK_DGRAM 的套接字后，应用程序可使用 **setsockopt()** 函数来控制与该套接字相关联的多点广播特征。setsockopt() 函数接受下列 IPPROTO_IP 级别标志:

- IP_ADD_MEMBERSHIP: 加入指定的多点广播组。
- IP_DROP_MEMBERSHIP: 离开指定的多点广播组。
- IP_MULTICAST_IF: 设置通过其发送出局多点广播数据报的接口。
- IP_MULTICAST_TTL: 在 IP 头中设置出局多点广播数据报的“有效时间”（TTL）。
- IP_MULTICAST_LOOP: 指定当发送主机是多点广播组的成员时，是否将出局多点广播数据报的副本传送至发送主机。

| **注**: OS/400 套接字支持对 AF_INET 地址系列使用 IP 多点广播。



套接字事件流：发送多点广播数据报

以下套接字调用序列提供图形的描述。它还描述发送和接收多点广播数据报的两个应用程序之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。发送多点广播数据报使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_DGRAM）的 INET（网际协议）地址系列。此套接字会将数据报发送至另一应用程序。
2. sockaddr_in 结构指定目标 IP 地址和端口号。在此示例中，地址为 225.1.1.1，而端口号为 5555。
3. **setsockopt()** 函数设置 IP_MULTICAST_LOOP 套接字选项，所以发送系统不会接收它传送的多点广播数据报的副本。
4. **setsockopt()** 函数使用 IP_MULTICAST_IF 套接字选项，它定义通过其发送多点广播数据报的本地接口。
5. **sendto()** 函数将多点广播数据报发送至指定组 IP 地址。
6. **close()** 函数关闭所有打开的套接字描述符。

套接字事件流：接收多点广播数据报

接收多点广播数据报使用以下函数调用序列:

1. **socket()** 函数返回表示端点的套接字描述符。该语句还标识将对此套接字使用带有 TCP 传输（SOCK_DGRAM）的 INET（网际协议）地址系列。此套接字会将数据报发送至另一应用程序。

2. **setsockopt()** 函数设置 SO_REUSEADDR 套接字选项，以允许多个应用程序接收目标为同一本地端口号的数据报。
3. **bind()** 函数指定本地端口号。在此示例中，IP 地址被指定为 INADDR_ANY 以接收发送至多点广播组的数据报。
4. **setsockopt()** 函数使用 IP_ADD_MEMBERSHIP 套接字选项，它将加入接收数据报的多点广播组。在加入组时，指定 D 类组地址和本地接口的 IP 地址。系统必须对接收多点广播数据报的每个本地接口调用 IP_ADD_MEMBERSHIP 套接字选项。在此示例中，将在本地 9.5.1.1 接口上加入多点广播组（225.1.1.1）。

   注: 必须对通过其接收多点广播数据报的每个本地接口调用 IP_ADD_MEMBERSHIP 选项。
5. **read()** 函数读取正在发送的多点广播数据报。
6. **close()** 函数关闭所有打开的套接字描述符。

示例: 发送多点广播数据报

以下示例允许套接字执行下面列出的步骤和发送多点广播数据报。如果想要复查此程序的套接字事件流的描述，参见示例: 使用多点广播。

有关使用代码示例的信息，参见代码不保证声明。

```
#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");
  }
}
```

示例: 接收多点广播数据报

以下示例允许套接字执行下面列出的步骤和接收多点广播数据报: 如果想要复查此程序的套接字事件流的描述, 参见示例: 使用多点广播。

有关使用代码示例的信息, 参见代码不保证声明。

```
#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);
  }

}
```

示例：更新和查询 DNS

以下示例显示如何查询和更新“域名系统”（DNS）记录。

有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************************************/
/* 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.",
```

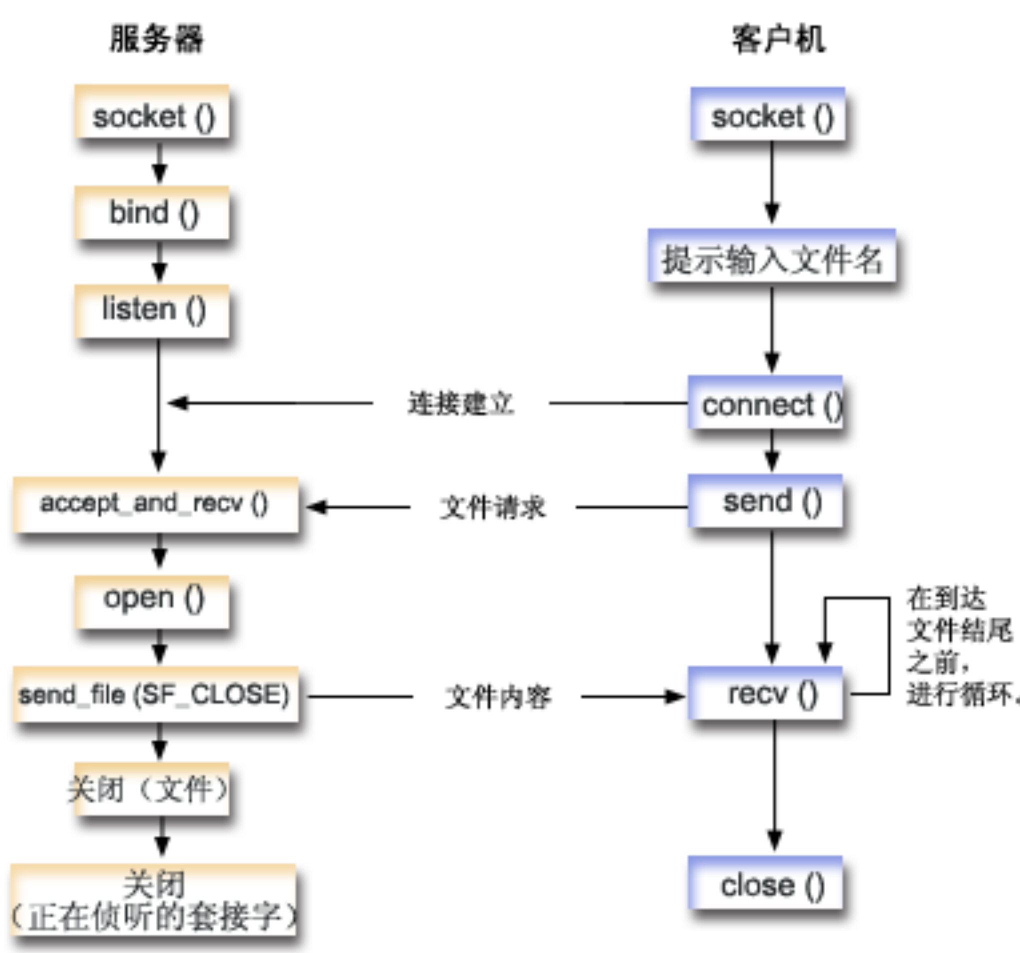
```
      ns_c_in,
      ns_t_a,                  /* IPv4 address  */
      10,
      (unsigned char *)"10.10.10.10",
      11,
      ns_uop_add,               /* to be added  */
      NULL,
      NULL,
      0
   },
   {
      {&update_records[2],NULL},
      {&update_records[2],NULL},
      ns_s_ud,                 /* 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;
 }
```

示例：使用 send_file() 和 accept_and_recv() API 传输文件数据

下列示例通过使用 **send_file()** 和 **accept_and_recv()** API 允许服务器与客户机通信。



套接字事件流：服务器发送文件内容

以下套接字调用序列提供图形的描述。它还描述发送和接收文件的两个应用程序之间的关系。每一组流包含指向有关特定 API 的使用注意事项的链接。如果需要有关使用特定 API 的更多详细信息，可使用这些链接。示例：使用 accept_and_recv() 和 send_file() API 发送文件内容使用以下函数调用序列：

1. 服务器调用 **socket()**、**bind()** 和 **listen()** 以创建侦听套接字。
2. 服务器初始化本地和远程地址结构。
3. 服务器调用 **accept_and_recv()** 以等待入局连接和等待通过此连接到达的第一个数据缓冲区。此调用返回接收到的字节数以及与此连接相关联的本地和远程地址。此调用是 **accept()**、**getsockname()** 和 **recv()** API 的组合。
4. 服务器调用 **open()** 以打开文件，其名称是在 **accept_and_recv()** 上作为数据从客户机应用程序获取的。
5. 使用 **memset()** 函数将 sf_parms 结构的所有字段设置为初始值零。服务器将文件描述符字段设置为 **open()** 返回的值。于是服务器将文件字节数字段设置为 -1，以指示服务器应发送整个文件。由于系统在发送整个文件，所以不需要指定文件偏移量字段。
6. 服务器调用 **send_file()** 以传送文件内容。**send_file()** 会一直执行直到发送整个文件或发生中断。**send_file()** 非常有效，原因是应用程序在文件完成之前不必进入 **read()** 和 **send()** 循环。
7. 服务器在 **send_file()** API 上指定 `SF_CLOSE` 标志。`SF_CLOSE` 标志通知 **send_file()** API 它应在成功发送文件的最后一个字节和结尾缓冲区（如果指定的话）后自动关闭套接字连接。如果指定了 `SF_CLOSE` 标志的话，应用程序就不必调用 **close()**。

套接字事件流：文件的客户机请求

示例：文件的客户机请求使用以下函数调用序列：

1. 客户机程序采用零到两个参数。

   第一个参数（如果指定的话）是点分十进制 IP 地址或服务器应用程序所在的主机名。

   第二个参数（如果指定的话）是客户机尝试从服务器获取的文件的名称。服务器应用程序将指定文件的内容发送至客户机。如果用户不指定任何参数，则客户机对服务器的 IP 地址使用 INADDR_ANY。如果用户不指定第二个参数，则程序将提示用户输入文件名。
2. 客户机调用 **socket()** 以创建套接字描述符。
3. 客户机调用 **connect()** 以建立与服务器的连接。步骤一获取服务器的 IP 地址。
4. 客户机调用 **send()** 以通知服务器它想要获取的文件名。步骤一获取文件的名称。
5. 客户机进入调用 **recv()** 的“do”循环，直至到达文件结尾。**recv()** 上的返回码零表示服务器已关闭该连接。
6. 客户机调用 **close()** 以关闭套接字。

示例：使用 accept_and_recv() 和 send_file() API 发送文件内容

以下示例允许服务器执行下面列示的步骤以通过使用 **send_file()** 和 **accept_and_recv()** API 与客户机通信。

有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************/
/* 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);
}
```

示例：客户机对文件的请求

以下示例允许客户机请求服务器中的文件并等待服务器发送回该文件的内容。

有关使用代码示例的信息，参见代码不保证声明。

```
/**************************************************/
/* 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);
}
```

第 11 章 Xsockets 工具

Xsocket 工具是随 iSeries 一起交付的多种工具之一。所有工具都存储在 QUSRTOOL 库中。Xsockets 允许程序员以交互方式使用套接字 API。Xsocket 工具允许您执行下列任务:

- 了解套接字 API
- 以交互方式重新创建特定方案以帮助调试

注: Xsockets 工具是以“按现状”格式交付的。

使用 Xsockets 的先决条件

- 已安装 ILE C/400 语言。
- 已安装许可程序 5722 — SS1 “Openness Include”。
- 已安装许可程序 5722 — DG1 “IBM HTTP Server”。

  注: 如果计划在 Web 浏览器中使用 Xsockets，则需要此项。
- 已安装许可程序 5722 — JV1 “Developer Kit for Java”。

  注: 如果计划在 Web 浏览器中使用 Xsockets，则需要此项。

使用下列主题来安装和使用 Xsocket 工具:

配置 Xsockets
本主题说明如何创建 Xsockets 工具，该工具将帮助您设计和编译套接字程序。

使用 Xsockets
本主题说明如何使用 Xsockets 工具。

定制 Xsockets
本主题说明如何定制 Xsockets 工具。

配置 Xsockets

可创建两种版本的工具。第一种版本是本地 iSeries 客户机。本地版本完全是由第一组指令创建的。第二种版本将 Web 浏览器用作客户机。如果想要使用 Web 浏览器客户机，则必须先完成本地版本的本地安装指令。

要创建 Xsockets 工具，完成下列步骤:

1. 要将该工具解包，在命令行上输入
   ```
   CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)。
   ```
   注: 左单引号和右单引号之间必须是 10 个字符。
2. 要将 QUSRTOOL 库添加至库列表，在命令行上输入
   ```
   ADDLIBLE QUSRTOOL。
   ```
3. 通过在命令行上输入
   ```
   CRTLIB <library-name>
   ```
   来创建要在其中创建 Xsocket 程序文件的库。`<library-name>` 是想要在其中创建 Xsocket 工具对象的库。例如，

```
CRTLIB MYXSOCKET
```

将是有效的库名称。

注: 不要将 Xsocket 工具对象添加至 QUSRTOOL 库。这可能会对在该目录中使用其它工具有干扰。

4. 要将此库添加至库列表，在命令行上输入

```
ADDLIBLE <library-name>。
```

`<library-name>` 是在步骤三中创建的库。例如，我们将 MYXSOCKET 用作库名，于是输入:

```
ADDLIBLE MYXSOCKET
```

5. 通过在命令行上输入:

```
CRTCLPGM <library-name>/TSOCRT QUSRTOOL/QATTCL
```

来创建将自动安装 Xsockets 工具的安装程序 TSOCRT。`<library-name>` 是在步骤三中创建的库。例如，我们将 MYXSOCKET 用作库名，于是输入:

```
CRTCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
```

6. 要调用安装程序，在命令行上输入:

```
CALL TSOCRT library-name。
```

在 library-name 位置上使用在步骤 3 中创建的库。例如。要在 MYXSOCKET 库中创建该工具，输入:

```
CALL TSOCRT MYXSOCKET
```

注: 这可能要花几分钟才能完成。

如果在调用 TSOCRT 创建套接字工具时没有作业控制（*JOBCTL）特权，在尝试将描述符传递至并非您正在运行的作业时，**givedescriptor()** 套接字函数将返回错误。

TSOCRT 将创建 CL 程序、ILE C/400 程序（创建了 2 个模块）、2 个 ILE C/400 服务程序（创建了 2 个模块）和 3 个显示文件。每当想要使用该工具时，应将该库添加至库列表。由该工具创建的所有对象都具有以 TSO 为前缀的名称。

下一步要执行的操作:

使用本地 Xsockets
如果想要在本地使用 Xsocket，可转至此步骤。本主题包括使用本地 Xsockets 工具的基础。

注: 本地版本不支持 GSKit 安全套接字 API。如果想要编写使用这些 API 的套接字程序，应使用基于浏览器的版本的工具。

将 Xsockets 配置为使用 Web 浏览器

注: 此步骤是可选的。

本地 Xsocket 安装创建的内容

下表列示安装程序创建的对象。创建的所有对象将驻留在指定库中。

表 20. Xsocket 安装期间创建的对象

对象名	成员名	源文件名	对象类型	扩展	描述

表 20. Xsocket 安装期间创建的对象（续）

TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	用于在 JSP 与 TSOSTSOC 之间进行交互的模块
TSODLT	TSODLT	QATTCL	*PGM	CLP	用来删除工具对象和／或源文件成员的 CL 程序。
TSOXSOCK	N/A	N/A	*PGM	C	用于 SOCKETS 交互式工具的主程序。
TSOXGJOB	N/A	N/A	*SRVPGM	C	在 SOCKETS 交互式工具的支持中使用的服务程序
TSOJNI	N/A	N/A	*SRVPGM	C	用于在 SOCKETS 交互式工具支持中的 JSP 与 TSOSTSOC 之间进行交互的服务程序。
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	在创建 TSOXSOCK 程序时使用的模块。源文件包含 main() 例程。
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	在创建 TSOXSOCK 程序时使用的模块。源文件包含实际调用套接字函数的例程。
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	在创建 TSOXGJOB 服务程序时使用的模块。源文件包含标识内部作业的例程。此内部作业标识符由作业名、用户标识和作业号组成。
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	Xsockets 工具对包含套接字函数的主窗口使用的显示文件。
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	XSockets 在各种套接字函数的支持中使用的显示文件。
TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	支持菜单栏的 Xsockets 工具使用的显示文件。

表 20. Xsocket 安装期间创建的对象（续）

QATTIFS2	N/A	N/A	*FILE	PF-DTA	包含 Tomcat Server 使用的 JAR 文件。

将 Xsockets 配置为使用 Web 浏览器

以下一组指令允许您启用 Xsockets 工具以通过 Web 浏览器进行访问。可对同一系统多次实现这些指令以创建不同的服务器实例。这允许同时在不同侦听端口上运行多个版本。要将 Xsockets 配置为使用 Web 浏览器，必须完成下列任务:

1. 配置 HTTP Server（基于 Apache）
2. 配置 Tomcat
3. 更新配置文件
4. 在浏览器中测试 Xsockets 工具

配置 HTTP Server（基于 Apache）

在将 Web 浏览器配置为使用 Xsockets 工具之前，必须先完成本地 Xsockets 配置。下列步骤配置 HTTP Server（基于 Apache），以便在 Web 浏览器中使用 Xsockets 工具。

1. 验证 HTTP 管理实例是否正在 QHTTPSVR 子系统中运行。如果 HTTP 管理实例没有运行，则可以使用 CL 命令来启动它:

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```

2. 在 Web 浏览器中，输入

   ```
   http://<system_name>:2001/。
   ```

   其中 <system_name> 是 iSeries 的机器名。例如: http://myiSeries:2001/。
3. 在 iSeries 任务页面上选择 **IBM HTTP Server for iSeries**。
4. 从顶部菜单选择**设置**选项卡。
5. 单击**创建新的 HTTP Server**。
6. 选择 **HTTP Server（基于 Apache）**，然后单击**下一步**。
7. 输入服务器实例的名称。例如，由于此实例将在浏览器中使用 Xsocket 工具，所以可使用名称 `xsocket`。单击**下一步**。
8. 选择**否**。这将创建并非基于现有服务器的新服务器实例。单击**下一步**。
9. 单击**下一步**以接受服务器根目录的缺省值。
10. 单击**下一步**以接受文档根目录的缺省值。
11. 选择该 IP 地址和想要使用的可用端口。使用大于 1024 的端口号。单击**下一步**。

    注: 不要选择缺省端口号 80。
12. 选择**是**或**否**以指示是否想要为此服务器创建的访问作业记录。单击**下一步**。
13. 下一页显示 HTTP Server（基于 Apache）配置设置。如果这些设置是正确的，单击**完成**。
14. 单击**管理新创建的服务器**。现在使用 Apache 配置来完成。

下一步要执行的操作:

配置 Tomcat

配置 Tomcat

在配置 HTTP Server（基于 Apache）服务器实例之后，必须配置 Tomcat 才能在 Web 浏览器中启动 Xsockets 工具。

1. 在**动态内容**标题下，选择 **ASF Tomcat 设置服务器任务**。
2. 选择**对此 HTTP Server 启用 servlet**。这将填写工作程序定义文件。单击**下一步**。
3. 通过单击**下一步**接受**工作程序定义**页面上的缺省值。
4. 在 **URL 至工作程序映射**页面上，单击**添加**。
5. 在 **URL（安装点）**列中，输入 `/xsock`。单击**继续**。
6. 单击**添加**。
7. 在 **URL（安装点）**列中，输入 `/xsock/*`。单击**继续**。
8. 单击**下一步**。
9. 在**正在运行的应用程序上下文定义**页面上，单击**添加**。
10. 在 **URL 路径**列中，输入 `/xsock`。
11. 在**应用程序基本目录**列中，输入 `webapps/xsock`。
12. 单击**继续**。您将会看到一条警告消息，告诉您需要配置其它信息。
13. 在**配置应用程序**下单击**配置**。
14. 在新打开的浏览器窗口中，在**会话对象超时**字段中选择 3 天。

    注：这是建议值；但是，可对**会话对象超时**指定另一个值。
15. 单击**添加**以添加 Servlet 定义并完成下列步骤:
    a. 对于 **Servlet 类名**，输入 `com.ibm.iseries.xsocket.XSocketServlet`。
    b. 对于 **URL 模式**，输入 `/*`。
    c. 将**启动装入序列**设置为 3。
    d. 单击**继续**。
    e. 单击**确定**。这将关闭浏览器窗口。
16. 在 Tomcat 主安装窗口中，单击**下一步**。
17. 单击**完成**。
18. 单击**确定**。现在就完成了 Xsockets 工具的 Tomcat 配置。

下一步要执行的操作:

更新配置文件

更新配置文件

在 HTTP Server（基于 Apache）上配置 Tomcat 以使用 Xsockets 工具之后，必须对实例的几个配置文件完成手工更改。有三个文件需要更新: web.xml 文件、JAR 文件和 httpd.conf。要完成这些步骤，需要知道下列信息:

- 包含 Xsockets 应用程序文件的库名。此文件是在本地客户机的初始 Xsockets 配置期间创建的。
- 在 HTTP Server（基于 Apache）配置期间创建的服务器名。

1. **更新 web.xml 文件**
   a. 从命令行输入

```
wrklnk '/www/<server_name>/webapps/xsock/WEB-INF/web.xml'
```

其中 <server_name> 是在 Apache 配置期间创建的服务器实例的名称。例如，如果选择 xsocks 作为服务器名，则输入：

```
wrklnk '/www/xsocks/webapps/xsock/WEB-INF/web.xml'
```

b. 按 2 以编辑该文件。

c. 在 web.xml 文件中查找 </servlet-class> 行。

d. 在这一行后面插入下列代码：

```
<init-param>
        <param-name>library</param-name>
        <param-value>XXXX</param-value>
</init-param>
```

在 XXXX 位置上插入在 Xsockets 配置期间创建的库名。

e. 保存该文件并退出编辑会话。

2. **移动 JAR 文件**

a. 从命令行输入以下命令：

```
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)
```

其中 XXXX 是在 Xsockets 配置期间创建的库名，而 <server_name> 是在 HTTP Server（基于 Apache）配置期间创建的服务器实例的名称。

3. **将权限检查添加至 httpd.conf 文件**（此步骤是可选的）。
这将强制 Apache 认证尝试访问 Xsockets Web 应用程序的用户。

注： 这对获取创建 UNIX 套接字的写访问权也是必需的。

a. 从命令行输入

```
wrklnk '/www/<server_name>/conf/httpd.conf'
```

其中 <server_name> 是在 Apache 配置期间创建的服务器实例的名称。例如，如果选择 xsocks 作为服务器名，则输入：

```
wrklnk '/www/xsocks/conf/httpd.conf''
```

b. 按 2 以编辑该文件。

c. 在文件结束位置插入下列行。

```
<Location /xsock>
   AuthName "X Socket"
   AuthType Basic
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
   Require valid-user
   order allow,deny
   allow from all
</Location>
```

d. 保存该文件并退出编辑会话。

下一步要执行的操作

在 Web 浏览器中测试 Xsockets 工具

在 Web 浏览器中测试 Xsockets 工具

完成对配置文件的手工更新后，就可以在浏览器内测试 Xsocket 工具了。

1. 要启动服务器实例，在命令行上输入以下命令:

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<server_name>)
   ```

   其中 <server_name> 是在 Apache 配置期间创建的服务器实例的名称。

   注: 这需要一点时间。

2. 通过从命令行界面发出 WRKACTJOB 命令来检查它的状态。如果带有 server_name 的所有作业都处于 SIGW 状态，就可以继续进行至下一步。
3. 在浏览器中，输入以下 URL:

   ```
   http://<system_name>:<port>/xsock/index
   ```

   其中 <system_name> 和 <port> 是在 Apache 配置期间选择的服务器实例名和端口号。

4. 在出现提示时，输入服务器的用户名和密码。应该出现 Xsocket 的 Web 客户机。

使用 Xsockets

现在有两种方法使用 Xsockets 工具。可从本地客户机使用该工具或在 Web 浏览器中使用该工具。要使用本地版本的 Xsocket，必须配置 Xsockets 工具。如果想要在浏览器环境中使用 Xsockets 工具的话，除了为本地客户机配置 Xsockets 工具之外，还必须完成将 Xsockets 配置为使用 Web 浏览器中的步骤。两个版本的工具之间的许多概念是相似的。两个版本的工具都允许以交互方式发出套接字调用，也都会提供发出的套接字调用的错误号；但是，两者的界面有一些差别。下面一组指令显示如何在两种环境下使用 Xsocket 工具。

注: 如果想要使用套接字程序（这些程序使用 GSKit 安全套接字 API），则必须使用 Web 版本的工具。

下面一组指令描述如何使用每一种工具:

- 使用本地 Xsockets
- 使用基于浏览器的 Xsockets

使用本地 Xsockets

在完成这些步骤之前，确保您已经完成了针对本地 Xsockets 的所有配置步骤。要在本地客户机上使用 Xsocket，完成下列步骤:

1. 在命令行上，通过发出以下命令，将 Xsocket 工具所在的库添加至库列表:

   ```
   ADDLIBLE <library-name>
   ```

   其中 <library-name> 是在本地 Xsockets 配置期间创建的库的名称。例如，如果库名为 MYXSOCKET，则输入:

   ```
   ADDLIBLE MYXSOCKET
   ```

2. 在命令行界面上，输入

   ```
   CALL TSOXSOCK
   ```

3. 在显示的 Xsocket 窗口中，允许您通过其菜单栏和选择字段访问所有套接字例程。此窗口总是在选择套接字函数之后显示。可使用此界面选择已经存在的套接字程序。要使用新套接字，完成下列步骤:
   a. 在套接字函数列表中，选择**套接字**然后按**执行**键。
   b. 在显示的 **socket() 提示**窗口中，为套接字选择相应的“地址系列”、“套接字类型”和“协议”，并按**执行**键。

c. 选择**描述符**并选择**选择描述符**。

注: 如果已存在其它套接字描述符，这将显示活动套接字描述符的列表。

d. 从显示的列表中，选择您创建的套接字描述符

注: 如果存在其它套接字描述符，该工具将自动对最新的套接字描述符应用套接字函数。

4. 从套接字函数列表中，选择想要使用的套接字函数。无论您在步骤 3c 中选择了什么套接字描述符，都将在该套接字函数上使用它。一旦选择了套接字函数，就会显示一系列窗口，可在其中提供有关该套接字函数的特定信息。例如，如果选择 **connect()**，则需要在生成的窗口中提供地址长度、地址系列和地址数据。然后使用您提供的这些信息调用选择的套接字函数。在套接字函数上发生的所有错误都将以错误号的形式显示给用户。

注:

1. Xsockets 工具对 DDS 使用图形外观支持。因此，您所看到的输入数据的方式以及在窗口／面板上作出选择的方式将取决于您是使用图形显示站还是使用非图形显示站。例如，在图形显示站上，您将会看到套接字函数的选择字段是复选框；否则您可能会看到单个字段。
2. 应该注意，有一些在套接字上可用的 **ioctl()** 请求在工具中没有实现。

在 Web 浏览器中使用 Xsockets

在 Web 浏览器中使用 Xsockets 工具之前，确保已经完成了 Xsockets 的所有本地配置和所有必需的 Web 浏览器配置。要在 Web 浏览器中使用 Xsockets 工具，完成下列步骤:

1. 在 Web 浏览器中，输入:

   ```
   http://server-name:2001/
   ```

   其中 server-name 是包含服务器实例的 iSeries 的名称。
2. 选择**管理**。
3. 从左边的导航栏中，选择**管理 HTTP Server**。
4. 选择实例名，并单击**启动**。还可从命令行输入以下命令来启动服务器实例:

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<instance_name>)
   ```

   其中 <instance_name> 是在 Apache 配置中创建的 HTTP Server 的名称。例如，可使用服务器实例名 xsocks。
5. 要访问 Xsockets Web 应用程序，在浏览器中输入以下 URL:

   ```
   http://<system_name>:<port>/xsock/index
   ```

   其中 <system_name> 是 iSeries 的机器名，而 <port> 是在创建 HTTP 实例时指定的端口。例如，如果系统名为 myiSeries，HTTP Server 服务器实例在端口 1025 上进行侦听，则输入:

   ```
   http://myiSeries:1025/xsock/index
   ```
6. 一旦 Xsocket 工具在 Web 浏览器中装入，就可以使用现有套接字描述符或创建新的套接字描述符。两个版本的工具之间有许多概念是相似的。两种工具都允许以交互方式发出套接字调用，也都会提供发出的套接字调用的错误号；但是，两种工具在界面上有一些差别。要创建新的套接字描述符，可执行下列操作:
   a. 从 **Xsocket 菜单**中选择**套接字**。
   b. 在显示的 **Xsocket 查询**窗口中，为此套接字描述符选择相应的“地址系列”、“套接字类型”和“协议”。单击**提交**。
   c. 一旦页面重新装入，将在**套接字**下拉菜单中显示新的套接字描述符。

d. 从 **Xsocket 菜单**中，选择想要对此套接字描述符应用的函数调用。就本地版本的 Xsockets 工具而言，如果不选择套接字描述符，该工具将自动对最新的套接字描述符应用函数调用。

删除 Xsocket 工具创建的对象

您可能需要删除 Xsockets 工具创建的对象。名为 TSODLT 的程序是由安装程序创建的，用来除去该工具创建的对象（库和程序 TSODLT 除外）和／或除去 Xsocket 工具使用的源成员。以下一组命令允许您删除这些对象:

要仅删除该工具使用的源成员，输入以下命令:

```
CALL TSODLT   (*YES *NONE)
```

要仅删除该工具创建的对象，输入以下命令:

```
CALL TSODLT   (*NO  library-name)
```

要同时删除该工具创建的源成员和对象，输入以下命令:

```
CALL TSODLT   (*YES library-name)
```

定制 Xsockets

可通过添加对套接字网络例程的附加支持来更改 Xsockets 工具，例如 **inet_addr()**。如果选择定制此工具以满足您自己的需要，建议不要在 QUSRTOOL 库中进行更改。而是将源文件复制到单独的库中并在该库中进行更改。这将保留 QUSRTOOL 库中的原始文件，所以在将来有需要的时候，这些文件仍然是可用的。可在作出更改后使用 TSOCRT 程序重新编译该工具（注意，如果将源文件复制到单独的库中，您还需要在 TSOCRT 中进行更改才能使用它）。在创建该工具之前，使用 TSODLT 程序来除去工具的旧版本。

第 12 章 可服务性工具

由于套接字和安全套接字的使用量持续增长以适应电子商务应用程序和服务器，所以目前的可服务性工具需要跟上这一需求。增强型可服务性工具允许您完成对套接字程序的跟踪，以找出套接字和启用 SSL 的应用程序中出现的错误的解决方案。这些工具通过选择套接字特征（如 IP 地址或端口信息）帮助您和支持中心工作人员确定发生套接字问题的位置。

下表给出每个服务工具的概述。

表 21. 用于套接字和安全套接字的可服务性工具

可服务性工具	描述
LIC 跟踪过滤（TRCINT 和 TRCCNN）	提供对套接字的选择性跟踪。现在可将套接字跟踪限制为地址系列、套接字类型、协议、IP 地址和端口信息。还可将跟踪限制为仅针对某个类别的套接字 API 以及设置了 SO_DEBUG 套接字选项的套接字。LIC 跟踪是 V5R2 的新增内容，现在可按线程、任务、用户概要文件、作业名或服务器名进行过滤。
使用 STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR)) 跟踪作业	STRTRC 命令是 V5R2 的新增内容，它提供了一些附加参数，它生成的输出与所有其它非套接字相关跟踪点不同。如果在套接字操作期间遇到错误，此输出会包含返回码和错误号信息。有关详细信息，参见“信息中心”中的 STRTRC（启动跟踪）命令描述。
飞行记录器跟踪	套接字 LIC 组件跟踪将包括每个执行的套接字操作的飞行记录器项的转储。
关联作业信息	允许服务人员和程序员查找与连接的套接字或侦听套接字相关联的所有作业。可对使用地址系列 AF_INET 或 AF_INET6 的套接字应用程序使用 NETSTAT 来查看此信息。
NETSTAT 连接状态（选项“3”），用来启用 SO_DEBUG	在套接字应用程序上设置了 SO_DEBUG 的情况下提供增强型低级别调试信息。
安全套接字返回码和消息处理	通过两个 SSL_ API 提供标准化安全套接字返回码消息。这些 API 包括 **SSL_Strerror()** 和 **SSL_Perror()**。此外，**gsk_strerror()** 提供与 GSKit API 类似的功能。**hstrerror()** API 还提供了来自解析程序例程的返回码信息。
性能数据收集（PDC）跟踪点	允许通过套接字和 TCP/IP 堆栈跟踪来自应用程序的数据流。

© Copyright IBM Corp. 2001, 2002

第 13 章 相关信息

下面列示的内容包括 IBM 红皮书（PDF 格式）、Web 站点和提供有关套接字编程的更多信息的“信息中心”主题。可查看或打印任何 PDF。

IBM 红皮书

- Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More

  （大约 454 页）
- IBM @server iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements

  （大约 506 页）

请求注释文件（RFC）

- **IPv6**
  - RFC 2553: ″Basic Socket Interface Extensions for IPv6″
  - RFC 2292: ″Advanced Sockets API for IPv6″
- 域名系统
  - RFC 1034: ″Domain names - concepts and facilities″
  - RFC 1035: ″Domain names - implementation and specification″
  - RFC 2136: ″Dynamic Updates in the Domain Name System (DNS UPDATE)″
  - RFC 2181: ″Clarifications to the DNS Specification″
  - RFC 2308: ″Negative Caching of DNS Queries (DNS NCACHE)″
  - RFC 2845: ″Secret Key Transaction Authentication for DNS (TSIG)″
- 安全套接字层／传输层安全性
  - RFC 2246: ″The TLS Protocol Version 1.0 ″

其它 Web 资源

- Technical Standard: Networking Services (XNS), Issue 5.2 Draft 2.0.

© Copyright IBM Corp. 2001, 2002

第 14 章 代码不保证声明信息

本文档包含编程示例。

IBM 授予您非专有版权许可证以使用所有编程代码示例，您可根据这些示例生成相似的功能以满足您自己的特定需要。

IBM 提供所有样本代码仅出于举例说明目的。这些示例未在所有情况下经过严格测试。因此 IBM 并不保证或暗示这些程序的可靠性、可服务性或功能。

此处包含的所有程序“按现状”提供给您，但并未作出任何保证。我们明确放弃对非侵权，为特定目的的适销性和适用性的隐含保证。

© Copyright IBM Corp. 2001, 2002

IBM

中国印刷