iSeries

Socket programming

Version 5 Release 3

iSeries

Socket programming

Version 5 Release 3

Note

Before using this information and the product it supports, be sure to read the information in "Notices" on page 179.

fifth Edition (August 2005)

This edition applies to version 5, release 3, modification 0 of the Operating System/400® (5722–SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 2001, 2005. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2001, 2005

Socket programming

A **socket** is a communications connection point (endpoint) that you can name and address in a network. The processes that use a socket can reside on the same system or on different systems on different networks. Sockets are useful for both stand-alone and network applications. Sockets allow you to exchange information between processes on the same machine or across a network, distribute work to the most efficient machine, and allows access to centralized data easily. Socket application program interfaces (APIs) are the network standard for TCP/IP. A wide range of operating systems support socket APIs. OS/400® sockets support multiple transport and networking protocols. Socket system functions and the socket network functions are threadsafe.

Socket programming shows how to use socket APIs to establish communication links between remote and local processes. Programmers who use Integrated Language Environment® (ILE) C can use the information to develop socket applications. You can also code to the sockets API from other ILE languages, such as RPG. For more information on ILE RPG, see the IBM® Redbook Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More.

Java™ also supports a socket programming interface. See the Java topic in the Information Center for details.

Socket programming topics:

The following topics provide concepts, design recommendations, and examples to help you develop socket applications. see:

- **Print this topic**
  Use this page to print or download a Portable Document Format (PDF) version of the Sockets programming information.
- **Prerequisites for sockets programming**
  This topic discusses the required tasks that must be completed before writing applications with socket APIs.
- **Basic socket design**
  This topic provides an overview for example programs for the most basic types of sockets. Use the links to sample programs for examples that illustrate basic socket design strategies.
- **Socket concepts**
  This topic describes more advanced socket concepts, such as Asynchronous input/output (I/O) and Global Secure Toolkit (GSKit). Use the links in this topic to review sample programs that are associated with these concepts.
- **Socket Scenario: Create an application to accept IPv4 and IPv6 clients**
  This topic describes a typical situation in which you may want to use the AF_INET6 address family. New for V5R2, this address family provides support for Internet Protocol version 6 (IPv6). IPv6 supports 128–bit IP addresses. This topic also provides planning information and links to example programs that you can use to implement socket applications that use the AF_INET6 address family.
- **Socket application design recommendations**
  This topic provides hints to designing more effective socket applications.
- **Examples: Socket application design**
  This topic provides example socket programs that you can use to create socket applications.

  Note: This information contains example code. See Code disclaimer information for details on the use of these sample programs.

© Copyright IBM Corp. 2001, 2005

- **Xsockets tool**
  This topic provides a description of the Xsockets tool which socket programmers can use to help develop socket applications. Use the links in this topic to read instructions on installing and using the tool.
- **Serviceability tools**
  This topic provides descriptions of serviceability tools for sockets.
- **Related information**
  This topic provides links and descriptions of other socket information.

Print this topic

You can view or download a Portable Document Format (PDF) version of this document for viewing or printing. You must have Adobe® Acrobat® Reader installed to view PDF files. You can download a copy from http://www.adobe.com/prodindex/acrobat/readstep.html .

To view or download the PDF version, select Socket programming. (444KB and 132 pages)

To save a PDF on your workstation for viewing or printing:

1. Open the PDF in your browser (click the link above).
2. In the menu of your browser, click **File**.
3. Click **Save As...**
4. Navigate to the directory in which you would like to save the PDF.
5. Click **Save**.

Prerequisites for socket programming

Before writing socket applications you must first complete these steps:

Compiler requirements

1. Install QSYSINC library. This library provides necessary header files that are needed when compiling socket applications.
2. Install the C Compiler licensed program (5722–CX2).

Requirements for AF_INET and AF_INET6 address families

In addition to the compiler requirements, you must complete the following:

1. Plan TCP/IP
2. Install TCP/IP
3. Configure TCP/IP for the first time
4. Configure TCP/IP for IPv6. This step is optional. Configure an IPv6 interface for TCP/IP if you plan to write applications that use the AF_INET6 address family.

Requirements for Secure Sockets Layer (SSL) and Global Secure Toolkit (GSKit) APIs

In addition to compiler and AF_INET and AF_INET6 address requirements, you must complete the following tasks to work with secure sockets:

1. Install and configure Digital Certificate Manager licensed program (5722–SS1 Option 34). See Digital Certificate Management in the Information Center for details.
2. Install Cryptographic Access Provider licensed program (5722– AC3).
3. If you want to use SSL with the cryptographic hardware, you will need to install and configure either the 2058 Cryptographic Accelerator for iSeries™ or the 4758 PCI Cryptographic Coprocessor for iSeries. The 2058 Cryptographic Accelerator allows you to offload SSL cryptographic processing from

the operating system to the card. See 2058 Cryptographic Accelerator for iSeries for a complete description of the 2058 Cryptographic Accelerator and its features. The 4758 Cryptographic Coprocessor can be used for SSL cryptographic processing; however, unlike the 2058, this card provides a more cryptographic functions, like encrypting and decrypting keys. See 4758 PCI Cryptographic Coprocessor for iSeries for features and configuration steps for this card.

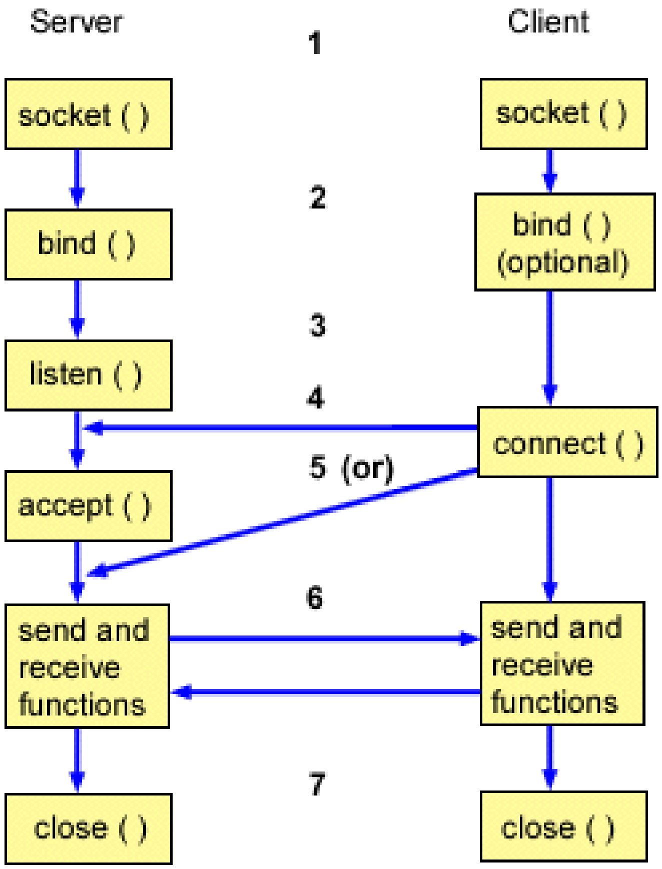
How do sockets work?

Sockets are commonly used for client/server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.

A socket has a typical flow of events. In a connection-oriented client-to-server model, the socket on the server process waits for requests from a client. To do this, the server first establishes (binds) an address that clients can use to find the server. When the address is established, the server waits for clients to request a service. The client-to-server data exchange takes place when a client connects to the server through a socket. The server performs the client's request and sends the reply back to the client.

Note: Currently, IBM supports two versions of most sockets APIs. The default OS/400 sockets use Berkeley Socket Distribution (BSD) 4.3 structures and syntax. The differences between the base OS/400 sockets and BSD 4.3 are outlined in Berkeley Socket Distribution (BSD) compatibility. The other version of sockets uses syntax and structures compatible with BSD 4.4 and the UNIX® 98 programming interface specifications. Programmers can specify _XOPEN_SOURCE macro to use the UNIX98 compatible interface. See UNIX 98 compatibility for descriptions of these API and structural differences.

The following figure shows the typical flow of events (and the sequence of issued functions) for a connection-oriented socket session. An explanation of each event follows the figure.



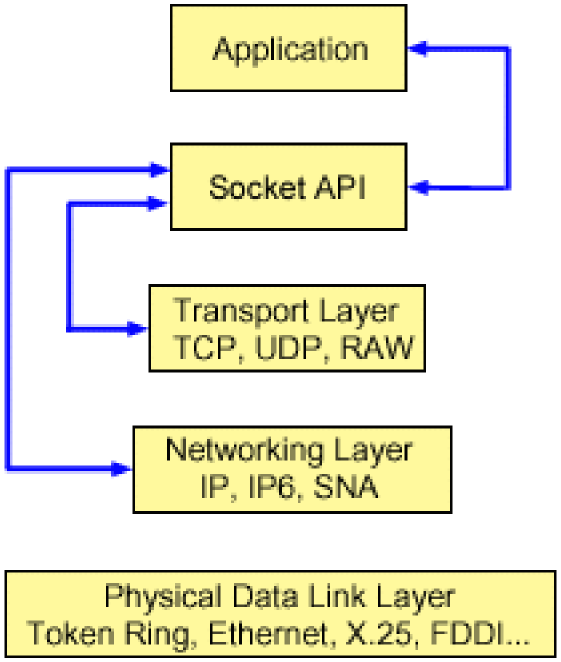
Typical flow of events for a connection-oriented socket:

1. The **socket()** function creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. When an application has a socket descriptor, it can bind a unique name to the socket. Servers must bind a name to be accessible from the network.
3. The **listen()** function indicates a willingness to accept client connection requests. When a **listen()** is issued for a socket, that socket cannot actively initiate connection requests. The **listen()** API is issued after a socket is allocated with a **socket()** function and the **bind()** function binds a name to the socket. A **listen()** function must be issued before an **accept()** function is issued.
4. The client application uses a **connect()** function on a stream socket to establish a connection to the server.
5. The server application uses the **accept()** function to accept a client connection request. The server must issue the **bind()** and **listen()** functions successfully before it can issue an **accept()**.
6. When a connection is established between stream sockets (between client and server), you can use any of the socket API data transfer functions. Clients and servers have many data transfer functions from which to choose, such as **send()**, **recv()**, **read()**, **write()**, and others.
7. When a server or client wants to cease operations, it issues a **close()** function to release any system resources acquired by the socket.

Note:

> The socket APIs are located in the communications model between the application layer and the transport layer. The socket APIs are not a layer in the communication model. Socket APIs allow applications to interact with the transport or networking layers of the typical communications

model. The arrows in the following figure show the position of a socket, and the communication layer that the socket provides.



Typically, a network configuration does not allow connections between a secure internal network and a less secure external network. However, you can enable sockets to communicate with server programs that run on a system outside a firewall (a very secure host).

Sockets are also a part of IBM's AnyNet® implementation for the Multiprotocol Transport Networking (MPTN) architecture. MPTN architecture provides the ability to operate a transport network over additional transport networks and to connect application programs across transport networks of different types.

The Information Center allows you to access API reference information several ways. The topic Socket API provides an overview of socket functions and structures. If you want to search for specific API or to search for categories of APIs, the API information provides an interactive API finder to assist you.

Socket characteristics

Sockets share the following characteristics:

- A socket is represented by an integer. That integer is called a **socket descriptor**.
- A socket exists as long as the process maintains an open link to the socket.
- You can name a socket and use it to communicate with other sockets in a communication domain.
- Sockets perform the communication when the server accepts connections from them, or when it exchanges messages with them.
- You can create sockets in pairs (only for sockets in the AF_UNIX address family).

The connection that a socket provides can be connection-oriented or connectionless. **Connection-oriented** communication implies that a connection is established, and a dialog between the programs will follow. The program that provides the service (the server program) establishes the available socket which is enabled to accept incoming connection requests. Optionally, the server can assign a name to the service that it supplies which allows clients to identify where to obtain and how to connect to that service. The client of the service (the client program) must request the service of the server program. The client does this by connecting to the distinct name or to the attributes associated with the distinct name that the server program has designated. It is similar to dialing a telephone number (an identifier) and making a connection with another party that is offering a service (for example, a plumber). When the receiver of the call (the server, in this example, a plumber) answers the telephone, the connection is established. The plumber verifies that you have reached the correct party, and the connection remains active as long as both parties require it.

Connectionless communication implies that no connection is established over which a dialog or data transfer can take place. Instead, the server program designates a name that identifies where to reach it (much like a post office box). If you send a letter to a post office box, you cannot be absolutely sure the receiver got the letter. You may need to wait for a response to your letter. There is no active, real time connection in which data is exchanged.

How socket characteristics are determined

When an application creates a socket with the **socket()** function, it must identify the socket by specifying these parameters:

- Socket address family determines the format of the address structure for the socket. This topic contains examples of each address families' address structure. See Socket address structures for a general definition of structure of socket addresses.
- Socket type determines the desired form of communication for the socket.
- Socket-supported protocols determines the protocol that the socket uses.

These parameters or characteristics define the socket application and how it interoperates with other socket applications. Depending on the address family of a socket, you may have different choices for the socket type and protocol. The table below shows the corresponding address family and its associated socket type and protocols:

Table 1. Summary of socket characteristics

Address family	Socket type	Socket protocol
AF_UNIX	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP, ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6, ICMP6
AF_TELEPHONY	SOCK_STREAM	N/A
AF_UNIX_CCSID	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A

In addition to these socket characteristics or parameters, constant values are defined in network routines and header files that are shipped with the QSYSINC library. For descriptions of header files, see the individual APIs listed in the Socket APIs topic in the Information Center. Each API lists its appropriate header file in the usage section of the API description.

Socket network routines allow socket applications to obtain information from the DNS, host, protocol, service, and network files. For a description of these routines, see Socket network routines.

Socket address structure

Sockets use the **sockaddr** address structure to pass and to receive addresses. This structure does not require the socket API to recognize the addressing format. Currently OS/400 supports Berkeley Software Distributions (BSD) 4.3 and X/Open Single Unix Specification (UNIX 98). The base OS/400 API uses BSD 4.3 structures and syntax. You can select the UNIX 98 compatible interface by defining the _XOPEN_SOURCE macro to a value of 520 or greater. Each socket address structure for BSD 4.3 that is used will have an equivalent UNIX 98 structure.

Table 2. Comparison of BSD 4.3 and UNIX 98/BSD 4.4 socket address structure

BSD 4.3 structure	**BSD 4.4/UNIX 98 compatible structure**

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];
	};

Table 3. Address structure

Address structure field	**Definition**
sa_len	This field contains the length of the address for UNIX 98 specifications. **Note:** The: sa_len field is only provided for BSD 4.4 compatibility. It is not necessary to use this field even when using BSD 4.4/UNIX 98 compatibility. The field is ignored on input addresses.
sa_family	This field defines the address family. This value is specified for the address family on the **socket()** call.
sa_data	This field contains fourteen bytes that are reserved to hold the address itself. **Note:** The sa data length of 14 bytes is a placeholder for the address. The address can exceed this length. The structure is generic because it does not define the format of the address. The format of the address is defined by the type of transport which a socket is created for. Each of the transport providers will define the exact format for its specific addressing requirements in a similar address structure. The transport is identified by the protocol parameter values on the **socket()** API.
sockaddr_storage	Declares storage for any address family address. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as sockaddr structure for use on the APIs. The ss_family field of the sockaddr_storage will always align with the family field of any protocol-specific structure.

Socket address family

The address family parameter on a **socket()** determines the format of the address structure to use on socket functions. Address family protocols provide the network transportation of application data from one application to another (or from one process to another within the same machine). The application specifies the network transport provider on the protocol parameter of the socket.

The address family parameter (address_family) on the **socket()** function specifies the address structure that is used on the socket functions. The following topics describe each of these address families, their use, their related protocol, and examples of relevant structure:

- AF_INET address family
- AF_INET6 address family
- AF_UNIX address family
- AF_UNIX_CCSID address family
- AF_TELEPHONY address family

AF_INET address family

This address family provides Inter Process Communications between processes that run on the same system or on different systems. Addresses for AF_INET sockets are IP addresses and port number. You can specify an IP address for an AF_INET socket either as an IP address, such as 130.99.128.1, or in its 32–bit form, X'82638001'.

For a socket application that uses the Internet Protocol version 4 (IPv4), the AF_INET address family uses the **sockaddr_in** address structure. When you use _XOPEN_SOURCE macro, the AF_INET address structure changes to be compatible with BSD 4.4/UNIX 98 specifications. For the sockaddr_in address structure, these differences are summarized in the table:

Table 4. Differences between BSD 4.3 and BSD 4.4/UNIX 98 for sockaddr_in address structure

BSD 4.3 sockaddr_in address structure	**BSD 4.4/UNIX 98 sockaddr_in address structure**

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];
	};

Table 5. AF_INET address structure

Address structure field	**Definition**
sin_len	This field contains the length of the address for UNIX 98 specifications. **Note:** The: sin_len field is only provided for BSD 4.4 compatibility. It is not necessary to use this field even when using BSD 4.4/UNIX 98 compatibility. The field is ignored on input addresses.
sin_family	This field contains the address family, which is always AF_INET when TCP or UDP is used.
sin_port	This field contains the port number.
sin_addr	This field contains the Internet address.
sin_zero	This field is reserved. Set this field to hexadecimal zeros.

See Use AF_INET address family for information on using AF_INET and a sample program that uses AF_INET address family.

AF_INET6 address family

This address family provides support for the Internet Protocol version 6, (IPv6). AF_INET6 address family uses a 128 bit (16 byte) address. The basic architecture of these addresses includes 64 bits for a network number and another 64 bits for the host number. You can specify AF_INET6 addresses as x:x:x:x:x:x:x:x, where the ’x’s are the hexadecimal values of eight 16–bit pieces of the address. For example, a valid address would look like this: FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

For a socket application that uses TCP, UDP or RAW, the AF_INET6 address family uses the **sockaddr_in6** address structure. This address structure changes if you use _XOPEN_SOURCE macro to implement BSD 4.4/UNIX 98 specifications. For the sockaddr_in6 address structure, these differences are summarized in this table:

Table 6. Differences between BSD 4.3 and BSD 4.4/UNIX 98 for sockaddr_in6 address structure

BSD 4.3 sockaddr_in6 address structure	BSD 4.4/UNIX 98 sockaddr_in6 address structure

`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;`
	`};`

Table 7. AF_INET6 address structure

Address structure field	Definition
sin6_len	This field contains the length of the address for UNIX 98 specifications. **Note:** The: sin6_len field is only provided for BSD 4.4 compatibility. It is not necessary to use this field even when using BSD 4.4/UNIX 98 compatibility. The field is ignored on input addresses.
sin6_family	This field specifies the AF_INET6 address family.
sin6_port	This field contains the transport layer port.
sin6_flowinfo	This field contains two pieces of information: the traffic class and the flow label. **Note:** This field is currently not supported and should be set to zero for upward compatibility.
sin6_addr	This field specifies the IPv6 address.
sin6_scope_id	This field identifies a set of interfaces as appropriate for the scope of the address carried in the **sin6_addr** field. **Note:** This field is currently not supported and should be set to zero for upward compatibility.

AF_UNIX address family

This address family provides Inter Process Communications on the same system that use the socket APIs. The address is really a path name to an entry in the file system. You can create sockets in the root directory or any open file system but file systems such as QSYS or QDOC. The program must bind an AF_UNIX, SOCK_DGRAM socket to a name to receive any datagrams back. In addition, the program must explicitly remove the file system object with the **unlink()** API when the socket is closed.

Sockets with the address family AF_UNIX use the **sockaddr_un** address structure. This address structure changes if you use _XOPEN_SOURCE macro to implement BSD 4.4/UNIX 98 specifications. For the sockaddr_un address structure, these differences are summarized in the table:

Table 8. Differences between BSD 4.3 and BSD 4.4/UNIX 98 for sockaddr_un address structure

BSD 4.3 sockaddr_un address structure	BSD 4.4/UNIX 98 sockaddr_un address structure

Table 8. Differences between BSD 4.3 and BSD 4.4/UNIX 98 for sockaddr_un address structure (continued)

`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];`
	`};`

Table 9. AF_UNIX address structure

Address structure field	**Definition**
sun_len	This field contains the length of the address for UNIX 98 specifications. **Note:** The: sun_len field is only provided for BSD 4.4 compatibility. It is not necessary to use this field even when using BSD 4.4/UNIX 98 compatibility. The field is ignored on input addresses.
sun_family	This field contains the address family.
sun_path	This field contains the path name to an entry in the file system.

For AF_UNIX address family, protocol specifications do not apply because protocol standards are not involved. The communications mechanism the two processes use is specific to the machine.

See Use AF_UNIX address family for information on using AF_UNIX and provides a sample programs that use this address family.

AF_UNIX_CCSID address family

The AF_UNIX_CCSID family is compatible with the AF_UNIX address family and has the same limitations. They both can be either connectionless or connection-oriented, and no external communication functions connect the two processes. The difference is that sockets with the address family AF_UNIX_CCSID use the **sockaddr_unc** address structure. This address structure is similar to sockaddr_un, but it allows path names in UNICODE or any CCSID by using the **Qlg_Path_Name_T** format. See path name format in the Information Center.

However, because an AF_UNIX socket may return the path name from an AF_UNIX_CCSID socket in an AF_UNIX address structure, path size is limited. AF_UNIX only supports 126 characters so AF_UNIX_CCSID will also be limited to 126 characters.

A user may not exchange AF_UNIX and AF_UNIX_CCSID addresses on a single socket. When AF_UNIX_CCSID is specified on the **socket()** call, all addresses must be **sockaddr_unc** on later API calls.

```
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;
    } ;
```

Table 10. AF_UNIX_CCSID address structure

Address structure field	**Definition**

Table 10. AF_UNIX_CCSID address structure (continued)

sunc_family	This field contains the address family, which is always AF_UNIX_CCSID.
sunc_format	This field contains two defined values for the format of the path name: • **SO_UNC_DEFAULT** indicates a wide path name using the current default CCSID for integrated file system path names. The sunc_qlg field is ignored. • **SO_UNC_USE_QLG** indicates that the **sunc_qlg** field defines the format and CCSID of the path name.
sunc_zero	This field is reserved. Set this field to hexadecimal zeros.
sunc_qlg	This field specifies the path name format.
sunc_path	This field contains the path name. It is a maximum of 126 characters and may be single byte or double byte. It may be contained within the sunc_path field or allocated separately and pointed to by sunc_path. The format is determined by sunc_format and sunc_qlg.

For more information on AF_UNIX_CCSID sockets, see Use AF_UNIX_CCSID address family, which provides a sample program.

AF_TELEPHONY address family

AF_TELEPHONY address family permit users to dial and answer telephone calls through an ISDN telephone network using standard socket APIs. The socket that forms the endpoints for the connection in this domain are really the called and the calling parties of a telephone call. Addresses for this address family are represented by a 40–digit telephone number. This address family is most commonly used when developing fax support.

The system supports AF_TELEPHONY sockets only as a connection-oriented sockets, which have the socket type, SOCK_STREAM. A connection in a telephony domain socket provides no more reliability than the underlying telephone connection. If you want guaranteed delivery, you must work with the applications that provide these services, such as the fax applications that use this address family.

Sockets with AF_TELEPHONY address family use the **sockaddr_tel** address structure:

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

The telephony address consists of a 2–byte length followed by a telephone number of up to 40 digits (0–9).

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

Table 11. AF_TELEPHONY address structure

Address structure field	**Definition**
stel_family	This field contains the address family.
stel_addr	This field contains the telephony address.
stel_zero	This field is a reserved field.

For more information on AF_TELEPHONY address family, see Use AF_TELEPHONY address family, which provides steps on configuring the environment for using AF_TELEPHONY address family.

Socket type

The second parameter on a socket call determines the type of socket. Socket type provides identification of the type and characteristics of the connection that will be enabled for transportation of the data from one machine to another or from one process to another. The following list describes the socket types that iSeries supports :

Stream (SOCK_STREAM)
This type of socket is connection-oriented. Establish an end-to-end connection by using the **bind()**, **listen()**, **accept()**, and **connect()** functions. SOCK_STREAM sends data without errors or duplication, and receives the data in the sending order. SOCK_STREAM builds flow control to avoid data overruns. It does not impose record boundaries on the data. SOCK_STREAM considers the data to be a stream of bytes. In the iSeries implementation, you can use stream sockets over Transmission Control Protocol (TCP), Systems Network Architecture (SNA), AF_UNIX, AF_UNIX_CCSID, and AF TELEPHONY sockets. You can also use stream sockets to communicate with systems outside a secure host (firewall).

Datagram (SOCK_DGRAM)
In Internet Protocol terminology, the basic unit of data transfer is a **datagram**. This is basically a header followed by some data. The datagram socket is connectionless. It establishes no end-to-end connection with the transport provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. You can lose or duplicate data. Datagrams can arrive out of order. The size of the datagram is limited to the data size that you can send in a single transaction. For some transport providers, each datagram can use a different route through the network. You can issue a **connect()** function on this type of socket. However, on the **connect()** function, you must specify the destination address that the program sends to and receives from. In the iSeries implementation, you can use datagram sockets over user datagram protocol (UDP), SNA, and with AF_UNIX and AF_UNIX_CCSID address families.

Raw (SOCK_RAW)
This type of socket allows direct access to lower-layer protocols, such as Internet Protocol (IPv4 or IPv6) and Internet Control Message Protocol (ICMP or ICMP6). SOCK_RAW requires more programming expertise because you manage the protocol header information that is used by the transport provider. At this level, the transport provider may dictate the format of the data and the semantics that are transport-provider specific.

Socket protocols

Protocols provide the network transportation of an application's data from one machine to another (or from one process to another within the same machine). The application specifies the transport provider on the **protocol** parameter of the **socket()** function.

For the AF_INET address family, more than one transport provider is allowed. The protocols of SNA, TCP/IP, or UDP/IP can be active on the same socket at the same time. The ALWANYNET (Allow ANYNET support) network attribute allows a customer to select whether a transport other than TCP/IP can be used for AF_INET socket applications. This network attribute can be either ***YES** or ***NO**. The default value is ***NO**.

For example, if the current status (the default status) is *NO, the use of AF_INET over an SNA transport is not active. If AF_INET sockets are to be used over a TCP/IP transport only, the ALWANYNET status should be set to *NO to improve CPU utilization.

Note: The ALWANYNET network attribute also affects APPC over TCP/IP.

For information on APPC configuration options, see Configuring APPC, APPN, and HPR.

AF_INET sockets over TCP/IP can also specify a type of SOCK_RAW, which means that the socket communicates directly with the network layer known as Internet Protocol (IP). TCP or UDP transport providers normally communicate with this layer. When you use SOCK_RAW sockets, the application program specifies any protocol between 0 and 255 (except the TCP and UDP protocols). This protocol number then flows in the IP headers when machines are communicating on the network. In effect, the application program is the transport provider, because it must provide for all the transport services that UDP or TCP transports normally provide.

For the AF_UNIX, AF_UNIX_CCSID, and AF_TELEPHONY address families, a protocol specification is not really meaningful because there are no protocol standards involved. The communications mechanism between two processes on the same machine is specific to the machine.

Basic socket design

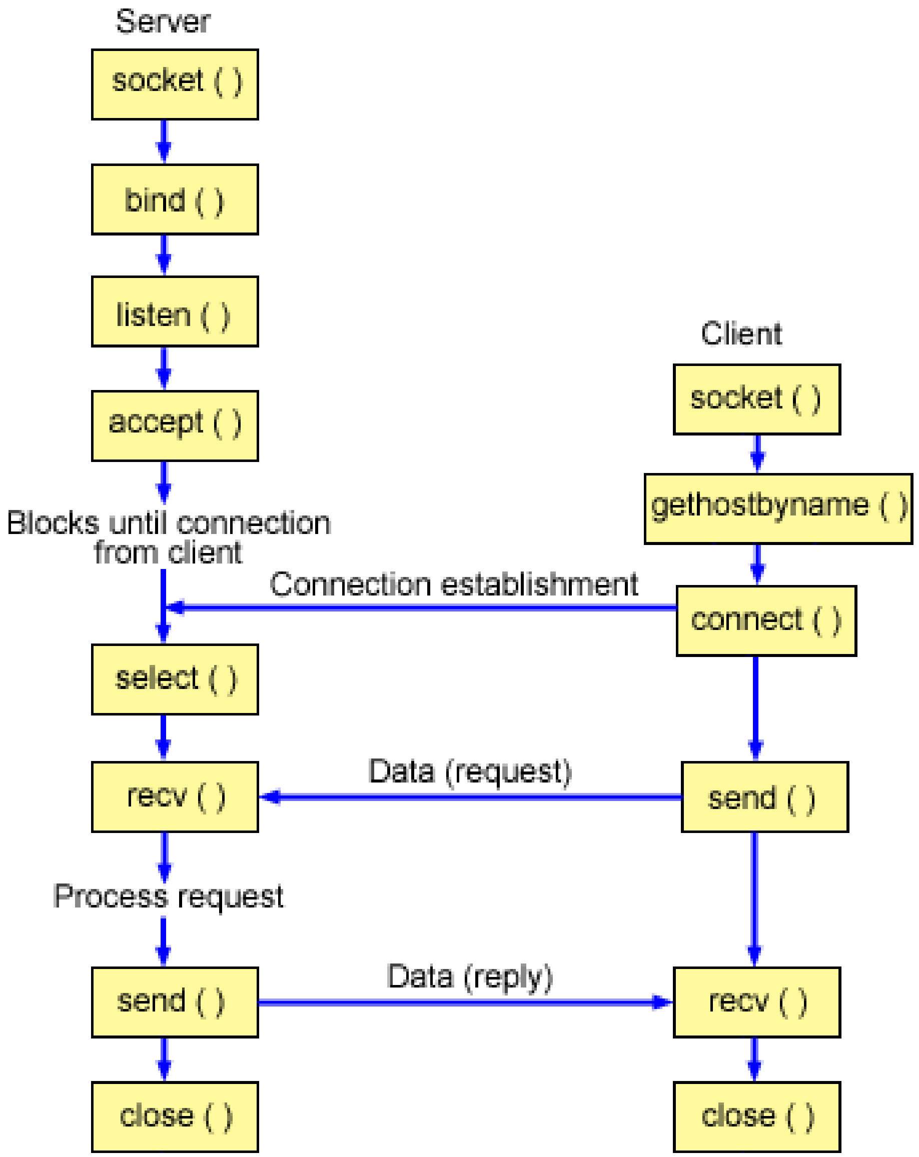
This topic provides examples of sockets programs that use the most basic design. These examples provide a basis for more complex socket designs. They implement some of the basic concepts that previous topics highlighted. The following sample programs provide examples of the most common types of socket programs.

- Create a connection-oriented socket
- Create a connectionless socket
- Design applications using address families

Create a connection-oriented socket

These server and client examples illustrate socket APIs written for a connection-oriented protocol such as Transmission Control Protocol (TCP).

The following figure illustrates the client/server relationship of the sockets API for a connection-oriented protocol.



Socket flow of events: Connection-oriented server
The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and client application in a connection-oriented design. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The Example: A connection-oriented server uses the following sequence of function calls:

1. 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.
2. The **setsockopt()** function allows the local address to be reused when the server is restarted before the required wait time expires.
3. After the socket descriptor is created, the **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 IPv4 client that specifies port 3005.

4. The **listen()** 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.
5. 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.
6. 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 be read. A 30 second timeout is used on this select call.
7. The **recv()** function receives data from the client application. 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 do not want our **recv()** to wake up until all 250 bytes of data have arrived.
8. The **send()** function echoes the data back to the client.
9. The **close()** function closes any open socket descriptors.

Socket flow of events: Connection-oriented client

The Example: A connection-orientated client uses the following sequence of function calls:

1. 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.
2. In the client example program, if the server string that was passed into the **inet_addr()** function was not a dotted decimal IP address, then it is assumed to be the hostname of the server. In that case, use the **gethostbyname()** function to retrieve the IP address of the server.
3. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
4. The **send()** function sends 250 bytes of data to the server.
5. The **recv()** function waits for the server to echo the 250 bytes of data back. In this example, we know that the server is going to respond with the same 250 bytes that we just sent. In client example, the 250 bytes of the data may arrive in separate packets, so we will use the **recv()** function over and over until all 250 bytes have arrived.
6. The **close()** function closes any open socket descriptors.

Example: A connection-oriented server

The following code example show how a connection-oriented server can be created. You can use this example to create your own socket server application. A connection-oriented server design is one of the most common models for socket applications. In a connection-oriented design, the server application creates a socket to accept client requests. For information on the use of code examples, see the code disclaimer.

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

Example: A connection-oriented client

The following example shows how to create a socket client program to connect a connection-oriented server in a connection-orient design. The client of the service (the client program) must request the service of the server program. You can use this code example to write your own client application. For information on the use of code examples, see the code disclaimer.

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

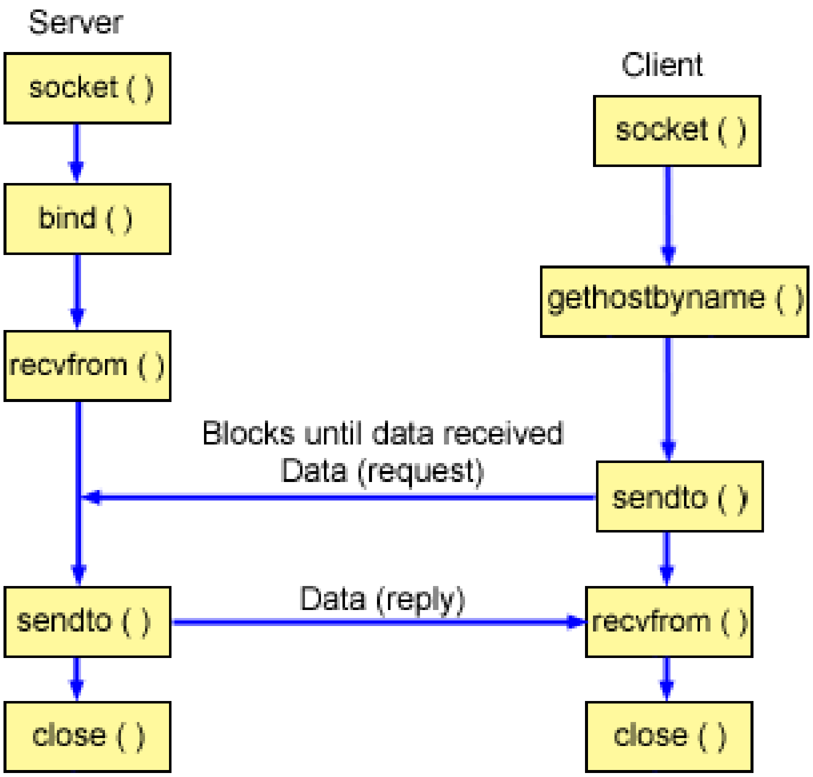
```
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* 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);
}
```

Create a connectionless socket

Connectionless sockets do not establish a connection over which data is transfered. Instead, the server application specifies its name where a client can send requests. Connectionless sockets use User Datagram Protocol (UDP) instead of TCP/IP.Example: A connectionless server and Example: A connectionless client illustrate the socket APIs that are written for User Datagram Protocol (UDP).

The following figure illustrates the client/server relationship from the socket APIs used in the code examples for a connectionless socket design.



Socket flow of events: Connectionless server
The following sequence of the socket calls provide a description of the graphic and the following example programs. It also describes the relationship between the server and client application in a connectionless design. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The Example: A connectionless server uses the following sequence of function calls:

1. 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.
2. 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 IPv4 addresses on the system.
3. The server uses the **recvfrom()** function to receive that data. The **recvfrom()** function waits indefinitely for data to arrive.

4. The **sendto()** function echoes the data back to the client.
5. The **close()** function ends any open socket descriptors.

Socket flow of events: Connectionless client
The Example: A connectionless client uses the following sequence of function calls:

1. 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.
2. In the client example program, it the server string that was passed into the inet_addr() function was not a dotted decimal IP address, then it is assumed to be the hostname of the server. In that case, use the **gethostbyname()** function to retrieve the IP address of the server.
3. Use the **sendto()** function to send the data to the server.
4. Use the **recvfrom()** function to receive the data back from the server.
5. The **close()** function ends any open socket descriptors.

Example: A connectionless server

The example can be used to create your our connectionless server design. This example uses UDP to create a connectionless socket server program. For information on the use of code examples, see the code disclaimer.

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

Example: A connectionless client

The following example shows how to use UDP to connect a connectionless socket client program to a server. For information on the use of code examples, see the code disclaimer.

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

Design applications with address families

The following topics provide sample programs that illustrate each of the socket address families:

- Use AF_INET address family
- Use AF_INET6 address family
- Use AF_UNIX address family
- Use AF_TELEPHONY address family
- Use AF_UNIX_CCSID address family

Use AF_INET address family

AF_INET address family sockets can be either connection-oriented (type SOCK_STREAM) or they can be connectionless (type SOCK_DGRAM). Connection-oriented AF_INET sockets use TCP as the transport protocol. Connectionless AF_INET sockets use UDP as the transport protocol. When you create an AF_INET domain socket, you specify AF_INET for the address family in the socket program. AF_INET sockets can also use a type of SOCK_RAW. If this type is set, the application connects directly to the IP layer and does not use either the TCP or UDP transports.

See Prerequisites for socket programming for details on setting up an environment to use the AF_INET address family.

For sample programs that use AF_INET address family, see Example: A connection-oriented server and Example: A connection-oriented client.

Use AF_INET6 address family

AF_INET6 sockets provide support for Internet Protocol version 6, (IPv6) 128 bit (16 byte) address structures. Programmers can write applications using AF_INET6 address family to accept client requests for either IPv4 or IPv6 nodes or from IPv6 nodes only.

Like AF_INET sockets, AF_INET6 sockets can be either connection-oriented (type SOCK_STREAM) or they can be connectionless (type SOCK_DGRAM). Connection-oriented AF_INET6 sockets use TCP as the transport protocol. Connectionless AF_INET6 sockets use UDP as the transport protocol. When you create an AF_INET6 domain socket, you specify AF_INET6 for the address family in the socket program. AF_INET6 sockets can also use a type of SOCK_RAW. If this type is set, the application connects directly to the IP layer and does not use either the TCP or UDP transports. See Prerequisites for socket programming for details on setting up an environment to use the AF_INET6 address family.

IPv6 applications compatibility with IPv4 applications

Socket applications written with AF_INET6 address family allow Internet Protocol version 6 (IPv6) applications to work with Internet Protocol version 4 (IPv4) applications (those applications that use

AF_INET address family). This feature allows socket programmers to use an IPv4-mapped IPv6 address format. This address format represents the IPv4 address of an IPv4 node to be represented as an IPv6 address. The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix 0:0:0:0:0:FFFF. For example, an IPv4- mapped address can look like this:

```
::FFFF:192.1.1.1
```

These addresses can be generated automatically by the **getaddrinfo()** function, when the specified host has only IPv4 addresses.

You can create applications that use AF_INET6 sockets to open TCP connections to IPv4 nodes. To accomplish this task, you can encode the destination's IPv4 address as an IPv4–mapped IPv6 address and pass that addresswithin a sockaddr_in6 structure in the **connect()** or **sendto()** call. When applications use AF_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the **accept()**, **recvfrom()**, or **getpeername()** calls using a sockaddr_in6 structure encoded this way.

While the **bind()** function allows applications to select the source IP address of UDP packets and TCP connections, applications often want the system to select the source address for them. Applications use in6addr_any similarly to the way they use the INADDR_ANY macro in IPv4 for this purpose. An additional feature of binding in this way is that it allows an AF_INET6 socket to communicate with both IPv4 and IPv6 nodes. For example, an application issuing an **accept()** on a listening socket bound to in6addr_any will accept connections from either IPv4 or IPv6 nodes. This behavior can be modified through the use of the IPPROTO_IPV6 level socket option IPV6_V6ONLY. Few applications will likely need to know which type of node with which they are interoperating. However, for those applications that do need to know, the IN6_IS_ADDR_V4MAPPED() macro defined in <netinet/in.h> is provided.

If you would like a more detailed comparison of IPv4 and IPv6, see Compare IPv4 to IPv6 in the Information Center. This information compares the features between these two protocols.

For examples programs and description of a situation where AF_INET6 socket communicates to both IPv4 and IPv6 nodes, see Create an application to accept IPv4 and IPv6 clients.

IPv6 Restrictions

Currently, OS/400 support of IPv6 is limited in some functionality in V5R2. The following table lists these restrictions and their implications to socket programmers.

Table 12. IPv6 restrictions and implications

Restriction	Implication
IPv6 does not support fragmentation.	AF_INET6 (SOCK_DGRAM) should not attempt to send datagrams larger than the MTU of the interface minus the size of the headers.
IPv6 anycast is not supported.	You cannot connect to or send to anycast addresses.
IPv6 multicast is not supported.	You cannot send or receive multicast datagrams.
iSeries host table does not support IPv6 addresses.	The **getaddrinfo()** and **getnameinfo()** APIs will not be able to locate IPv6 addresses in the host table. These APIs will find addresses in the DNS only.
The **gethostbyname()** and **gethostbyaddr()** APIs only support IPv4 address resolution.	Use **getaddrinfo()** and **getnameinfo()** APIs if IPv6 address resolution is required.

Use AF_UNIX address family

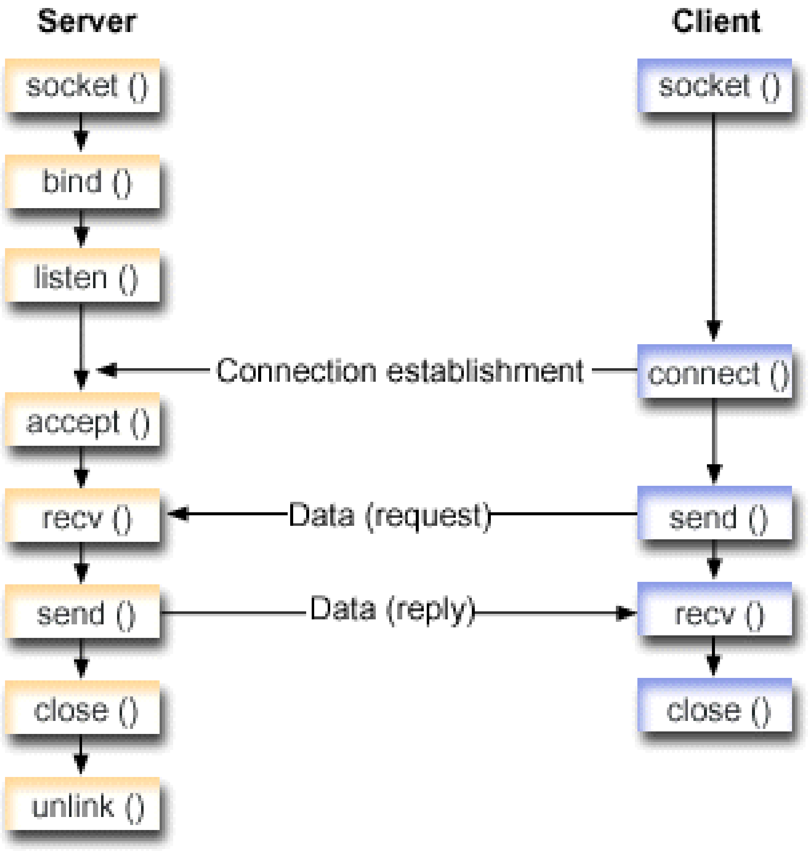
The AF_UNIX address family (sockets using the AF_UNIX or AF_UNIX_CCSID address families) can be connection-oriented (type SOCK_STREAM) or they can be connectionless (type SOCK_DGRAM). Both types are reliable because there are no external communication functions connecting the two processes.

UNIX domain datagram sockets act differently than UDP datagram sockets. With UDP datagram sockets, the client program does not have to call the **bind()** function because the system assigns an unused port number automatically. The server can then send a datagram back to that port number. However, with UNIX domain datagram sockets, the system does not automatically assign a path name for the client. Thus, all client programs using UNIX domain datagrams must call the **bind()** function. The exact path name specified on the client's **bind()** is what is passed to the server. Thus, if the client specifies a relative path name (that is, a path name that is not fully qualified by starting with /), the server cannot send the client a datagram unless it is running with the same current directory.

An example path name that an application might use for this address family is /tmp/myserver or servers/thatserver. With servers/thatserver, you have a path name that is not fully qualified (no / was specified). This means that the location of the entry in the file system hierarchy should be determined relative to the current working directory.

Note: Path names in the file system are NLS-enabled.

The following figure illustrates the client/server relationship of the AF_UNIX address family. See Prerequisites for socket programming for details on setting up an environment to use the AF_UNIX address family.



Socket flow of events: Server application that uses AF_UNIX address family
The Example: Server application that uses AF_UNIX address family uses the following sequence of function calls:

1. 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. The function returns a socket descriptor representing an endpoint. You can also use the **socketpair()** function to initialize a UNIX socket.

AF_UNIX or AF_UNIX_CCSID are the only address families to support the **socketpair()** function. The **socketpair()** function returns two socket descriptors that are unnamed and connected.

2. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.

   The name space for UNIX domain sockets consists of path names. When a sockets program calls the **bind()** function, an entry is created in the file system directory. If the path name already exists, the **bind()** fails. Thus, a UNIX domain socket program should always call an **unlink()** functions to remove the directory entry when it ends.
3. The **listen()** 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.
4. The **recv()** function receives data from the client application. 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 do not want our **recv()** to wake up until all 250 bytes of data have arrived.
5. The **send()** function echoes the data back to the client.
6. The **close()** function closes any open socket descriptors.
7. The **unlink()** function removes the UNIX path name from the file system.

Socket flow of events: Client application that uses AF_UNIX address family
The Example: Client application that uses AF_UNIX address family uses the following sequence of function calls:

1. 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. The function returns a socket descriptor representing an endpoint. You can also use the **socketpair()** function to initialize a UNIX socket.

   AF_UNIX or AF_UNIX_CCSID are the only address families to support the **socketpair()** function. The **socketpair()** function returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
3. The **send()** function sends 250 bytes of data specified which is specified in the server application with the SO_RCVLOWAT socket option.
4. The **recv()** function loops until all 250 bytes of the data until all 250 bytes have arrived.
5. The **close()** function closes any open socket descriptors.

Example: Server application that uses AF_UNIX address family: This example provides a sample server for the AF_UNIX address family. AF_UNIX address family uses many of the same socket calls as other address families, except it uses path name structure to identify the server application. The following sample programs use the AF_UNIX address family. For information on the use of code examples, see the code disclaimer.

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

Example: Client application that uses AF_UNIX address family: This example provides a sample client applicationfor the AF_UNIX address family. AF_UNIX address family uses many of the same socket calls as other address families, except it uses path name structure to identify the server application. The following sample program uses the AF_UNIX address family to create a client connection to a server. For information on the use of code examples, see the code disclaimer.

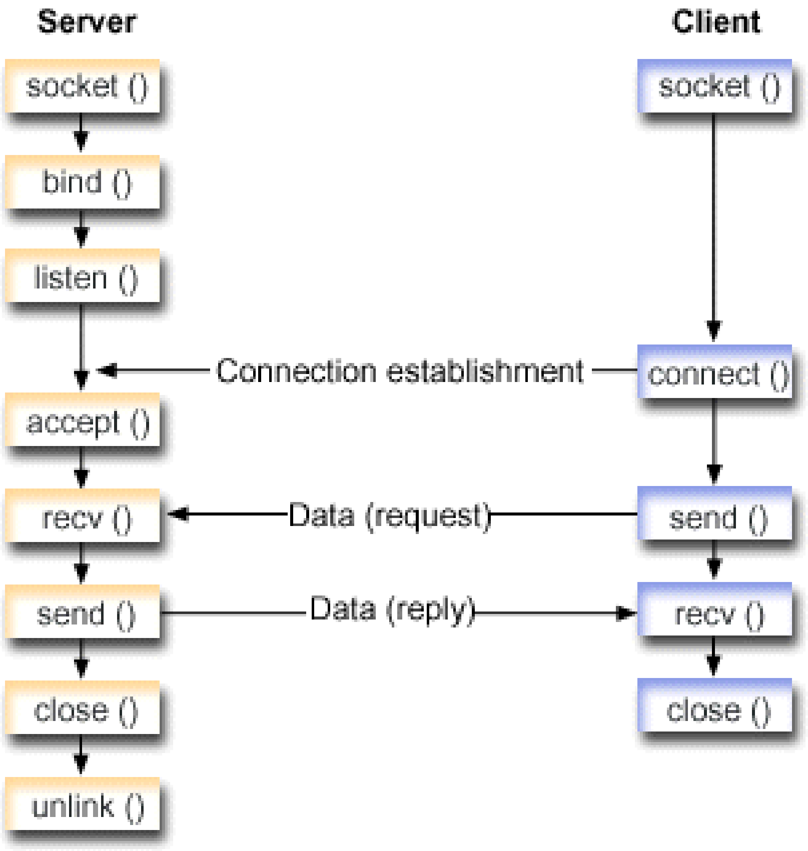
```
/**************************************************************************/
/* This sample program provides code for a client application that uses     */
/* AF_UNIX address family                                                 */
/**************************************************************************/
/**************************************************************************/
/* 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);
}
```

Use AF_UNIX_CCSID address family

AF_UNIX_CCSID address family socket have the same specifications as AF_UNIX address family sockets. They can be used for connection-oriented or connectionless and provide communications on the same system. See Use AF_UNIX address family for details.

Before working with AF_UNIX_CCSID socket application, you should be familiar with the **Qlg_Path_Name_T** structure to determine the output format. See Path name structures in the API Reference information in the Information Center for details.

When working with an output address structure, such as one returned from **accept()**, **getsockname()**, **getpeername()**, **recvfrom()**, and **recvmsg()**, the application must examine the socket address structure (sockaddr_unc) to determine its format. The sunc_format and sunc_qlg fields will determine the output format of the path name. But sockets will not necessarily use the same values on output as the application used on input addresses.



Socket flow of events: Server application that uses AF_UNIX_CCSID address family
The Example: Server application that uses AF_UNIX_CCSID address family uses the following sequence of function calls:

1. 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. You can also use the **socketpair()** function to initialize a UNIX socket.

   AF_UNIX or AF_UNIX_CCSID are the only address families to support the **socketpair()** function. The **socketpair()** function returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is created, the **bind()**function gets a unique name for the socket.

The name space for UNIX domain sockets consists of path names. When a sockets program calls the **bind()** function, an entry is created in the file system directory. If the path name already exists, the **bind()** fails. Thus, a UNIX domain socket program should always call an **unlink()** functions to remove the directory entry when it ends.

3. The **listen()** 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.
4. 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.
5. The **recv()** function receives data from the client application. 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 do not want our **recv()** to wake up until all 250 bytes of data have arrived.
6. The **send()** function echoes the data back to the client.
7. The **close()** function closes any open socket descriptors.
8. The **unlink()** function removes the UNIX path name from the file system.

Socket flow of events: Client application that uses AF_UNIX_CCSID address family
The Example: Client application that uses AF_UNIX_CCSID address family uses the following sequence of function calls:

1. 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. The function returns a socket descriptor representing an endpoint. You can also use the **socketpair()** function to initialize a UNIX socket.

   AF_UNIX or AF_UNIX_CCSID are the only address families to support the **socketpair()** function. The **socketpair()** function returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
3. The **send()** function sends 250 bytes of data specified which is specified in the server application with the SO_RCVLOWAT socket option.
4. The **recv()** function loops until all 250 bytes of the data until all 250 bytes have arrived.
5. The **close()** function closes any open socket descriptors.

Example: Server application that uses AF_UNIX_CCSID address family: The following sample programs use the AF_UNIX_CCSID address family. For information on the use of code examples, see the code disclaimer.

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

Example: Client application that uses AF_UNIX_CCSID address family: The following sample programs use the AF_UNIX_CCSID address family. For information on the use of code examples, see the code disclaimer.

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

Use AF_TELEPHONY address family

Telephony address family (sockets that use the AF_TELEPHONY address family) permit the user to initiate (dial) and complete (answer) telephone calls through an attached ISDN telephone network using standard socket APIs. The sockets forming the endpoints of a connection in this domain are really the called (passive endpoint) and calling (active endpoint) parties of a telephone call. The AF_TELEPHONY addresses are telephone numbers that consist of up to 40 digits (0 - 9) that are contained in sockaddr_tel address structures.

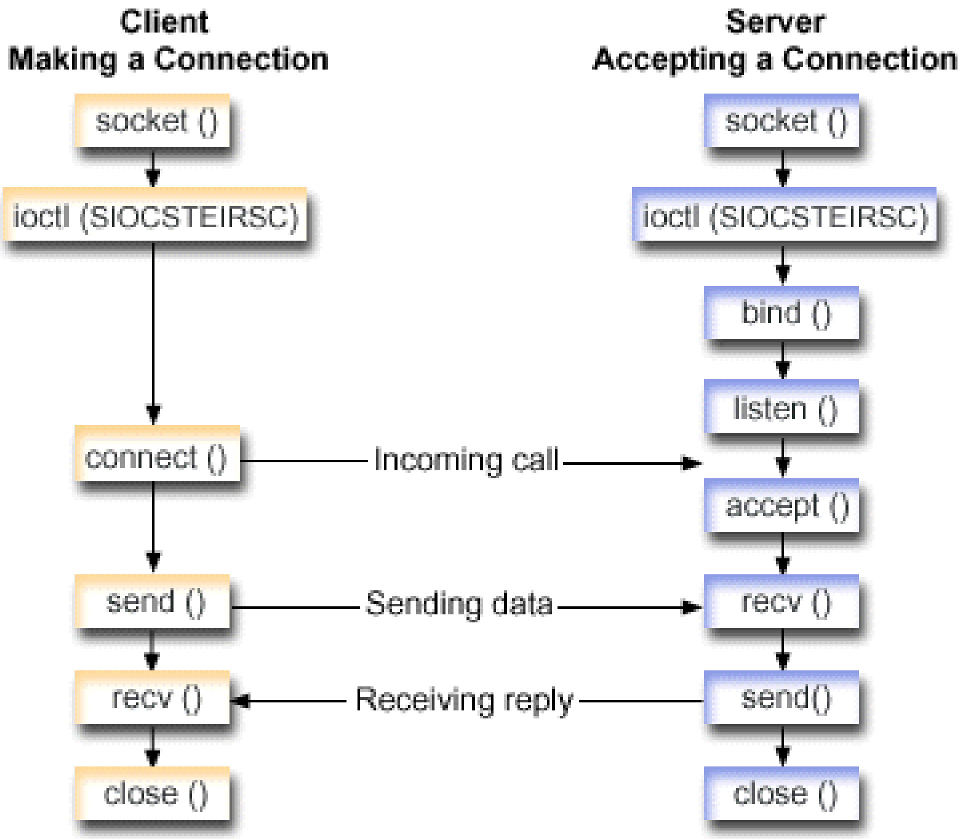
AF_TELEPHONY sockets are supported as connection-oriented (type SOCK_STREAM) sockets only. These sockets have semantics and function that are similar to that of other connection-oriented protocols. The major difference is that a connection in the telephony domain provides no more reliability than that of the underlying telephone connection. If guaranteed delivery is desired, it must be accomplished at the application level, such as in FAX applications using this family. Also, the idea of out-of-band data is not supported on telephony address family.

AF_TELEPHONY sockets must be associated with a network telephony device (logically, a telephone) before a connection can be initiated or completed. Use the special **ioctl()** command, **SIOCSTELRSC** (set telephony resources), to make this association. You must configure these devices and make them ready for use before issuing this command.

Before making an SIOCSTELRSC **ioctl()** call, the application must resolve the device name. The device name must be resolved to a system pointer, and you must use this pointer as input for the SIOCSTELRSC command.

The device will remain associated with the socket until the socket is closed. Finally, you can associate more than one device with a socket. This multiple association permits an application to listen for and answer calls on more than one device through a single socket.

The following figure illustrates the relationship of socket calls that are used with AF_TELEPHONY address family. See Prerequisites for socket programming for details on setting up an environment to use the AF_UNIX address family.



Socket flow of events: Client that uses AF_TELEPHONY address family

The Example: Make an AF_TELEPHONY connection uses the following sequence of function calls:

1. The **socket()** function returns a socket descriptor representing an endpoint.
2. You must associate the socket with a divice by resolving the device name to a system pointer and filling the structure for the request. Issue the **ioctl()** function to associate the device and the system pointer.
3. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
4. The **send()** function send the contents of the send buffer to the client.
5. The **recv()** function receives data from the client application
6. The **close()** function closes any open socket descriptors.

Socket flow of events: Server application that uses AF_TELEPHONY address family

The Example: Accept an AF_TELEPHONY connection uses the following sequence of function calls:

1. The **socket()** function returns a socket descriptor representing an endpoint.
2. You must associate the socket with a device by resolving the device name to a system pointer and filling the structure for the request. Issue the **ioctl()** function to associate the device and the system pointer.
3. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
4. The **listen()** allows the server to accept incoming client connections.
5. The server uses the **accept()** function to accept an incoming connection request.
6. The **recv()** function receives data from the client application
7. The **send()** function send the contents of the send buffer to the client.

8. The **close()** function closes any open socket descriptors.

Example: Make an AF_TELEPHONY connection: Programs can communicate with each other through a telephony domain socket. Use the following code to enable the socket to make the connection with a client. For information on the use of code examples, see the code disclaimer.

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

Example: Accept an AF_TELEPHONY connection: AF_TELEPHONY address family are used for application that use telephone numbers to identify the socket. This address family is primarily used in facsimile applications. Programs can communicate with each other through a telephony domain socket. Use the following code to enable the socket to accept a connection from the server. For information on the use of code examples, see the code disclaimer.

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

Socket concepts

The following topics discuss advanced socket concepts which go beyond a general discussion of what sockets are and how they work. They provide ways to design socket applications for larger and more complex networks. Each of the following concepts are linked to a corresponding sample program.

- Asynchronous I/O
- Secure sockets

- Client SOCKS support
- Thread safety
- Non-blocking I/O
- Signals
- IP multicasting
- File data transfer—send_file() and accept_and_recv()
- Out-of-band data
- I/O multiplexing—select()
- Socket network functions
- Domain name system (DNS) support
- BSD compatibility
- Pass descriptors between processes—sendmsg() and recvmsg()

Asynchronous I/O

Asynchronous I/O APIs provide a method for threaded client server models to perform highly concurrent and memory efficient I/O. In previous threaded client/server models, typically two I/O models have prevailed. The first model dedicated one thread per client connection. The first model consumes too many threads and could incur a substantial sleep and wake up cost. The second model minimizes the number of threads by issuing the **select()** API on a large set of client connections and delegating a readied client connection or request to a thread. In the second model, you must select or mark on each subsequent select, which can cause a substantial amount of redundant work.

Asynchronous I/O and overlapped I/O resolves both these dilemmas by passing data to and from user buffers after control has been returned to the user application. Asynchronous I/O notifies these worker threads when data is available to be read or when a connection has become ready to transmit data.

Asynchronous I/O advantages

- Uses system resources more efficiently.
  Data copies from and to user buffers are asynchronous to the application initiating the request. This overlapped processing makes efficient use of multiple processors and in many cases will improve paging rates because system buffers are freed for reuse when data arrives.
- Minimizes process/thread wait time.
- Provides immediate service to client requests.
- Lessons the sleep and wake up cost on average.
- Handles ″bursty application″ efficiently.
- Provides better scalability.
- Provides the most efficient method of handling large data transfers.
  The fillBuffer flag on the **QsoStartRecv()** API informs the operating system to acquire a large amount of data before completing the Asynchronous I/O. Large amounts of data can also be sent with one asynchronous operation.
- Minimizes the number of threads that are needed.
- Optionally may use timers to specify the maximum time allowed for this operation to complete asynchronously. Servers will close a client connection if it has been idle for a set amount of time. The asynchronous timers allow the server to enforce this time limit.
- Initiates secure session asynchronously with the **gsk_secure_soc_startInit()** API.

Table 13. Asynchronous I/O APIs

Function	Description

Table 13. Asynchronous I/O APIs (continued)

gsk_secure_soc_startInit()	Starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session. **Note:** This API only supports sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
gsk_secure_soc_startRecv()	Starts an asynchronous receive operation on a secure session. **Note:** This API only supports sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
gsk_secure_soc_startSend()	Start an asynchronous send operation on a secure session. **Note:** This API only supports sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
QsoCreateIOCompletionPort()	Create a common wait point for completed asynchronous overlapped I/O operations. The **QsoCreateIOCompletionPort()** function returns a port handle that represents the wait point. This handle is specified on the **QsoStartRecv()**, **QsoStartSend()**, **QsoStartAccept()**, **gsk_secure_soc_startRecv()**, or **gsk_secure_soc_startSend()** functions to initiate asynchronous overlapped I/O operations. Also this handle can be used with **QsoPostIOCompletion()** to post an event on the associated I/O completion port.
QsoDestroyIOCompletionPort()	Destroys an I/O completion port.
QsoWaitForIOCompletionPort()	Waits for completed overlapped I/O operation. The I/O completion port represents this wait point.
QsoStartAccept()	Starts an asynchronous accept operation. **Note:** This API only supports sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
QsoStartRecv()	Starts an asynchronous receive operation. **Note:** This API only supports sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
QsoStartSend()	Starts an asynchronous send operation. **Note:** This API only supports sockets with the AF_INET or AF_INET6 address families with the SOCK_STREAM socket type.
QsoPostIOCompletion()	Allows an application to notify a completion port that some function or activity has occurred.

How Asynchronous I/O works

An application will create an I/O completion port using the **QsoCreateIOCompletionPort()** API. This API will return a handle which can be used to schedule and wait for completion of asynchronous I/O requests. The application will start an input or an output function, specifying a I/O completion port handle. When the I/O completes, status information and an application-defined handle will be posted to the specified I/O completion port. The post to the I/O completion port will wake up exactly one of possibly many threads that are waiting. The application receives:

- a buffer that was supplied on the original request
- the length of data that was processed to or from that buffer
- a indication of what type of I/O operation has completed
- and application-defined handle that was passed on the initial I/O request

This application handle could simply be the socket descriptor identifying the client connection, or a pointer to storage that contains extensive information about the state of the client connection. Since the operation completed and the application handle was passed, the worker thread determines the next step to complete the client connection. Worker threads that process these completed asynchronous operations may handle many different client requests and are not tied to just one. Because copying to and from user buffers occurs asynchronously to the server processes, wait time for client request diminishes. This can be beneficial on systems where there are multiple processors.

For an example of a simple server model that uses asynchronous I/O, see Example: Use asynchronous I/O.

Secure sockets

Currently, OS/400 supports two methods of creating secure socket applications on the iSeries. The SSL_ APIs and Global Secure Toolkit (GSKit) APIs provide communications privacy over an open communications network, which in most cases is the Internet. These APIs allow client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. Both support server and client authentication and both allow an application to use Secure Sockets Layer (SSL) protocol. However, GSKit APIs are supported across all IBM eServer platforms, while the SSL_ APIs are native to the OS/400 operating system. To ensure interoperatability across platforms, it is recommended that you used GSKit APIs when developing applications for secure socket connections.

For descriptions of each of these APIs, see the following topics:

- Global Secure Toolkit (GSKit) APIs
- SSL_ APIs

The Secure socket API error code messages topic provides a list of common error code messages that can occur with secure socket APIs.

Overview of secure sockets

Originally developed by Netscape, Secure Sockets Layer (SSL) protocol is a layered protocol that is intended to be used on top of a reliable transport such as Transmission Control Protocol (TCP) to provide secure communications for an application. A few of the many applications that require secure communications are HTTPs, FTPs, SMTP, and TELNETs.

An SSL-enabled application usually needs to use a different port than an application that is not SSL-enabled. For example, an SSL-enabled browser accesses an SSL-enabled Hypertext Transfer Protocol (HTTP) server with a Universal Resource Locator (URL) that begins ″HTTPs″ rather than ″HTTP.″ In most cases, a URL of ″HTTPs″ attempts to open a connection to port 443 of the server system instead of to port 80 that the standard HTTP server uses.

There are multiple versions of the SSL protocol defined. The latest version, Transport Layer Security (TLS) Version 1.0, provides an evolutionary upgrade from SSL Version 3.0. Both iSeries-native SSL_ APIs and the GSKit APIs support TLS Version 1.0, TLS Version 1.0 with SSL Version 3.0 compatibility, SSL Version 3.0, SSL Version 2.0, and SSL Version 3.0 with 2.0 compatibility. For more details on TLS Version 1.0, see Internet Engineering Task Force (IETF) RFC 2246, ″Transport Layer Security″.

Global Secure ToolKit (GSKit) APIs

Global Secure ToolKit (GSKit) is a set of programmable interfaces that allow an application to be SSL enabled. Just like the SSL_ APIs, GSKit APIs allow you to access SSL and TLS functions from your socket application program. However, GSKit APIs are supported across IBM eServer platforms and are easier to program in then the previous SSL_ APIs. In addition, a new GSKit API has been added to create an asynchronous instance of a secure sockets session. This API provides a secure connection for handling multiple clients or if the number of incoming requests are high and require multiple jobs. However this API is native to the OS/400 and can not be ported to other @server platforms.

Note: These APIs only support sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

The following table describes the GSKit APIs:

Table 14. Global secure toolkit APIs

Function	Description
gsk_attribute_get_buffer()	Obtains specific character string information about a secure session or an SSL environment, such as certificate store file, certificate store password, application ID, and ciphers.
gsk_attribute_get_cert_info()	Obtains specific information about either the server or client certificate for a secure session or an SSL environment.
gsk_attribute_get_enum_value()	Obtains values for specific enumerated data for a secure session or an SSL environment.
gsk_attribute_get_numeric_value()	Obtains specific numeric information about a secure session or an SSL environment.
gsk_attribute_set_buffer()	Sets a specified buffer attribute to a value inside the specified secure session or an SSL environment.
gsk_attribute_set_enum()	Sets a specified enumerated type attribute to an enumerated value in the secure session or SSL environment.
gsk_attribute_set_numeric_value()	Sets specific numeric information for a secure session or an SSL environment.
gsk_environment_close()	Closes the SSL environment and releases all storage associated with the environment.
gsk_environment_init()	Initializes the SSL environment after any required attributes are set.
gsk_environment_open()	Returns an SSL environment handle that must be saved and used on subsequent gsk calls.
gsk_secure_soc_close()	Closes a secure session and free all the associated resources for that secure session.
gsk_secure_soc_init()	Negotiates a secure session, using the attributes set for the SSL environment and the secure session.
gsk_secure_soc_misc()	Performs miscellaneous functions for a secure session.
gsk_secure_soc_open()	Obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related function calls.
gsk_secure_soc_read()	Receives data from a secure session.
gsk_secure_soc_startInit()	Starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session.
gsk_secure_soc_write()	Writes data on a secure session.
gsk_secure_soc_startRecv()	Initiates an asynchronous receive operation on a secure session.
gsk_secure_soc_startSend()	Initiates an asynchronous send operation on a secure session.

Table 14. Global secure toolkit APIs (continued)

gsk_strerror()	Retrieves an error message and associated text string which describes a return value that was returned from calling a GSK API.

An application that uses the sockets and GSKit APIs contains the following elements:

1. A call to **socket()** to obtain a socket descriptor.
2. A call to **gsk_environment_open()** to obtain a handle to an SSL environment.
3. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the SSL environment. At a minimum, either a call to **gsk_attribute_set_buffer()** to set the GSK_OS400_APPLICATION_ID value or to set the GSK_KEYRING_FILE value. Only one of these should be set. It is prefered that you use the GSK_OS400_APPLICATION_ID value. Also ensure you set the type of application (client or server), GSK_SESSION_TYPE, using **gsk_attribute_set_enum()**.
4. A call to **gsk_environment_init()** to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that will run using this environment.
5. Socket calls to activate a connection. It calls **connect()** to activate a connection for a client program, or it calls **bind()**, **listen()**, and **accept()** to enable a server to accept incoming connection requests.
6. A call to **gsk_secure_soc_open()** to obtain a handle to a secure session.
7. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the secure session. At a minimum, a call to **gsk_attribute_set_numeric_value()** to associate a specific socket with this secure session.
8. A call to **gsk_secure_soc_init()** to initiate the SSL handshake negotiation of the cryptographic parameters.

   Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server which the client is connecting to has enabled client authentication. The **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) or **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API calls identify (though in dissimilar ways) the key database file from which the certificate and private key that are used during the handshake are obtained.
9. Calls to **gsk_secure_soc_read()** and **gsk_secure_soc_write()** to receive and send data.
10. A call to **gsk_secure_soc_close()** to end the secure session.
11. A call to **gsk_environment_close()** to close the SSL environment.
12. A call to **close()** to destroy the connected socket.

See these examples for sample programs that use the GSKit APIs:

- Example: Establish a secure server with asynchronous data receive
- Example: Establish a secure server that uses asynchronous handshake
- Example: Establish a secure client with Global Secure ToolKit (GSKit) APIs

SSL_ APIs

The SSL_ APIs provide allow programmers to create secure socket applications on iSeries. Unlike GSKit APIs, SSL_ APIs are native to the OS/400 system only. The following table describes nine SSL_ APIs that are supported in the OS/400 implementation. Use the links to learn details about the individual APIs that are listed in the API information in the Information Center.

Table 15. SSL_ APIs

Function	Description
SSL_Create()	Enable SSL support for the specified socket descriptor.

Table 15. SSL_ APIs (continued)

SSL_Destroy()	End SSL support for the specified SSL session and socket.
SSL_Handshake()	Initiate the SSL handshake protocol.
SSL_Init()	Initialize the current job for SSL and establish the SSL security information for the current job. **Note:** Either an **SSL_Init()** or **SSL_Init_Application()** API must be executed in a process before SSL can be used.
SSL_Init_Application()	Initialize the current job for SSL and establish the SSL security information for the current job. **Note:** Either an **SSL_Init()** or **SSL_Init_Application()** API must be executed in a process before SSL can be used.
SSL_Read()	Receive data from an SSL-enabled socket descriptor.
SSL_Write()	Write data to an SSL-enabled socket descriptor.
SSL_Strerror()	Retrieve SSL runtime error message.
SSL_Perror()	Print SSL error message.

An application that uses the sockets and SSL_ APIs contains the following elements:

- A call to **socket()** to obtain a socket descriptor.
- Either call **SSL_Init()** or **SSL_Init_Application()** to initialize the job environment for SSL processing and to establish the SSL security information for all SSL sessions that will run in the current job. Only one of these APIs should be used. It is prefered that you use the **SSL_Init_Application()** API.
- Socket calls to activate a connection. It calls **connect()** to activate a connection for a client program, or it calls **bind()**, **listen()**, and **accept()** to enable a server to accept incoming connection requests.
- A call to **SSL_Create()** to enable SSL support for the connected socket.
- A call to **SSL_Handshake()** to initiate the SSL handshake negotiation of the cryptographic parameters.

  Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server which the client is connecting to has enabled client authentication. The **SSL_Init()** or **SSL_Init_Application()** APIs identify (though in dissimilar ways) the key database file from which the certificate and private key that are used during the handshake are obtained.
- Calls to **SSL_Read()** and **SSL_Write()** to receive and send data.
- A call to **SSL_Destroy()** to disable SSL support for the socket.
- A call to **close()** to destroy the connected sockets.

For sample programs that use these SSL_ APIs, see these sample programs:

- Example: Establish a secure server with SSL_ APIs
- Example: Establish a secure client with SSL_ APIs

Secure socket API error code messages

To access information on the following secure socket error code messages, complete the following:

1. On a command line, type

   ```
   DSPMSGD RANGE(XXXXXXX)
   ```

   where XXXXXXX is message ID for the return code. For example if the return code was 3, you would enter

```
DSPMSGD RANGE(CPDBCB9)
```

2. Select **1** to display message text.

Table 16. Secure Socket API Error Code Messages

Return Code	Message ID	Constant Name
0	CPCBC80	GSK_OK
4	CPCBC80	GSK_INSUFFICIENT_STORAGE
502	CPE3406	GSK_WOULD_BLOCK
1	CPDBCA1	GSK_INVALID_HANDLE
2	CPDBCB3	GSK_API_NOT_AVAILABLE
3	CPDBCB9	GSK_INTERNAL_ERROR
5	CPDBC95	GSK_INVALID_STATE
107	CPDBC98	GSK_KEYFILE_CERT_EXPIRED
201	CPDBCA4	GSK_NO_KEYFILE_PASSWORD
202	CPDBCB5	GSK_KEYRING_OPEN_ERROR
301	CPDBCA5	GSK_CLOSE_FAILED
402	CPDBC81	GSK_ERROR_NO_CIPHERS
403	CPDBC82	GSK_ERROR_NO_CERTIFICATE
404	CPDBC84	GSK_ERROR_BAD_CERTIFICATE
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

Table 16. Secure Socket API Error Code Messages (continued)

6002	CPDBCC9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPDBCAD	GSK_OS400_ERROR_NO_ACCESS
6004	CPDBCB8	GSK_OS400_ERROR_CLOSED
6005	CPDBCCB	GSK_OS400_ERROR_NO_CERTIFICATE_AUTHORITIES
6007	CPDBCB4	GSK_OS400_ERROR_NO_INITIALIZE
6008	CPDBCAE	GSK_OS400_ERROR_ALREADY_SECURE
6009	CPDBCAF	GSK_OS400_ERROR_NOT_TCP
6010	CPDBC9C	GSK_OS400_ERROR_INVALID_POINTER
6011	CPDBC9B	GSK_OS400_ERROR_TIMED_OUT
6012	CPCBCBA	GSK_OS400_ASYNCHRONOUS_RECV
6013	CPCBCBB	GSK_OS400_ASYNCHRONOUS_SEND
6014	CPDBCBC	GSK_OS400_ERROR_INVALID_OVERLAPPEDIO_T
6015	CPDBCBD	GSK_OS400_ERROR_INVALID_IOCOMPLETIONPORT
6016	CPDBCBE	GSK_OS400_ERROR_BAD_SOCKET_DESCRIPTOR
6017	CPDBCBF	GSK_OS400_ERROR_CERTIFICATE_REVOKED
6018	CPDBC87	GSK_OS400_ERROR_CRL_INVALID
6019	CPCBC88	GSK_OS400_ASYNCHRONOUS_SOC_INIT
0	CPCBC80	Successful return
-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 (map to -4)
-16	CPDBC8B	SSL_ERROR_BAD_PEER (map to -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

Table 16. Secure Socket API Error Code Messages (continued)

-99	CPDBCB9	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPDBCD8	SSL_ERROR_NO_REUSE

Client SOCKS support

iSeries uses SOCKS version 4 to enable programs that use AF_INET address family with SOCK_STREAM socket type to communicate with server programs that run on systems outside a firewall. A firewall is a very secure host that a network administrator places between a secure internal network and a less secure external network. Typically such a network configuration does not allow communications that originate from the secure host to be routed on the less secure network, and vice-versa. Proxy servers that exist on the firewall help manage required access between secure hosts and less secure networks.

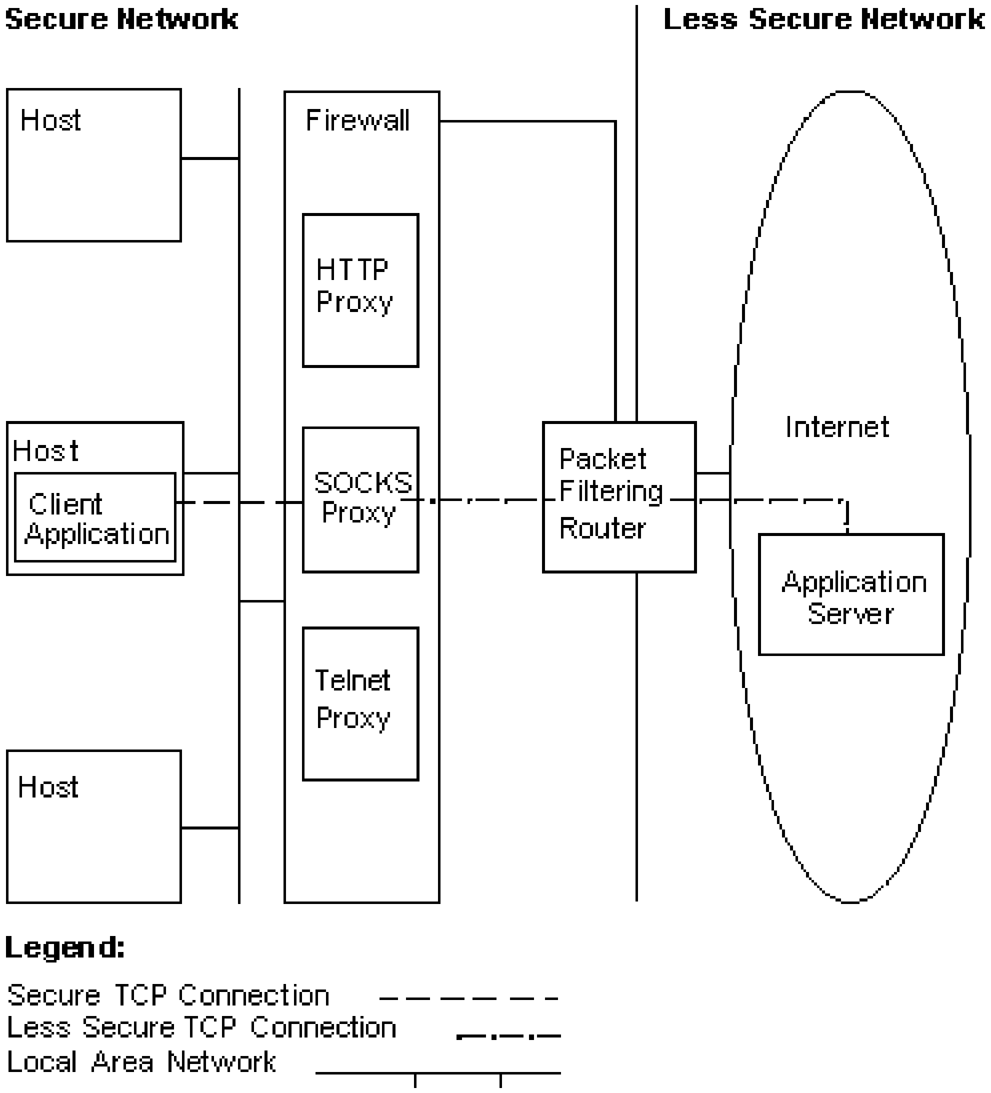
Applications that run on hosts in a secure internal network must send their requests to firewall proxy servers to navigate the firewall. The proxy servers can then forward these requests to the real server on the less secure network and relay the reply back to the applications on the originating host. A common example of a proxy server is an HTTP proxy server. Proxy servers perform a number of tasks for HTTP clients:

- They hide your internal network from outside systems.
- They protect the host from direct access by outside systems.
- They can filter data that comes in from outside if they are properly designed and configured.

HTTP proxy servers handle only HTTP clients.

A common alternative to running multiple proxy servers on a firewall is to run a more robust proxy server known as a SOCKS server. A SOCKS server can act as a proxy for any TCP client connection that is established using the sockets API. The key advantage to iSeries Client SOCKS support is that it enables client applications to access a SOCKS server transparently without changing any client code.

The figure below shows a common firewall arrangement with an HTTP proxy, a telnet proxy, and a SOCKS proxy on the firewall. Notice that the two separate TCP connections used for the secure client that is accessing a server on the internet. One connection leads from the secure host to the SOCKS server, and the other leads from the less secure network to the SOCKS server.



Two actions are required on the secure client host to use a SOCKS server:

1. Configuration of a SOCKS server. On February 15, 2000, IBM announced that the IBM Firewall for iSeries product (5769-FW1), which provides SOCKS server support, will not be enhanced beyond its current V4R4 capability.
2. On the secure client system, define all outbound Client TCP connections that are to be directed to the SOCKS server on the Client system. You can define the secure client SOCKS configuration entries by using the SOCKS tab found under the iSeries Navigator function of iSeries Access 95 or Windows NT®. The SOCKS tab has substantial help on configuring the secure client system for Client SOCKS support.

   To configure client SOCKS support, do the following:

   a. In iSeries Navigator, expand your **iSeries server --> Network --> TCP/IP Configuration**.
   b. Right-click **TCP/IP Configuration**.
   c. Click **Properties**.
   d. Click the **SOCKS** tab.
   e. Enter your connection information on the SOCKS page.

Note: The secure client SOCKS configuration data is saved in the file QASOSCFG in library QUSRSYS on the secure client host system.

Once configured, the system automatically directs certain outbound connections to the SOCKS server you specified on the SOCKS page. You do not need to make any changes to the secure client application. When it receives the request, the SOCKS server establishes a separate external TCP/IP connection to the server in the less secure network. The SOCKS server then relays data between the internal and external TCP/IP connections.

Note: The remote host on the less secure network connects directly to the SOCKS server. It does not have direct access to the secure client.

Up to this point we have addressed ″outbound″ TCP connections that originate from the secure client. Client SOCKS support also lets you tell the SOCKS server to allow an inbound connection request across a firewall. An **Rbind()** call from the secure client system allows this communication. For **Rbind()** to operate, the secure client must have previously issued a **connect()** call and the call must have resulted in an outbound connection over the SOCKS server. The **Rbind()** inbound connection must be from the same IP address that was addressed by the outbound connection that the **connect()** established.

The following figure shows a detailed overview of how sockets functions interact with a SOCKS server transparent to the application. In the example, the FTP client calls the **Rbind()** function instead of a **bind()** function[1]. It makes this call by recompiling the FTP client code with the __Rbind preprocessor #define, which defines **bind()** to be **Rbind()**. Alternatively, an application can explicitly code **Rbind()** in the pertinent source code. If an application does not require inbound connections across a SOCKS server, **Rbind()** should not be used.

Notes:

1. The FTP client uses **Rbind()** because the FTP protocol allows the FTP server to establish a data connection because of a request from the FTP client to send files or data.
2. The SOCKS server establishes a data connection with the FTP client and relays data between the FTP client and the FTP server. Many SOCKS servers allow a fixed length of time for the server to connect to the secure client. If a the server does not connect within this time, errno ECONNABORTED will be encountered on the **accept()**.
3. FTP client initiates an outbound TCP connection to a less secure network through a SOCKS server. The destination address that the FTP client specifies on the connect is the IP address and port of the FTP server located on the less secure network. The secure host system has been configured through the SOCKS page to direct this connection through the SOCKS server. Once configured, the system automatically directs the connection to the SOCKS server that was specified through the SOCKS page.
4. A socket is opened and **Rbind()** is called in order to establish an inbound TCP connection. Once established, this inbound connection is from the same destination-outbound IP address that was specified above. Outbound and inbound connections over the SOCKS server must be paired for a particular thread. In other words, all **Rbind()** inbound connections should immediately follow the outbound connection over the SOCKS server and no intervening non-SOCKS connections relating to this thread can be attempted before the **Rbind()** runs.
5. **getsockname()** returns the SOCKS server address. The socket is logically bound to the SOCKS server IP address coupled with a port that is selected through the SOCKS server. In this example, the address is sent through the ″control connection″ Socket CTLed to the FTP server that is located on the less secure network. This is the address to which the FTP server connects. The FTP server connects to the SOCKS server and not directly to the secure host.
6. The SOCKS server establishes a data connection with the FTP client and relays data between the FTP client and the FTP server. Many SOCKS servers allow a fixed length of time for the server to connect to the secure client. If a server does not connect within this time, errno ECONNABORTED will be encountered on the **accept()**.

Thread safety

A function is considered threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe if and only if all the functions it calls are threadsafe also. Socket APIs are comprised of system and network functions, which are both thread safe.

All network functions with names that end in ″_r″ have similar semantics and are also threadsafe. For a sample program that uses thread safe Socket APIs, see Example: Use gethostbyaddr_r() for threadsafe network routines.

The other resolver routines are threadsafe with each other but they use the _res data structure. This data structure is shared between all threads in a process and can be changed by an application during a resolver call. For a sample program that uses resolver routines, see Example: Update and query DNS.

Non-blocking I/O

When an application issues one of the socket input functions, and there is no data to read, the function blocks, and does not return until there is data to read. Similarly, an application can block on a socket output function when data cannot be sent immediately. Finally, **connect()** and **accept()** can block while waiting for connection establishment with the partner's programs.

Sockets provides a method that enables application programs to issue functions that block so that the function returns without delay. This is done by either calling **fcntl()** to turn on the **O_NONBLOCK** flag, or calling **ioctl()** to turn on the **FIONBIO** flag. Once running in this non-blocking mode, if a function cannot be completed without blocking, it returns immediately. A **connect()** may return with **[EINPROGRESS]**, which means that the connection initiation has been started. You can then use **select()** to determine when the connection has completed. For all other functions that are affected by running in the non-blocking mode, an error code of **[EWOULDBLOCK]** indicates that the call was unsuccessful.

You can use non-blocking with the following socket functions:

- **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()**

For a sample program that uses non-blocking I/O, see Example: Non-blocking I/O and select().

Signals

An application program can request to be notified asynchronously (request that the system send a **signal**) when a condition that the application is interested in occurs. There are two asynchronous signals that sockets will send to an application.

1. **SIGURG** is a signal that will be sent when out-of-band (OOB) data is received on a socket for which the concept of OOB data is supported. For example, a socket with an address family of AF_INET and a type of SOCK_STREAM can be conditioned to send a SIGURG signal.
2. **SIGIO** is a signal that will be sent when normal data, OOB data, error conditions, or just about anything happens on any type of socket.

The application should ensure that it is able to handle receiving a signal before it requests the system to send signals. This is done by setting up **signal handlers**. One way to set a signal handler is by issuing the **sigaction()** call.

An application requests the system to send the **SIGURG** signal by one of the following methods:

- Issuing a **fcntl()** call and specifying a process ID or a process group ID with the F_SETOWN command.
- Issuing an **ioctl()** call and specifying the FIOSETOWN or the SIOCSPGRP command (request).

An application requests the system to send the **SIGIO** signal in two steps. First it must set the process ID or the process group ID as described above for the **SIGURG** signal. This is to inform the system of where the application wants the signal to be delivered. Second, the application must do either of the following:

- Issue the **fcntl()** call and specify the F_SETFL command with the FASYNC flag.
- Issue the **ioctl()** call and specify the FIOASYNC command.

This step requests the system to generate the SIGIO signal. Note that these steps can be done in any order. Also note that if an application issues these requests on a listening socket, the values set by the requests are inherited by all sockets that are returned to the application from the **accept()** function. That is, newly accepted sockets will also have the same process ID or process group ID as well as the same information with regard to sending the SIGIO signal.

A socket can also generate synchronous signals on error conditions. Whenever an application receives [EPIPE] an **errno** on a socket function, a SIGPIPE signal is delivered to the process that issued the operation receiving the **errno** value. On a BSD implementation, by default the SIGPIPE signal ends the process that received the **errno** value. To remain compatible with previous releases of the OS/400 implementation, the OS/400 implementation uses a default behavior of ignore for the SIGPIPE signal. This ensures that existing applications will not be negatively affected by the addition of the signals function.

When a signal is delivered to a process that is blocked on a sockets function, the function returns from the wait with the **[EINTR] errno** value, allowing the application's signal handler to run. The functions for which this will occur are:

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

- **write()**
- **writev()**

It is important to note that signals do not provide the application program with a socket descriptor that identifies where the condition being signalled actually exists. Thus, if the application program is using multiple socket descriptors, it will have to either poll the descriptors or use the **select()** call to determine why the signal was received.

For a sample program that uses signals, see Example: Use signals with blocking socket APIs.

IP multicasting

IP multicasting allows an application to send a single IP datagram that a group of hosts in a network can receive. The hosts that are in the group may reside on a single subnet or may be on different subnets that multicast capable routers connect. Hosts may join and leave groups at any time. There are no restrictions on the location or number of members in a host group. A class D Internet address in the range 224.0.0.1 to 239.255.255.255 identifies a host group.

You can currently use IP multicasting with AF_INET address family only.

An application program can send or receive multicast datagrams using the Sockets API and connectionless, SOCK_DGRAM type sockets. Multicasting is a one-to-many transmission method. Connection-oriented sockets of type SOCK_STREAM cannot be used for multicasting. When a socket of type SOCK_DGRAM is created, an application can use the **setsockopt()** function to control the multicast characteristics associated with that socket. The **setsockopt()** function accepts the following IPPROTO_IP level flags:

- IP_ADD_MEMBERSHIP: Joins the multicast group specified
- IP_DROP_MEMBERSHIP: Leaves the multicast group specified
- IP_MULTICAST_IF: Sets the interface over which outgoing multicast datagrams should be sent
- IP_MULTICAST_TTL: Sets the Time To Live (TTL) in the IP header for outgoing multicast datagrams
- IP_MULTICAST_LOOP: Specifies whether or not a copy of an outgoing multicast datagram should be delivered to the sending host as long as it is a member of the multicast group

For examples of IP multicasting, see the following examplesExample: Use multicasting.

File data transfer—send_file() and accept_and_recv()

OS/400 Socket provide the **send_file()** and **accept_and_recv()**APIs that enable faster and easier file transfers over connected sockets. These two APIs are especially useful for file-serving applications such as Hypertext Transfer Protocol (HTTP) servers.

The **send_file()** enables the sending of file data directly from a file system over a connected socket with a single API call.

The **accept_and_recv()** is a combination of three socket functions: **accept()**, **getsockname()**, and **recv()**.

For sample programs for the **send_file()** and **accept_and_recv()** APIs, see Example: Transfer file data using send_file() and accept_and_recv().

Out-of-band data

Out-of-band (OOB) data is user-specific data that only has meaning for connection-oriented (stream) sockets. Stream data is generally received in the same order it is sent. OOB data is received independent of its position in the stream (independent of the order in which it was sent). This is possible because the data is marked in such a way that, when it is sent from program A to program B, program B is notified of its arrival.

OOB data is supported on AF_INET (SOCK_STREAM) and AF_INET6 (SOCK_STREAM) only.

OOB data is sent by specifying the MSG_OOB flag on the **send()**, **sendto()**, and **sendmsg()** functions.

The transmission of OOB data is the same as the transmission of regular data. It is sent after any data that is buffered. In other words, OOB data does not take precedence over any data that may be buffered; data is transmitted in the order that it was sent.

On the receiving side, things are a little more complex:

- The sockets API keeps track of OOB data that is received on a system by using an OOB marker. The OOB marker points to the last byte in the OOB data that was sent.

  Note: The value that indicates which byte the OOB marker points to is set on a system basis (all applications use that value). This value must be consistent between the local and remote ends of a TCP connection. Socket applications that use this value must use it consistently between the client and server applications. To change which byte the OOB marker points to, see Change TCP Attributes (CHGTCPA) command in the Information Center.

  The **SIOCATMARK ioctl()** request determines if the read pointer is pointing to the last OOB byte.

  Note: If multiple occurrences of OOB data are sent, the OOB marker points to the last OOB byte of the final OOB data occurrence.
- Independent of whether OOB data is received in-line or not, an input operation processes data up to the OOB marker, if OOB data was sent.
- A **recv()**, **recvmsg()**, or **recvfrom()** function (with the MSG_OOB flag set) is used to receive OOB data. An error of [EINVAL] is returned if one of the receive functions has completed and one of the following occurs:
  - The socket option SO_OOBINLINE is not set and there is no OOB data to receive.
  - The socket option SO_OOBINLINE is set.

  If the socket option SO_OOBINLINE is not set, and the sending program sent OOB data with size greater than one byte, all the bytes but the last are considered normal data. (Normal data means that the receiving program can receive data without specifying the MSG_OOB flag.) The last byte of the OOB data that was sent is not stored in the normal data stream. This byte can only be retrieved by issuing a **recv()**, **recvmsg()**, or **recvfrom()** function with the MSG_OOB flag set. If a receive is issued with the MSG_OOB flag not set, and normal data is received, the OOB byte is deleted. Also, if multiple occurrences of OOB data are sent, the OOB data from the preceding occurrence is lost, and the position of the OOB data of the final OOB data occurrence is remembered.

  If the socket option SO_OOBINLINE is set, then all of the OOB data that was sent is stored in the normal data stream. Data can be retrieved by issuing one of the three receive functions without specifying the MSG_OOB flag (if it is specified, an error of [EINVAL] is returned). OOB data is not lost if multiple occurrences of OOB data are sent.
- OOB data is not discarded if SO_OOBINLINE is not set, OOB data has been received, and the user then sets SO_OOBINLINE on. The initial OOB byte is considered normal data.
- If SO_OOBINLINE is not set, OOB data was sent, and the receiving program issued an input function to receive the OOB data, then the OOB marker is still valid. The receiving program can still check if the read pointer is at the OOB marker, even though the OOB byte was received.

I/O multiplexing—select()

Because Asynchronous I/O provides a more efficient way to maximize your application resources, we recommend using the Asynchronous I/O APIs rather than the **select()** API. However, your specific application design may allow for **select()** to be used. Like Asynchronous I/O, **select()** creates a common point to wait on multiple conditions at the same time. However, **select()** allows an application to specify sets of descriptors to do the following:

- To see if there is data to be read.

- To see if data can be written.
- To see if an exception condition is present.

The descriptors that can be specified in each set can be socket descriptors, file descriptors, or any other object that is represented by a descriptor.

The **select()** function also allows the application to specify if it wants to wait for data to become available. The application can specify how long to wait. See Example: Non-blocking I/O and select() for a sample program.

Socket network functions

Socket network functions allow application programs to obtain information from the host, protocol, service, and network files. The information can be accessed by name or by address, or by sequential access of the file. These network functions (or routines) are required when setting up communications between programs that run across networks, and thus are not used by AF_UNIX sockets. For a brief summary of the individual these network functions routines, see Sockets Network Functions (Routines) in the API reference topic in the Information Center.

The routines do the following:
- Map host names to network addresses.
- Map network names to network numbers.
- Map protocol names to protocol numbers.
- Map service names to port numbers.
- Convert the byte order of Internet network addresses.
- Convert Internet address and dotted decimal notation.

Included in the network routines is a group of routines called resolver routines. These routines make, send, and interpret packets for name servers in the Internet domain and are also used to do name resolution. The resolver routines normally get called by **gethostbyname()**, **gethostbyaddr()**, **getnameinfo()**, and **getaddrinfo()** but can be called directly. For examples that use these resolver routines, see Example: Use gethostbyaddr_r() for threadsafe network routines. Primarily resolver routines are used for accessing domain name system (DNS) through socket application. See Domain name service support for details on how sockets can be used with DNS.

Domain name system (DNS) support

iSeries provides applications with access to the domain name system (DNS) through the resolver functions. The DNS has the following three major components:
- **Domain name space and resource records**
  Specifications for a tree-structured name space and the data associated with the names.
- **Name servers**
  Server programs that hold information about the domain tree structure and set information. For more information on name servers, see the DNS topic in the Information Center.
- **Resolvers**
  Programs that extract information from name servers in response to client requests.

The resolvers provided in the OS/400 implementation are socket functions that provide communication with a name server. These routines are used to make, send, update, and interpret packets, and perform name caching for performance. They also provide function for ASCII to EBCDIC and EBCDIC to ASCII conversion. Optionally, the resolver will use transaction signatures (TSIG) to securely communicate with the DNS. For a brief summary of the individual resolver routines, see Sockets Network Functions (Routines) in the API reference topic in the Information Center. This link also provides information on the _res structure. The _res structure contains global information that are used by these routines.

For more information on domain names, see the following RFC, which you can locate from the RFC search page.

- 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″

For more information on other ways to use DNS with socket applications, see the following topics:

- Environment variables
  This topic describes the environment variables that can be used for name resolution.
- Data caching
  This topic provides details on using sockets to cache responses to DNS queries to lessen the amount of traffic on a network. Example: Update and query DNS provides a sample program that shows how DNS records can be queried and updated with socket APIs.

Environment variables

You can use enviroment variables to override default initialization of resolver functions. Environment variables are only checked after a successful call to **res_init()** or **res_ninit()**. So if the structure has been manually initialized, environment variables are ignored. Also note that the structure is only initialized once so later changes to the environment variables will be ignored.

Note: The name of the environment variable must be uppercased. The string value may be mixed case. Japanese systems using CCSID 290 should use uppercase characters and numbers only in both environment variables names and values. The list contains the descriptions of environment variables that can be used with the **res_init()** and **res_ninit()** APIs.

LOCALDOMAIN
Set this environment variable to a space-separated list of up to six search domains with a total of 256 characters (including spaces). This will override the configured search list (struct state.defdname and struct state.dnsrch). If a search list is specified, the default local domain is not used on queries.

RES_OPTIONS
Allows certain internal resolver variables to be modified. The environment variable can be set to one or more of the following space-separated options:

- **NDOTS:n**
  Sets a threshold for the number of dots which must appear in a name given to **res_query()** before an initial absolute query will be made. The default for n is ``1'', meaning that if there are any dots in a name, the name will be tried first as an absolute name before any search list elements are appended to it.
- **TIMEOUT:n**
  Sets the amount of time (in seconds) the resolver will wait for a response from a remote name server before giving up and retrying the query.
- **ATTEMPTS:n**
  Sets the number of queries the resolver will send to a given nameserver before giving up and trying the next listed nameserver.
- **ROTATE**
  Sets RES_ROTATE in _res.options, which rotates the selection of nameservers from among those listed. This has the effect of spreading the query load among all listed servers, rather than having all clients try the first listed server first every time.

- **NO-CHECK-NAMES**
  Sets RES_NOCHECKNAME in _res.options , which disables the modern BIND checking of incoming host names and mail names for invalid characters such as underscore (_), non-ASCII, or control characters.

QIBM_BIND_RESOLVER_FLAGS
Set this environment variable to a space separated list of resolver option flags. This will override the RES_DEFAULT options (struct state.options) and system configured values (Change TCP/IP Domain - CHGTCPDMN). The state.options structure will be initialized normally, using RES_DEFAULT, OPTIONS environment values and CHGTCPDMN configured values. Then this environment variable will be used to override those defaults. The flags named in this environment variable may be prepended with a '+', '-' or 'NOT_' to set ('+') or reset ('-','NOT_') the value.

For example, to turn on RES_NOCHECKNAME and turn off RES_ROTATE, use the following command from a character-based interface:

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

or

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

QIBM_BIND_RESOLVER_SORTLIST
Set this environment variable to a space-separated list of up to ten IP addresses/mask pairs in dotted decimal format (9.5.9.0/255.255.255.0) to create a sort list (struct state.sort_list).

Data caching

Caching of responses to DNS queries is done by OS/400 sockets in an effort to lessen the amount of network traffic. The cache is added to and updated as needed.

If RES_AAONLY (authoritative answers only) is set in _res.options, the query is always sent on the network. In this case, the cache is never checked for the answer. If RES_AAONLY is not set, the cache is checked for an answer to the query before any attempt to send it on the network is performed. If the answer is found and the time to live has not expired, the answer is returned to the user as the answer to the query. If the time to live has expired, the entry is removed, and the query is sent on the network. Also, if the answer is not found in the cache, the query is sent on the network.

Answers from the network are cached if the responses are authoritative. Non-authoritative answers are not cached. Also, responses received as a result of an inverse query are not cached. You can clear this cache by updating the DNS configuration with either the CHGTCPDMN, CFGTCP option 12, or through iSeries Navigator.

For an example program that uses data caching, see Example: Update and update DNS.

Berkeley Software Distributions (BSD) compatibility

Sockets is a Berkeley Software Distributions (BSD) interface. The semantics, such as the return codes that an application receives and the arguments available on supported functions, are BSD semantics. Some BSD semantics, however, are not available in the OS/400 implementation, and changes may need to be made to a typical BSD socket application in order for it to run on the system.

The following list summarizes the differences between the OS/400 implementation and the BSD implementation.

/etc/hosts, /etc/services, /etc/networks, and /etc/protocols
For these files, the OS/400 implementation supplies the following database files which serve the same functions, respectively.

QUSRSYS file	Contents

QATOCHOST	List of host names and the corresponding IP addresses.
QATOCPN	List of networks and the corresponding IP addresses.
QATOCPP	List of protocols that are used in the Internet.
QATOCPS	List of services and the specific port and protocol that the service uses.

/etc/resolv.conf

The OS/400 implementation requires that this information be configured using the TCP/IP properties page in iSeries Navigator. To access the TCP/IP properties page, complete the following steps:

1. In iSeries Navigator, expand your **iSeries server --> Network --> TCP/IP Configuration**.
2. Right-click **TCP/IP Configuration**.
3. Click **Properties**.

bind() On a BSD system, a client can create an AF_UNIX socket using socket(), connect to a server using connect(), and then bind a name to its socket using bind(). The OS/400 implementation does not support this scenario (the bind() fails).

close()

The OS/400 implementation supports the linger timer for close(), except for AF_INET sockets over SNA. Some BSD implementations do not support the linger timer for close().

connect()

On a BSD system, if a connect() is issued against a socket that was previously connected to an address and is using a connectionless transport service, and an invalid address or an invalid address length is used, the socket is no longer connected. The OS/400 implementation does not support this scenario (the connect() fails and the socket is still connected).

A connectionless transport socket for which a connect() has been issued can be disconnected by setting the address_length parameter to zero and issuing another connect().

accept(), getsockname(), getpeername(), recvfrom(), and recvmsg()

When using AF_UNIX or AF_UNIX_CCSID address family and the socket has not been bound, the default OS/400 implementation may return an address length of zero and an unspecified address structure. The OS/400 BSD 4.4/UNIX 98 and other implementations may return a small address structure with just the address family specified.

ioctl()

- On a BSD system, on a socket of type SOCK_DGRAM, the FIONREAD request returns the length of the data plus the length of the address. On the OS/400 implementation, FIONREAD only returns the length of data.
- Not all requests available on most BSD implementations of **ioctl()** are available on the OS/400 implementation of **ioctl()**.

listen()

On a BSD system, issuing a **listen()** with the backlog parameter set to a value that is less than zero does not result in an error. In addition, the BSD implementation, in some cases, does not use the backlog parameter, or uses an algorithm to come up with a final result for the backlog value. The OS/400 implementation returns an error if the backlog value is less than zero. If you set the backlog to a valid value, then the value is used as the backlog. However, setting the backlog to a value larger than {SOMAXCONN}, the backlog will default to the value set in {SOMAXCONN}.

Out-of-band (OOB) data

In the OS/400 implementation, OOB data is not discarded if SO_OOBINLINE is not set, OOB data has been received, and the user then sets SO_OOBINLINE on. The initial OOB byte is considered normal data.

protocol parameter of socket()

As a means of providing additional security, no user is allowed to create a SOCK_RAW socket specifying a protocol of IPPROTO_TCP or IPPROTO_UDP.

res_xlate() and res_close()

These functions are included in the resolver routines for the OS/400 implementation. **res_xlate()** translates DNS packets from EBCDIC to ASCII and from ASCII to EBCDIC. **res_close()** is used to close a socket that was used by **res_send()** with the RES_STAYOPEN option set. It also resets the _res structure.

sendmsg() and recvmsg()

The OS/400 implementation of **sendmsg()** and **recvmsg()** allows up to and including {MSG_MAXIOVLEN} I/O vectors. The BSD implementation allows {MSG_MAXIOVLEN - 1} I/O vectors.

shutdown()

On an OS/400 implementation, the output function that is currently blocked on the socket descriptor will continue to block after a **shutdown()**. On a BSD implementation, the blocking output function is ended with the [EPIPE] errno value. Similarly, a BSD implementation ends blocking input operations with a zero output value when they are blocking and a **shutdown()** is issued from another process or thread. The OS/400 implementation simply fails any subsequent input function with a zero output value, but the blocking input function continues to block until data is received or some other action is taken to remove it from a waiting state.

Signals

There are several differences relating to signal support:

- BSD implementations issue a SIGIO signal each time an acknowledgement is received for data sent on an output operation. The OS/400 sockets implementation does not generate signals related to outbound data.
- The default action for the SIGPIPE signal is to end (terminate) the process in BSD implementations. To maintain downward compatibility with previous releases of OS/400, the OS/400 implementation uses a default action of ignore for the SIGPIPE signal.

SO_REUSEADDR option

On BSD systems, a **connect()** on a socket of family AF_INET and type SOCK_DGRAM causes the system to change the address to which the socket is bound to the address of the interface that is used to reach the address specified on the **connect()**. For example, if you bind a socket of type SOCK_DGRAM to address INADDR_ANY, and then connect it to an address of a.b.c.d, the system changes your socket so it is now bound to the IP address of the interface that was chosen to route packets to address a.b.c.d. In addition, if this IP address that the socket is bound to is a.b.c.e, for example, address a.b.c.e now appears on **getsockname()** instead of INADDR_ANY, and the SO_REUSEADDR option must be used to bind any other sockets to the same port number with an address of a.b.c.e.

In contrast, in this example, the OS/400 implementation does not change the local address from INADDR_ANY to a.b.c.e. **getsockname()** continues to return INADDR_ANY after the connect is performed.

SO_SNDBUF and SO_RCVBUF options

The values set for SO_SNDBUF and SO_RCVBUF on a BSD system provide a greater level of control than on an OS/400 implementation. On an OS/400 implementation, these values are taken as advisory values.

UNIX 98 compatibility

Created by Open Group, a consortium of developers and venders, UNIX 98 improved the inoperability of the UNIX operating system while incorporating much of the Internet-related function for which UNIX had become known. New for V5R2, OS/400 sockets provides programmers the ability to write socket applications that are compatible with UNIX 98 operating environment. Currently, IBM supports two versions of most sockets APIs. The base OS/400 API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with BSD 4.4 and the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface by defining the _XOPEN_SOURCE macro to a value of 520 or greater.

Differences in address structure for UNIX 98 compatible applications

When you specify the _XOPEN_OPEN macro, you can write UNIX 98 compatible applications with the same address families that are used in default OS/400 implementations; however, there are differences in the **sockaddr** address structure. The table compares the BSD 4.3 **sockaddr** address structure with the UNIX 98 compatible address structure:

Table 17. Comparison of BSD 4.3 and UNIX 98/BSD 4.4 socket address structure

<table>
<tr><th>BSD 4.3 structure</th><th>BSD 4.4/UNIX 98 compatible structure</th></tr>
<tr><td colspan="2"><b>sockaddr address structure</b></td></tr>
<tr><td><pre>
struct sockaddr {
 u_short sa_family;
 char   sa_data[14];
};
</pre></td><td><pre>
struct sockaddr {
 uint8_t   sa_len;
 sa_family_t sa_family
 char   sa_data[14];
};
</pre></td></tr>
<tr><td colspan="2"><b>sockaddr_in address structure</b></td></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>
<tr><td colspan="2"><b>sockaddr_in6 address structure</b></td></tr>
<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"><b>sockaddr_un address structure</b></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 differences

When you develop in ILE-based languages and an application is compiled with the _XOPEN_SOURCE macro, some sockets APIs are mapped to internal names. These internal names provide the same function as the original API. The table lists these affected APIs. If you are writing socket applications in some other C-based language, you can write directly to the internal name of these APIs. Use the link to the original API to see usage notes and details for both versions of these APIs.

Table 18. API and UNIX 98 equivelant name

API name	Internal name

Table 18. API and UNIX 98 equivelant name (continued)

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()
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()

Table 18. API and UNIX 98 equivelant name (continued)

sethostent()	qso_sethostent98()
setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setprotoent98()
setsockopt()	qso_setsockopt98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

Pass descriptors between processes—sendmsg() and recvmsg()

The ability to pass an open descriptor between jobs can lead to a new way of designing client/server applications. Passing an open descriptor between jobs allows one process, typically a server, to do everything that is required to obtain the descriptor (open a file, establish a connection, wait for the **accept()** API to complete) and let another process, typically a worker, handle all the data transfer operations once the descriptor is open. This design results in simpler logic for both the server and the worker jobs. This design also allows different types of worker jobs to be easily supported. The server can make a simple check to determine which type of worker should receive the descriptor.

Sockets provides three sets of APIs that can pass descriptors between server jobs:

- **spawn()**

  Note: spawn() is not a socket API. It is supplied as part of the OS/400 Process-Related APIs.
- **givedescriptor()** and **takedescriptor()**
- **sendmsg()** and **recvmsg()**

The **spawn()** API starts a new server job (often called a ″child job″) and gives certain descriptors to that child job. If the child job is already active, then the **givedescriptor()** and **takedescriptor()** or the **sendmsg()** and **recvmsg()** APIs need to be used.

However, the **sendmsg()** and **recvmsg()** APIs offer many advantages over **spawn()** and **givedescriptor()** and **takedescriptor()**:

Portability
The **givedescriptor()** and **takedescriptor()** APIs are non-standard and unique to the iSeries. If the portability of an application between iSeries and UNIX is an issue, you may want to use the **sendmsg()** and **recvmsg()** APIs instead.

Communication of control information
Often the worker job needs to know additional information when it receives a descriptor, such as the following:

- What type of descriptor is it?
- What should the worker job do with it?

The **sendmsg()** and **recvmsg()** APIs allow you to transfer data, which may be control information, along with the descriptor; the **givedescriptor()** and **takedescriptor()** APIs do not.

Performance
Applications that use the **sendmsg()** and **recvmsg()** APIs tend to perform slightly better than those that use the **givedescriptor()** and **takedescriptor()** APIs in three areas:

- Elapsed time
- CPU utilization

- Scalability

The amount of performance improvement of an application depends on the extent that the application passes descriptors.

Pool of worker jobs

You may want to set up a pool of worker jobs so that a server can pass a descriptor and only one of the jobs in the pool will become active and receive it. The **sendmsg()** and **recvmsg()** APIs can be used to accomplish this by having all of the worker jobs wait on a shared descriptor. When the server calls **sendmsg()**, only one of the worker jobs will receive the descriptor.

Unknown worker job ID

The **givedescriptor()** API requires the server job to know the job identifier of the worker job. Typically the worker job obtains the job identifier and transfers it over to the server job with a data queue. The **sendmsg()** and **recvmsg()** do not require the extra overhead to create and manage this data queue.

Adaptive server design

When a server is designed using the **givedescriptor()** and **takedescriptor()**, a data queue is typically used to transfer the job identifiers from worker jobs over to the server. The server then does a **socket()**, **bind()**, **listen()**, and an **accept()**. When the **accept()** API completes, the server pulls off the next available job ID from the data queue. It then passes the inbound connection to that worker job. Problems arise when many incoming connection requests occur at once and there are not enough worker jobs available. If the data queue that contains the worker job identifiers is empty, the server blocks waiting for a worker job to become available, or the server creates additional worker jobs. In many environments, neither of these alternatives are desirable because additional incoming requests may fill the listen backlog.

Servers that use **sendmsg()** and **recvmsg()** APIs to pass descriptors remain unhindered during heavy activity because they do not need to know which worker job is going to handle each incoming connection. When a server calls **sendmsg()**, the descriptor for the incoming connection and any control data are put into an internal queue for the AF_UNIX socket. When a worker job becomes available, it will call **recvmsg()** and receive the first descriptor and the control data that was in the queue.

Inactive worker job

The **givedescriptor()** API requires the worker job to be active while the **sendmsg()** API does not. The job that calls **sendmsg()** does not require any information about the worker job. The **sendmsg()** API requires only that an AF_UNIX socket connection has been set up.

An example of how the **sendmsg()** API can be used to pass a descriptor to a job that does not exist follows:

A server can use the **socketpair()** API to create a pair of AF_UNIX sockets, use the **sendmsg()** API to send a descriptor over one of the AF_UNIX sockets created by **socketpair()**, and then call **spawn()** to create a child job that inherits the other end of the socket pair. The child job calls **recvmsg()** to receive the descriptor that the server passed. The child job was not active when the server called **sendmsg()**.

Pass more than one descriptor at a time

The **givedescriptor()** and **takedescriptor()** APIs allow only one descriptor to be passed at a time. The **sendmsg()** and **recvmsg()** APIs can be used to pass an array of descriptors.

For a sample program that uses the **sendmsg()** and **recvmsg()** APIs, see Example: Pass descriptors between processes.

Socket scenario: Create an application to accept IPv4 and IPv6 clients

Situation

Suppose you are a socket programmer that works for an application development company that specializes in socket applications for iSeries. To keep ahead of their competitors, you have decided to develop a suite of applications that will use the AF_INET6 address family, which accept connections from IPv4 and IPv6 . You want to create an application that will process requests from both IPv4 and IPv6 nodes. You know that the OS/400 supports the AF_INET6 address family sockets which provides interoperability with AF_INET address family sockets. You also know this can be accomplished by using an IPv4-mapped IPv6 address format. See IPv6 applications compatibility with IPv4 applications for more details on working with interoperability between IPv6 and IPv4 applications.

Scenario objectives

This scenario has the following objectives and goals:

1. Create an server application that accepts and processes requests from IPv6 and IPv4 clients
2. Create a client application that requests data from an IPv4 or IPv6 server application

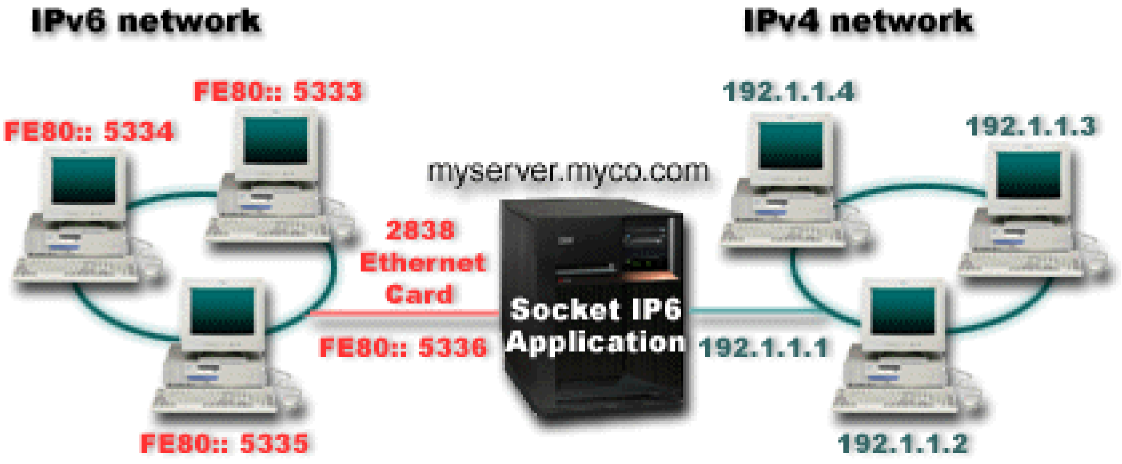
Prerequisite steps

Before developing your application that meets these objectives, you complete the following tasks:

1. Install QSYSINC library. This library provides necessary header files that are needed when compiling socket applications.
2. Install the C Compiler licensed program (5722–CX2).
3. Install and configure the 2838 Ethernet card. For information on Ethernet options, see Ethernet topic in the Information Center.
4. Set up TCP/IP and IPv6 network.

Scenario details

The following graphic describes the IPv6 network for which you will be creating applications to handle requests from IPv6 and IPv4 clients. The iSeries contains the program that will listen and process requests from these clients. The network is comprised of two separate domains, one that contains IPv4 clients exclusively and the other remote network containing only IPv6 clients. The domain name of the iSeries is myserver.myco.com. The server application will use the AF_INET6 address family to process these incoming requests with the in6addr_any specified on the **bind()** function call.



See the following example programs that are used within this scenario:

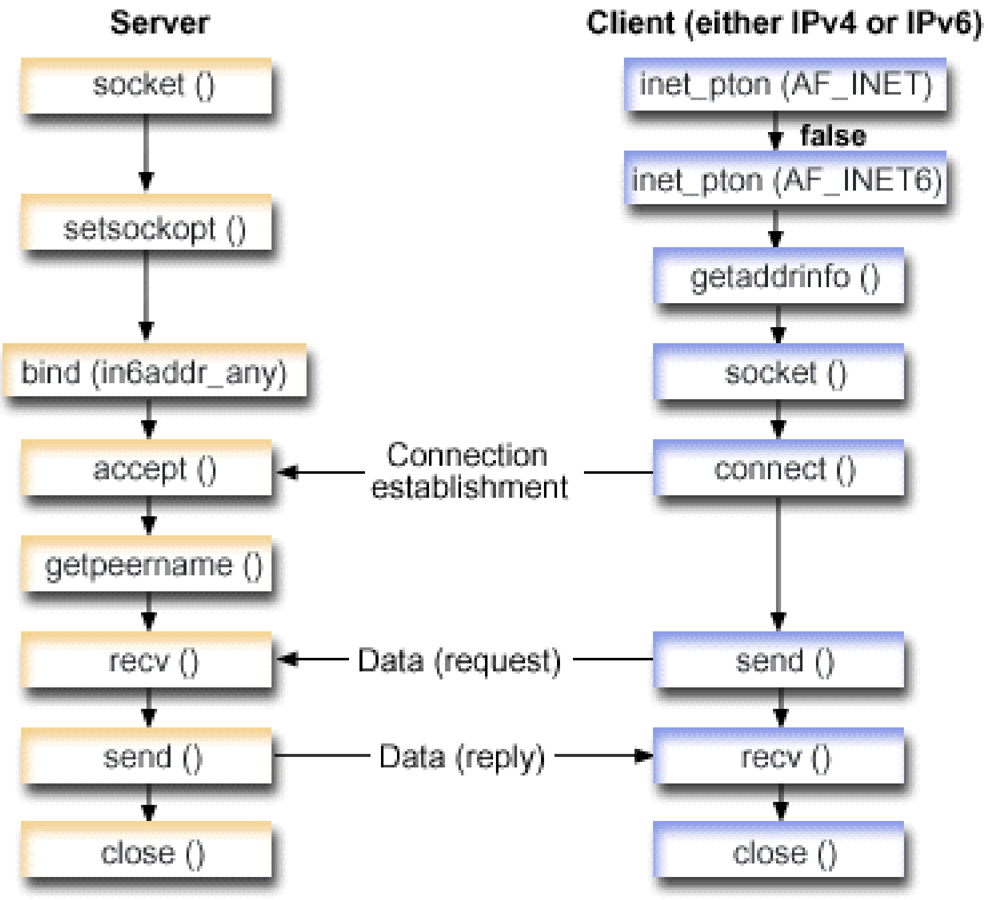
- Example: Accept connections from both IPv6 and IPv4 clients

- Example: IPv4 or IPv6 client

Example: Accept connections from both IPv6 and IPv4 clients

Use this sample program to create a server/client model that accepts requests from both IPv4 (those socket applications that use AF_INET address family) and IPv6 (those applications that use the AF_INET6 address family). Currently your socket application may only use AF_INET address family, which allows for TCP and UDP protocol; however, this may change with the increase in the use of IPv6 addresses. You can use this sample program to create your own applications that accomodate both address families.

This graphic shows how this example program works:



Socket flow of events: Server application that accepts requests from both IPv4 and IPv6 clients

This flow describes each of the function calls and what they do within the socket application that accepts requests both IPv4 and IPv6 clients.

1. The **socket()** API specifies a socket descriptor that creates an endpoint. It also specifies the AF_INET6 address family, which supports IPv6, and the TCP transport (SOCK_STREAM) will be used for this socket.
2. The **setsockopt()** function allows application to reuse the local addresswhen the server is restarted before the required wait time expires.

3. A **bind()** function supplies a unique name for the socket. In this example, the programmer set 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 port spaces).

   Note: If the server only needs to handle IPv6 clients, then IPV6_ONLY socket option can be used.
4. The **listen()** function allows the server to accept incoming client connections. In this example, the programmer set the backlog to 10, which allows the system to queue 10 connection requests before the system starts rejecting incoming requests.
5. 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.
6. The **getpeername()** function returns the client's address to the application. If the client is an IPv4 client, the address will be shown as an IPv4–mapped IPv6 address.
7. The **recv()** function receives that 250 bytes data from the client. In this example, the programmer knows that the client will send 250 bytes of data over. Knowing this, the programmer uses the SO_RCVLOWAT socket option and specify that she don't want the **recv()** to wake up until all 250 bytes of data have arrived.
8. The **send()** function echoes the data back to the client.
9. The **close()** function closes any open socket descriptors.

Socket flow of events: Requests from either IPv4 or IPv6 clients

Note: This client example can be used with other server application designs that wish to accept request for either IPv4 or IPv6 nodes. Other server designs, like those described in Examples: Connection-oriented designs, can be used with this client example.

1. The **inet_pton()** call converts the text form of the address to the binary form. In this example, two of these calls are issued. The first determines if the server is a valid AF_INET address. The second **inet_pton()** call determines whether the server has an AF_INET6 address. If it is numeric then we want to prevent **getaddrinfo()** from doing any name resolution. Otherwise a host name was provided that needs to be resolved whe the **getaddrinfo()** call is issued.
2. The **getaddrinfo()** call retrieves the address information needed for the subsequent **socket()** and **connect()** calls.
3. 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()**.
4. The **connect()** function establishes a connection to the server regardless of whether the server is IPv4 or IPv6..
5. The **send()** function sends the data request to the server.
6. The **recv()** function receives data from the server application.
7. The **close()** function closes any open socket descriptors.

The following sample code, shows the server application for this scenario. For the client application, see Example: IPv4 or IPv6 client. For information on the use of this code, see code disclaimer information.

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

Example: IPv4 or IPv6 client

The sample program can be used with the server application that accepts requests from either IPv4 or IPv6 clients.

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

Socket application design recommendations

Before working with socket applications, assess the functional requirements, goals, and needs of the socket application. Consider performance requirements and system resource impacts of the application too. The following list of recommendations will help you address some of these issues for your socket application and point out better ways to use sockets and to design your socket applications:

Table 19. Socket design applications

Recommendation	Reason	Best used in
Use Asynchronous I/O	Asynchronous I/O used in a threaded server model is preferable over the more conventional select() model. For more information on advantages of using Asynchronous I/O, see Asynchronous I/O. For a sample program that uses the asynchronous I/O APIs, see Example: Using asynchronous I/O.	Socket server applications which will be handling numerous concurrent clients.
When using Asynchronous I/O, adjust the number of threads in the process to an optimum number for the number of clients to be processed.	If too few threads are defined then some clients might time out before being handled. If too many threads are defined then some system resource will not be used efficiently. **Note:** It is better to have too many threads than too few threads.	Socket applications using Asynchronous I/O.
Design socket application to avoid the use of the postflag on all start operations for Asynchronous I/O.	Avoids the performance overhead of transition to a completion port if the operation has already been satisfied synchronously.	Socket applications using Asynchronous I/O.
Use **send()** and **recv()** over **read()** and **write()**.	**send()** and **recv()** APIs provide a small performance and serviceability improvement over **read()** and **write()**.	Any socket program that knows it uses a socket descriptor and not a file descriptor.

Table 19. Socket design applications (continued)

Use the receive low water (SO_RCVLOWAT) socket option to avoid looping on a receive operation until all data has arrived.	Allows your application to wait for a minimum amount of data to be received on the socket before satisfying a blocked receive operation.	Any socket application that receives data
Use the MSG_WAITALL flag to avoid looping on a receive operation until all data has arrived.	Allows your application to wait for the entire buffer provided on the receive operation to be received before satisfying a blocked receive operation.	Any socket application that receives data and knows in advance how much it expects to arrive.
Use **sendmsg()** and **recvmsg()** over **givedescriptor()** and **takedescriptor()**.	See Passing descriptors between processes—sendmsg() and recvmsg() for the advantages. See Example: Passing descriptors between processes for a sample program that uses **sendmsg()** and **recvmsg()**.	Any socket application passing socket or file descriptors between processes.
When using **select()**, try to avoid a large number of descriptors in the read, write or exception set. **Note:** If you have a large number of descriptors being used for **select()** processing see the Asynchronous I/O recommendation above.	When there are a large number of descriptors in a read, write or exception set, considerable redundant work occurs each time **select()** is called. Once a **select()** is satisfied the actual socket function must still be done, i.e. a read or write or accept must still be performed. Asynchronous I/O APIs combine the notification that something has occurred on a socket with the actual I/O operation.	Applications that have a large(> 50) number of descriptors active for **select()**.
Save your copy of the read, write and exception sets before using **select()** to avoid rebuilding the sets for every time you have to reissue the **select()**.	This saves the overhead of rebuilding the read, write, or exception sets every time you plan to issue the **select()**. See Example: Non-blocking I/O and select() for a sample program that uses **select()**.	Any socket application where you are using **select()** with a large number of socket descriptors enabled for read, write or exception processing.
Do not use **select()** as a timer. Use **sleep()** instead. **Note:** If granularity of the **sleep()** timer is not adequate, you might have to use **select()** as a timer. In this case, set maximum descriptor to 0 and the read, write and exception set to NULL.	Better timer response and less system overhead.	Any socket application where you are using **select()** just as a timer.
If your socket application has increased the maximum number of file and socket descriptors allowed per process using DosSetRelMaxFH() and you are using **select()** in this same application, be careful of the affict this new maximum value has on the size of the read, write and exception sets used for **select()** processing.	If you allocate a descriptor outside the range of the read, write or exception set, as specified by FD_SETSIZE, then you could overwrite and corrupt storage. Ensure your set sizes are at least large enough to handle whatever the maximum number of descriptors are set for the process and the maximum descriptor value specified on the **select()** API.	Any application or process where you use DosSetRelMaxFH() and **select()**.

Table 19. Socket design applications (continued)

Set all socket descriptors in the read or write sets to non-blocking. When a descriptor becomes enabled for read or write, loop and consume or send all of the data until EWOULDBLOCK is returned. See Example: Non-blocking I/O and select() for a sample program that uses **select()**.	This will allow you to minimize the number of **select()** calls when data is still available to be processed or read on a descriptor.	Any socket application where you are using **select()**.
Only specify the sets that you need to use for **select()** processing.	Most applications do not need to specify the exception set or write set.	Any socket application where you are using **select()**.
Use GSKit APIs instead of the SSL APIs.	Both the Global Secure Toolkit (GSKit) and OS/400 SSL_ APIs allow you to develop secure AF_INET or AF_INET6, SOCK_STREAM socket application Because the GSKit APIs are supported across IBM@server platforms, it is the preferred set of APIs to secure an application. The SSL_ APIs are native to the OS/400 system only.	Any socket application which needs to be enabled for SSL/TLS processing.
Avoid using signals.	The performance overhead of signals (on all platforms, not just the iSeries) is expensive. It is better to design your socket application to use Asynchronous I/O or **select()** APIs.	Any programmer considering the use of signals in their socket application.
Use protocol independent routines when available, such as **inet_ntop()**, **inet_pton()**, **getaddrinfo()**, and **getnameinfo()**.	Even if you are not yet ready to support IPv6, use these APIs, (instead of **inet_ntoa()**, **inet_addr()**, **gethostbyname()** and **gethostbyaddr**()) to prepare you for easier migration.	Any AF_INET or AF_INET6 application that uses network routines.
Use sockaddr_storage to declare storage for any address family address.	Simplifies writing code portable across multiple address families and platforms. Declares enough storage to hold the largest address family and ensures the correct boundary alignment.	Any socket application that stores addresses.

Examples: Socket application designs

The following examples provide many sample programs that illustrate the more advanced socket concepts. You can use these sample programs to create your own applications that complete a similar task. With these examples, there are graphics and a listing of calls that illustrate the flow of events in each of these applications. You can use the Xsocket tool interactively try some of these APIS in these programs, or you can make specific changes for your particular environment.

- Examples: Connection-oriented designs
- Example: Establish secure connections
- Example: Use gethostbyaddr_r() for threadsafe network routines
- Example: Non-blocking I/O and select()
- Example: Use signals with blocking socket APIs
- Example: Use multicasting with AF_INET address family
- Example: Query and update DNS

- Example: Transfer file data using send_file() and accept_and_recv() APIs

Examples: Connection-oriented designs

There are a number of ways that you can design a connection-oriented socket server on the iSeries. The example programs that follow can be used to create your own connection-oriented designs. While additional socket server designs are possible, the designs provided in the examples below are the most common:

Iterative server

In the iterative server example, a single server job handles all incoming connections and all data flows with the client jobs. When the **accept()** API completes, the server handles the entire transaction. This is the easiest server to develop, but it does have a few problems. While the server is handling the request from a given client, additional clients could be trying to get to the server. These requests fill the **listen()** backlog and some of the them will be rejected eventually.

All of the remaining examples are concurrent server designs. In these designs, the system uses multiple jobs and threads to handle the incoming connection requests. With a concurrent server there are usually multiple clients that connect to the server at the same time.

For multiple concurrent clients in a network, it is recommend that you use the Asynchronous I/O socket APIs. These APIs provide the best performance in networks that have multiple concurrent clients. Asynchronous I/O describes what these APIs do and how they work. For an example program that uses these APIs, see Example: Use asynchronous I/O.

spawn() server and spawn() worker

The spawn() server and spawn() worker example uses the spawn() API to create a new job to handle each incoming request. After spawn() completes, the server can then wait on the **accept()** API for the next incoming connection to be received.

The only problem with this server design is the performance overhead of creating a new job each time a connection is received. You can avoid the performance overhead of the spawn() server example by using prestarted jobs. Instead of creating a new job each time a connection is received, the incoming connection is given to a job that is already active. All of the remaining examples in this topic use prestarted jobs.

sendmsg() server and recvmsg() worker

The sendmsg() server and recvmsg() worker example pass the incoming connection to the worker (client) job. The server prestarted all of the worker jobs when the server job first started.

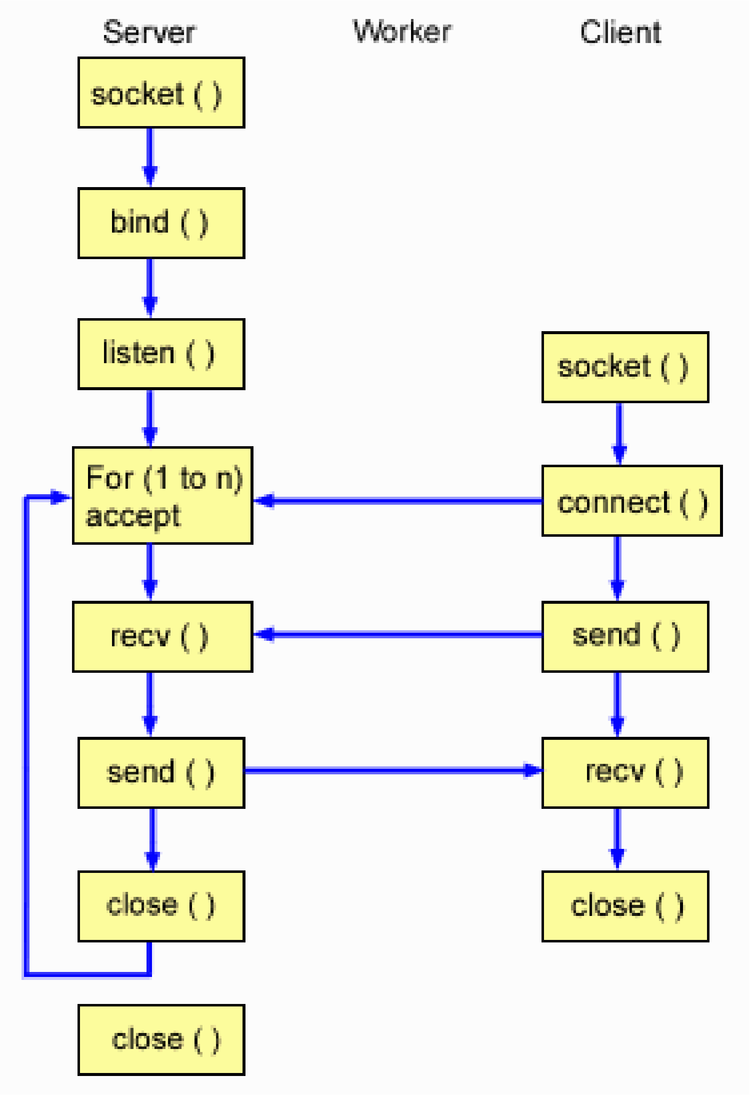
Multiple accept() servers and multiple accept() workers

In the previous examples, the worker job did not get involved until after the server received the incoming connection request. The multiple accept() servers and multiple accept() workers example of the system turns each of the worker jobs into an iterative server. The server job still calls the **socket()**, **bind()**, and **listen()** APIs. When the **listen()** call completes, the server creates each of the worker jobs and gives a listening socket to each one of them. All of the worker jobs then call the **accept()** API. When a client tries to connect to the server, only one **accept()** call completes, and that worker handles the connection.

All of these examples use a basic design for the client connection. See Example: Generic client for details.

Example: Write an iterative server program

Use this example to creat a single server job that handles all incoming connections. When the **accept()** API completes, the server handles the entire transaction. The figure illustrates how the server and client jobs interact when the system used the iterative server design. See Example: Generic client for an example that contains the code for a common client job that can be used with this example.



Flow of socket events: Iterative server

The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and worker applications. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. For a description of the client portion of this flow, see Example: Generic client. The following sequence shows the function calls for this sample program for and iterative server application:

1. 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.
2. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.
3. The **listen()** allows the server to accept incoming client connections.
4. 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.
5. The **recv()** function receives data from the client application.
6. The **send()** function echoes the data back to the client.
7. The **close()** function closes any open socket descriptors.

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

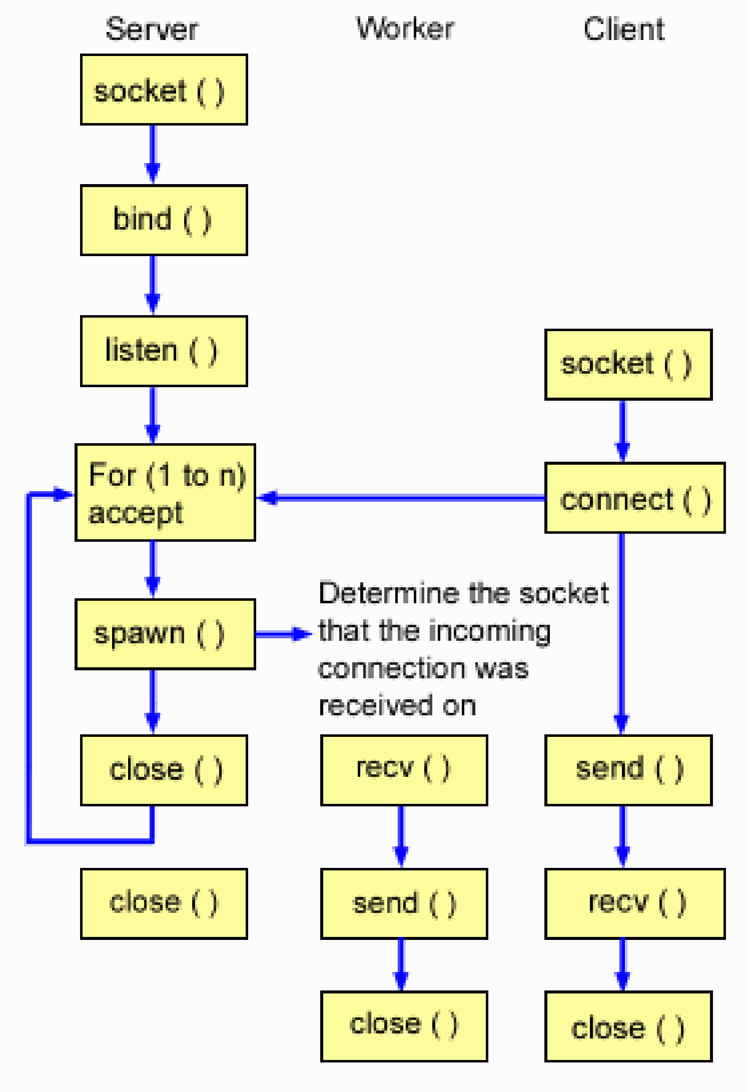
```
   /*************************************************/
   /* Close down the listen socket                  */
   /*************************************************/
   close(listen_sd);
}
```

Example: Use the spawn() API to create child processes

This example shows how a server program can use the **spawn()** API to create a child process that inherits the socket descriptor from the parent. The server job waits for an incoming connection, and then calls **spawn()** to create children jobs to handle the incoming connection. The child process inherits the following attributes with the **spawn()** function:

- The socket and file descriptors
- The signal mask
- The signal action vector
- The environment variables

The following figure illustrates how the server, worker, and client jobs interact when the **spawn()** server design is used.



Flow of socket events: Server that uses spawn() to accept and process requests
The following sequence of the socket calls provide a description of the graphic. It also describes the

relationship between the server and worker examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The Example: Create a server that uses spawn() uses the following socket calls to create a child process with the **spawn()** function call:

1. 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.
2. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.
3. The **listen()** allows the server to accept incoming client connections.
4. 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.
5. The **spawn()** function initialize the parameters for a work job to handle incoming requests. In this example, the socket descriptor for the new connection is mapped over to descriptor 0 in the child program.
6. In this example, the first **close()** function closes the listening socket descriptor. The second **close ()** call ends the accepted socket.

Socket flow of events: Worker job created by spawn()
The Example: Enable the worker job to receive a data buffer uses the following sequence of function calls:

1. After the **spawn()** function is called on the server, the **recv()** function receives the data from the incoming connection.
2. The **send()** function echos data back to the client.
3. The **close()** function ends the spawned worker job.

Example: Create a server that uses spawn(): This example shows how to use the **spawn()** API to create a child process that inherits the socket descriptor from the parent. For information on the use of code examples, see the code disclaimer.

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

See Example: Enable the worker job to receive a data buffer for a sample program that uses the socket descriptor to complete processes.

Example: Enable the worker job to receive a data buffer: This example contains the code that enables the worker job to receive a data buffer from the client job and echo it back. For information on the use of code examples, see the code disclaimer.

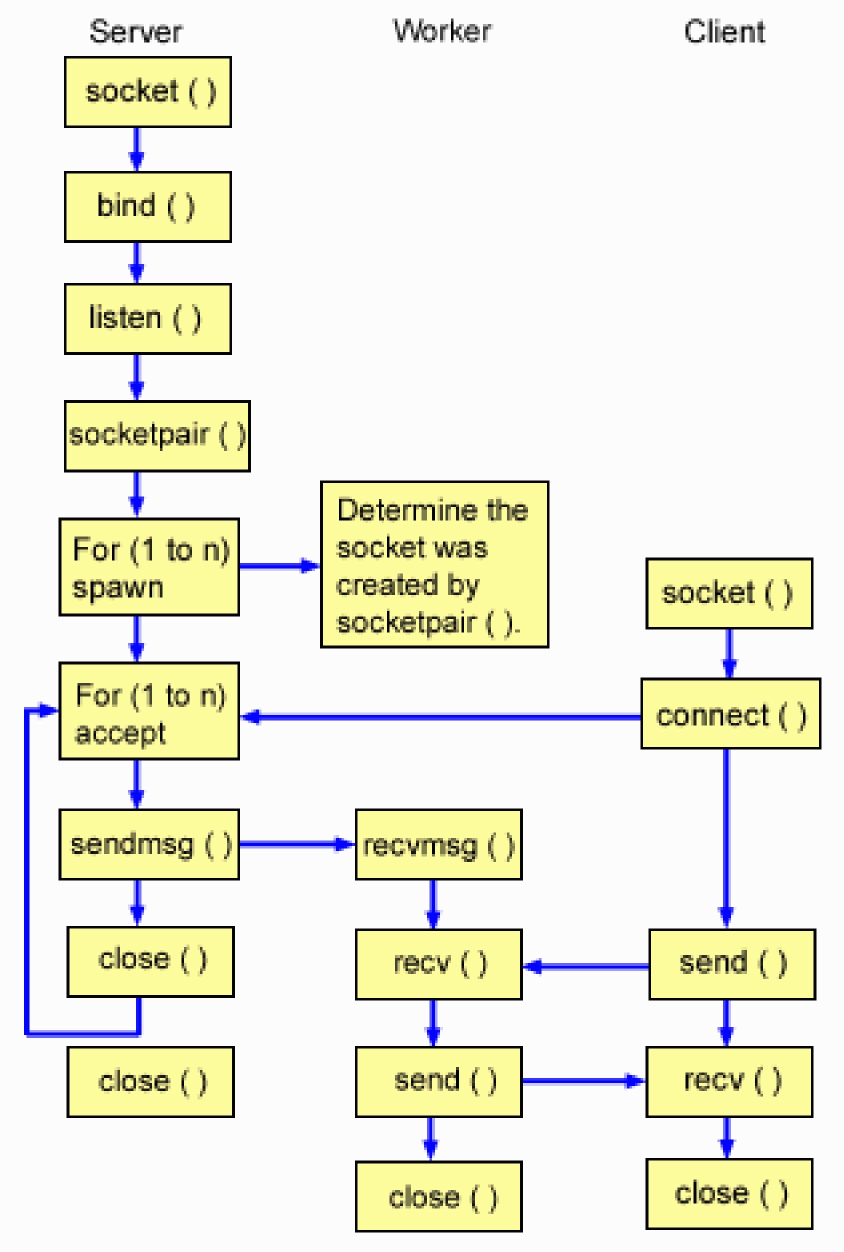
```
/**************************************************************************/
/* Worker job that receives and echoes back a data buffer to a client     */
/**************************************************************************/

#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);

}
```

Example: Pass descriptors between processes

The **sendmsg()** and **recvmsg()** examples show how to design a server program that uses these APIs to handle incoming connections. When the server starts, it creates a pool of worker jobs. These preallocated (spawned) worker jobs wait until needed. When the client job connects to the server, the server gives the incoming connection to one of the worker jobs.

The following figure illustrates how the server, worker, and client jobs interact when the system uses the **sendmsg()** and **recvmsg()** server design.

Note: For a description of the client portion of this diagram, see Example: Generic client.



Flow of socket events: Server that uses sendmsg() and recvmsg() functions
The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and worker examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. Example: Server program used for sendmsg() and recvmsg() uses the following socket calls to create a child process with the **sendmsg()** and **recvmsg()** function calls:

1. 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.
2. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.
3. The **listen()** allows the server to accept incoming client connections.
4. The **socketpair()** function creates a pair of UNIX datagram sockets. A server can use the **socketpair()** API to create a pair of AF_UNIX sockets.

5. The **spawn()** function initialize the parameters for a work job to handle incoming requests. In this example, the child job created inherits the socket descriptor that was created by the **socketpair()**.
6. 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.
7. The **sendmsg()** function sends a incoming connection to one of the worker jobs. The child process accepts the connection with the**recvmsg()** function. The child job is not active when the server called **sendmsg()**.
8. In this example, the first **close()** function closes the accepted socket. The second **close ()** call ends the listening socket.

Socket flow of events: Worker job that uses recvmsg()
Example: Worker program used for sendmsg () and recvmsg () uses the following sequence of function calls:

1. After the server has accepted a connection and passed its socket descriptor to the worker job, the **recvmsg()** function receives the descriptor. In this example, the **recvmsg()** function will wait until the server sends the descriptor.
2. The **recv()** function receives a message from the client.
3. The **send()** function echos data back to the client.
4. The **close()** function ends the worker job.

Example: Server program used for sendmsg() and recvmsg(): This example shows how to use the **sendmsg()** API to create a pool of worker jobs. See Example: Generic client for an example that contains the code for a common client job that can be used with this example. For information on the use of code examples, see the code disclaimer.

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

Example: Worker program used for sendmsg () and recvmsg (): This example shows how to use the **recvmsg()** API client job to receive the worker jobs. For information on the use of code examples, see the code disclaimer.

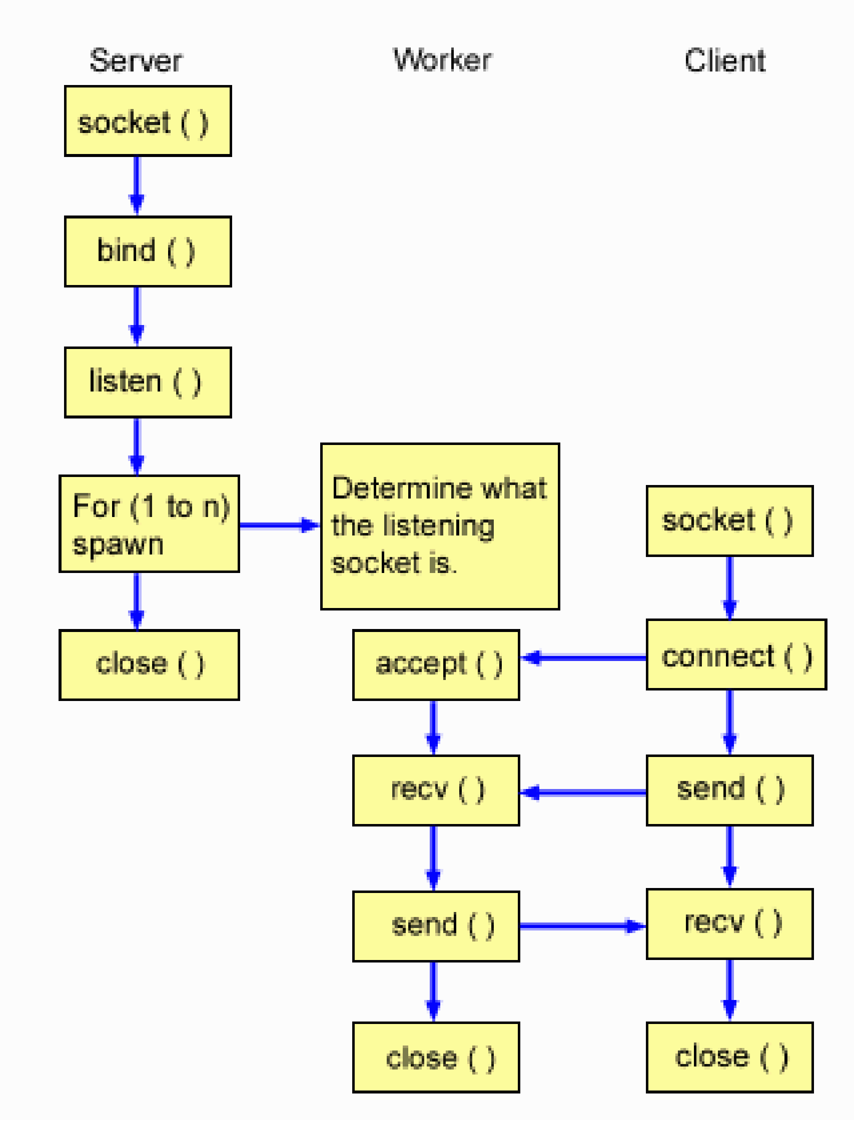
```
/**************************************************************************/
/* Worker job that uses the recvmsg to process client requests            */
/**************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

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);
}
```

Example: Use multiple accept() APIs to handle incoming requests

These examples show how to design a server program that uses the multiple **accept()** model for handling incoming connection requests. When the multiple **accept()** server starts up, it does a **socket()**, **bind()**, and **listen()** as normal. It then creates a pool of worker jobs and gives each worker job the listening socket. Each multiple **accept()** worker then calls **accept()**.

The following figure illustrates how the server, worker, and client jobs interact when the system uses the multiple **accept()** server design.

Note: For a description of the client portion of this diagram, see Example: Generic client.



Flow of socket events: Server that creates a pool of multiple accept() worker jobs
The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and worker examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. Example: Server program to create a pool of multiple accept() worker jobs uses the following socket calls to create a child process:

1. 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.
2. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.
3. The **listen()** allows the server to accept incoming client connections.
4. The **spawn()** function creates each of the worker jobs.
5. In this example, the first **close()** function closes the listening socket.

Socket flow of events: Worker job that multiple accept()
Example: Worker jobs for multiple accept() uses the following sequence of function calls:

1. After the server has spawned the worker jobs, the listen socket descriptor is passed to this worker job as a command line parameter. The **accept()** function waits for an incoming client connection.
2. The **recv()** function receives a message from the client.
3. The **send()** function echos data back to the client.
4. The **close()** function ends the worker job.

Example: Server program to create a pool of multiple accept() worker jobs: This example shows how to use the multiple **accept()** model to create a pool of worker jobs. See Example: Generic client for an example that contains the code for a common client job that can be used with this example. For information on the use of code examples, see the code disclaimer.

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

Example: Worker jobs for multiple accept(): This example shows how multiple **accept()** APIs receive the worker jobs and call the **accept()** server. For information on the use of code examples, see the code disclaimer.

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

Example: Generic client

The following code example contains the code for a common client job. The client job does a **socket()**, **connect()**, **send()**, **recv()**, and **close()**. The client job is not aware that the data buffer it sent and received are going to a worker job rather than the server. If you would like to create a client application that works whether the server is AF_INET address family or AF_INET6 address family, use Example: IPv4 or IPv6 client.

This client job will work with each of these common connection-oriented server designs:

- An iterative server. See Example: Write an iterative server program for a sample program.
- A spawn server and worker. See Example: Use the spawn() API to create child processes for a sample program.
- A sendmsg() server and rcvmsg() worker. See Example: Server program used for sendmsg() and recvmsg() for a sample program.
- A multiple accept() design. See Example: Server program to create a pool of multiple accept() worker jobs for a sample program.
- A non-blocking I/O and select() design. See Example: Non-blocking I/O and select() for a sample program.
- A server that accepts connections from either an IPv4 or IPv6 client.. See Example: Accept connections from both IPv6 and IPv4 clients for a sample program.

Socket flow of events: Generic client

The following sample program uses the following sequence of function calls:

1. 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.
2. After the socket descriptor is received, the **connect()** function is used to establish a connection to the server.
3. The **send()** function send data buffer to the worker job(s).
4. The **recv()** function receive data buffer from the worker job(s).
5. The **close()** function closes any open socket descriptors.

For information on the use of code examples, see the code disclaimer.

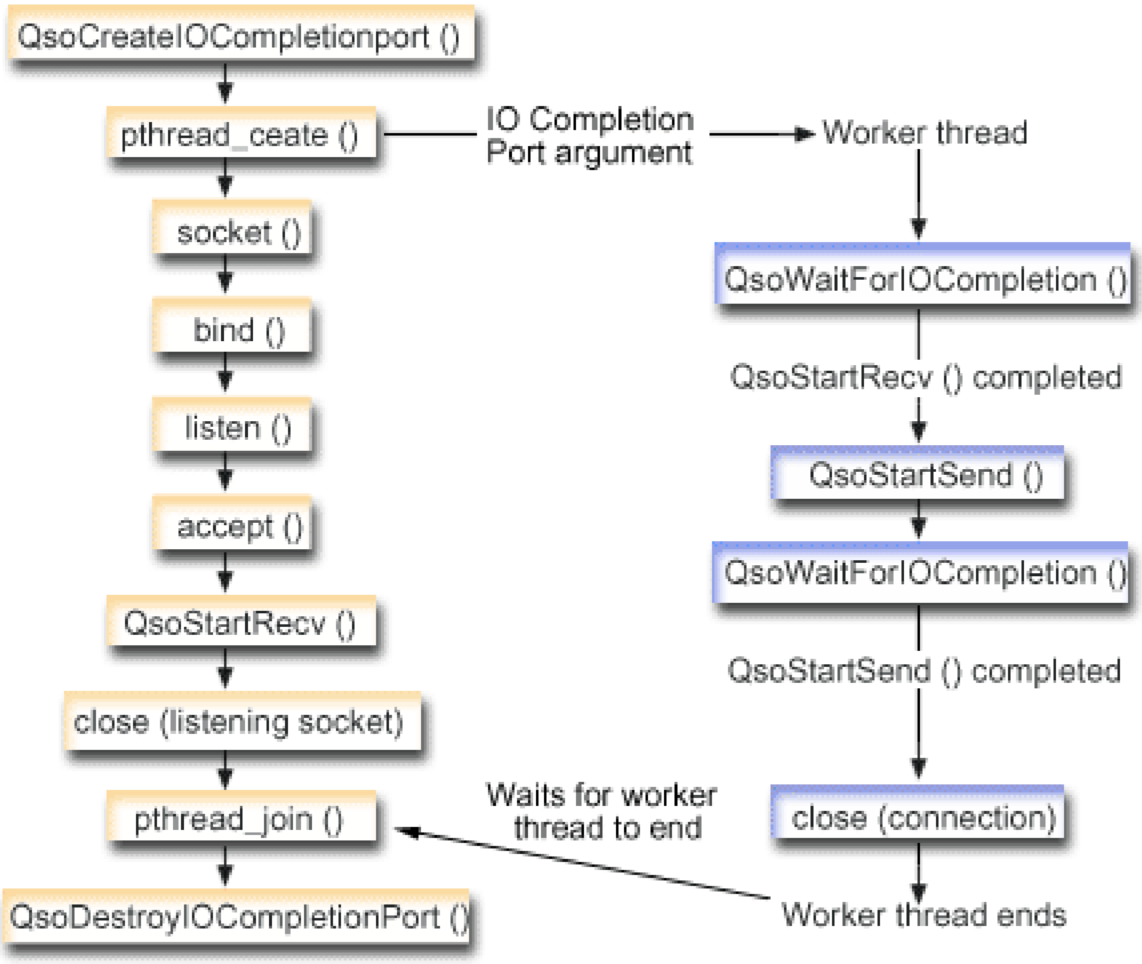
```
/**************************************************************************/
/* Generic client example is used with connection-oriented server designs */
/**************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include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

Example: Use asynchronous I/O

An application will create an I/O completion port using the **QsoCreateIOCompletionPort()** API. This API will return a handle which can be used to schedule and wait for completion of asynchronous I/O requests. The application will start an input or an output function, specifying a I/O completion port handle. When the I/O completes, status information and an application-defined handle will be posted to the specified I/O completion port. The post to the I/O completion port will wake up exactly one of possibly many threads that are waiting. The application receives:

- a buffer that was supplied on the original request
- the length of data that was processed to or from that buffer
- a indication of what type of I/O operation has completed
- and application-defined handle that was passed on the initial I/O request

This application handle could simply be the socket descriptor identifying the client connection, or a pointer to storage that contains extensive information about the state of the client connection. Since the operation completed and the application handle was passed, the worker thread determines the next step to complete the client connection. Worker threads that process these completed asynchronous operations may handle many different client requests and are not tied to just one. Because copying to and from user buffers occurs asynchronously to the server processes, wait time for client request diminishes. This can be beneficial on systems where there are multiple processors.


QsoCreateIOCompletionport ()
pthread_ceate ()
IO Completion Port argument
Worker thread
socket ()
QsoWaitForIOCompletion ()
bind ()
QsoStartRecv () completed
listen ()
QsoStartSend ()
accept ()
QsoWaitForIOCompletion ()
QsoStartRecv ()
QsoStartSend () completed
close (listening socket)
Waits for worker thread to end
pthread_join ()
close (connection)
QsoDestroyIOCompletionPort ()
Worker thread ends

Flow of socket events: Asynchronous I/O server

The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and worker examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. This flow describes the socket calls in the following sample application. Use this server example with the generic client example.

1. Master thread creates I/O completion port by calling **QsoCreateIOCompletionPort()**
2. Master thread creates pool of worker thread(s) to process any I/O completion port requests with the **pthread_create** function.
3. Worker thread(s) call **QsoWaitForIOCompletionPort()** which waits for client requests to process.
4. The master thread accepts a client connection and proceeds to issue a **QsoStartRecv()** which specifies the I/O completion port upon which the worker threads are waiting.

   Note: You can also use accept asynchronously by using the **QsoStartAccept()**.
5. At some point, a client request arrives asynchronous to the server process . The sockets operating system loads the supplied user buffer and sends the completed **QsoStartRecv()** request to the specified I/O completion port. One worker thread is awoken and proceeds to process this request.
6. The worker thread extracts the client socket descriptor from the application-defined handle and proceeds to echo the received data back to the client by performing a **QsoStartSend()** operation.
7. If the data can be immediately sent then **QsoStartSend()** returns indication of the fact, otherwise the sockets operating system will send the data as soon as possible and post indication of the fact to the specified I/O completion port. The worker thread gets indication of data sent and can wait on the I/O completion port for another request or terminate if instructed to do so. **QsoPostIOCompletion()** could be used by the master thread to post a worker thread termination event.
8. Master thread waits for worker thread to finish and then destroys the I/O completion port by calling **QsoDestroyIOCompletionPort()**.

Note: This server example works with the common client code describe in Example: Generic client.

This examples shows how a server program can use the asynchronous APIs. For information on the use of code examples, see the code disclaimer.

```
      #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 */
```

Examples: Establish secure connections

You can create secure server and clients using either the Global Secure ToolKit (GSKit) APIs or the SSL_ APIs. GSKit APIs are the preferred method because they provide secure connection across IBM @server platforms. SSL_APIs are native to OS/400 only. Each set of secure sockets APIs have return codes that will help you identify errors when establishing secure socket connections. See Secure socket API error code messages for details on accessing information on these error messages.

The following examples explain how to establish a secure server and client using each of these methods.

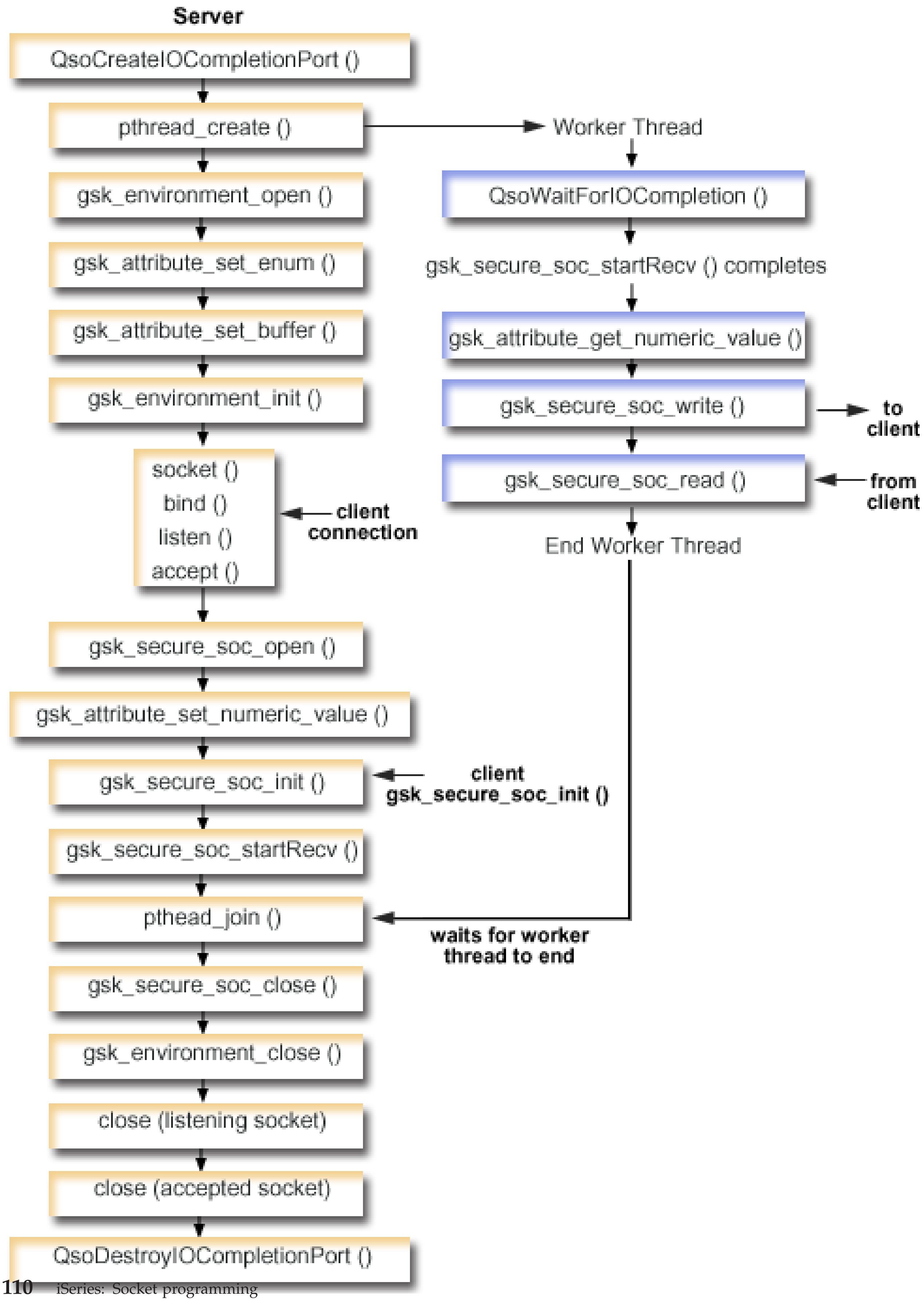
- Example: GSKit secure server with asynchronous data receive
- Example: GSKit secure server with asynchronous handshake
- Example: Establish a secure client with GSKit APIs
- Example: Establish secure server with SSL APIs
- Example: Establish secure client with SSL APIs

Example: GSKit secure server with asynchronous data receive

The following code example can be used to establish a secure server using Global Secure ToolKit (GSKit) APIs. The server opens the socket, prepares the secure environment, accepts and processes connection requests, exchanges data with the client and ends the session. The client also opens a socket, sets up the secure environment, calls the server and requests a secure connection, exchanges data with the server, and closes the session. The following diagram and description shows the server/client flow of events.

Note: The following example programs use AF_INET address family, but they can be modified to also use the AF_INET6 address family.

Socket flow of events: Secure server that uses asynchronous data receive


Server
QsoCreateIOCompletionPort ()
pthread_create ()
Worker Thread
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_enum ()
gsk_secure_soc_startRecv () completes
gsk_attribute_set_buffer ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_write ()
to client
socket ()
bind ()
listen ()
accept ()
client connection
gsk_secure_soc_read ()
from client
End Worker Thread
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
gsk_secure_soc_init ()
client gsk_secure_soc_init ()
gsk_secure_soc_startRecv ()
pthead_join ()
waits for worker thread to end
gsk_secure_soc_close ()
gsk_environment_close ()
close (listening socket)
close (accepted socket)
QsoDestroyIOCompletionPort ()

To view the client portion of this graphic, see Secure GSKit client graphic.

The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between the server and client examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. This flow describes the socket calls in the following sample application. .

1. The **QsoCreateIOCompletionPort()** function creates an I/O completion port.
2. The **pthread_create** function creates a worker thread to receive data and to echo it back to the client. The worker thread will wait for client requests to arrive on the I/O completion port just created.
3. A call to **gsk_environment_open()** to obtain a handle to an SSL environment.
4. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the SSL environment. At a minimum, either a call to **gsk_attribute_set_buffer()** to set the GSK_OS400_APPLICATION_ID value or to set the GSK_KEYRING_FILE value. Only one of these should be set. It is prefered that you use the GSK_OS400_APPLICATION_ID value. Also ensure you set the type of application (client or server), GSK_SESSION_TYPE, using **gsk_attribute_set_enum()**.
5. A call to **gsk_environment_init()** to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that will run using this environment.
6. The **socket** function creates a socket descriptor. The server then issues the standard set of socket calls: **bind()**, **listen()**, and **accept()** to enable a server to accept incoming connection requests.
7. The **gsk_secure_soc_open()** function obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related function calls.
8. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the secure session. At a minimum, a call to **gsk_attribute_set_numeric_value()** to associate a specific socket with this secure session.
9. A call to **gsk_secure_soc_init()** to initiate the SSL handshake negotiation of the cryptographic parameters.

   Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server which the client is connecting to has enabled client authentication. The **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) or **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API calls identify (though in dissimilar ways) the key database file from which the certificate and private key that are used during the handshake are obtained.
10. The **gsk_secure_soc_startRecv()** function initiates an asynchronous receive operation on a secure session.
11. The **pthread_join** synchronizes the server and worker programs. This function waits for the thread to terminate, detaches the thread, then returns the threads exit status to the server.
12. The **gsk_secure_soc_close()** function ends the secure session.
13. The **gsk_environment_close()** function closes the SSL environment.
14. The **close()** function ends the listening socket.
15. The **close()** ends the accepted (client connection) socket.
16. The **QsoDestroyIOCompletionPort()** function destroys the completion port.

Socket flow of events: Worker thread that uses GSKit APIs

1. After the server application creates a worker thread, it waits for server to send it the incoming client request to process client data with the **gsk_secure_soc_startRecv()** call. The **QsoWaitForIOCompletionPort()** function will wait on the supplied IO completion port that was specifed by the server.

2. Once the client request has been recieved, the **gsk_attribute_get_numeric_value()** function gets the socket descriptor associatied with the secure session.
3. The **gsk_secure_soc_write()** function sends the message to the client using the secure session.

For information on the use of code examples, see the code disclaimer.

```
/* 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 */
```

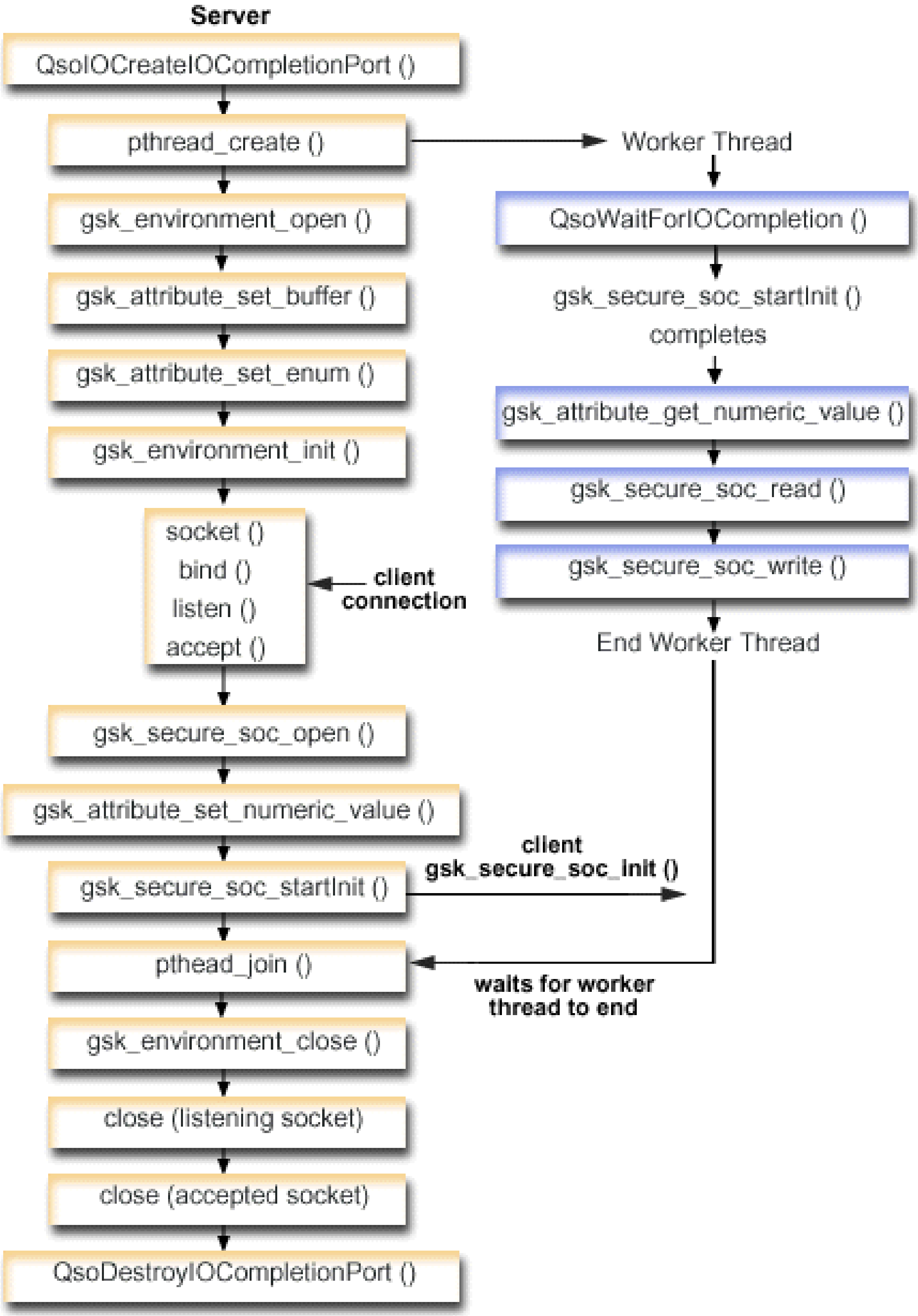
```
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
  perror("bind() failed");
  break;
}

/* 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 */
```

Example: GSKit secure server with asynchronous handshake

New for V5R2, OS/400 sockets introduced the **gsk_secure_soc_startInit()** API. This API allows you to create secure server applications that can handle request asynchronously. The following code sample provides an example of how this API can be used. It is similar to the GSKit secure server with asynchronous data receive example, but uses this new API to start a secure session.

The following graphic shows the function calls used to negotiate an asynchronous handshake on a secure server:


Server
QsoIOCreateIOCompletionPort ()
pthread_create ()
Worker Thread
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_buffer ()
gsk_secure_soc_startInit ()
completes
gsk_attribute_set_enum ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_read ()
socket ()
bind ()
listen ()
accept ()
client
connection
gsk_secure_soc_write ()
End Worker Thread
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
client
gsk_secure_soc_init ()
gsk_secure_soc_startInit ()
pthead_join ()
waits for worker
thread to end
gsk_environment_close ()
close (listening socket)
close (accepted socket)
QsoDestroyIOCompletionPort ()

To view the client portion of this graphic, see GSKit client .

Socket flow of events: GSKit secure server that uses asynchronous handshake

This flow describes the socket calls in the following sample application..

1. The **QsoCreateIOCompletionPort()** function creates an I/O completion port.
2. The **pthread_create** function creates 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.
3. A call to **gsk_environment_open()** to obtain a handle to an SSL environment.
4. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the SSL environment. At a minimum, either a call to **gsk_attribute_set_buffer()** to set the GSK_OS400_APPLICATION_ID value or to set the GSK_KEYRING_FILE value. Only one of these should be set. It is prefered that you use the GSK_OS400_APPLICATION_ID value. Also ensure you set the type of application (client or server), GSK_SESSION_TYPE, using **gsk_attribute_set_enum()**.
5. A call to **gsk_environment_init()** to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that will run using this environment.
6. The **socket** function creates a socket descriptor. The server then issues the standard set of socket calls: **bind()**, **listen()**, and **accept()** to enable a server to accept incoming connection requests.
7. The **gsk_secure_soc_open()** function obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related function calls.
8. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the secure session. At a minimum, a call to **gsk_attribute_set_numeric_value()** to associate a specific socket with this secure session.
9. The **gsk_secure_soc_startInit()** function starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session. Control will return to the program here. When the handshake processing completes, the completion port will be posted with the results. The thread could continue on with other processing; however, for simplicity we have chosen to wait here for worker thread to complete.

   Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server which the client is connecting to has enabled client authentication. The **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) or **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) API calls identify (though in dissimilar ways) the key database file from which the certificate and private key that are used during the handshake are obtained.
10. The **pthread_join** synchronizes the server and worker programs. This function waits for the thread to terminate, detaches the thread, then returns the threads exit status to the server.
11. The **gsk_secure_soc_close()** function ends the secure session.
12. The **gsk_environment_close()** function closes the SSL environment.
13. The **close()** function ends the listening socket.
14. The **close()** ends the accepted (client connection) socket.
15. The **QsoDestroyIOCompletionPort()** function destroys the completion port.

Socket flow of events: Worker thread that process secure asynchronous requests

1. After the server application creates a worker thread, it waits for server to send it the incoming client request to process. The **QsoWaitForIOCompletionPort()** function will wait on the supplied IO completion port that was specifed by the server. This call waits until the **gsk_secure_soc_startInit()** completes.
2. Once the client request has been recieved, the **gsk_attribute_get_numeric_value()** function gets the socket descriptor associatied with the secure session.

3. The **gsk_secure_soc_read()** function receives a message from the client using the secure session.
4. The **gsk_secure_soc_write()** function sends the message to the client using the secure session.

For information on the use of code examples, see the code disclaimer.

```
/* 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");
```

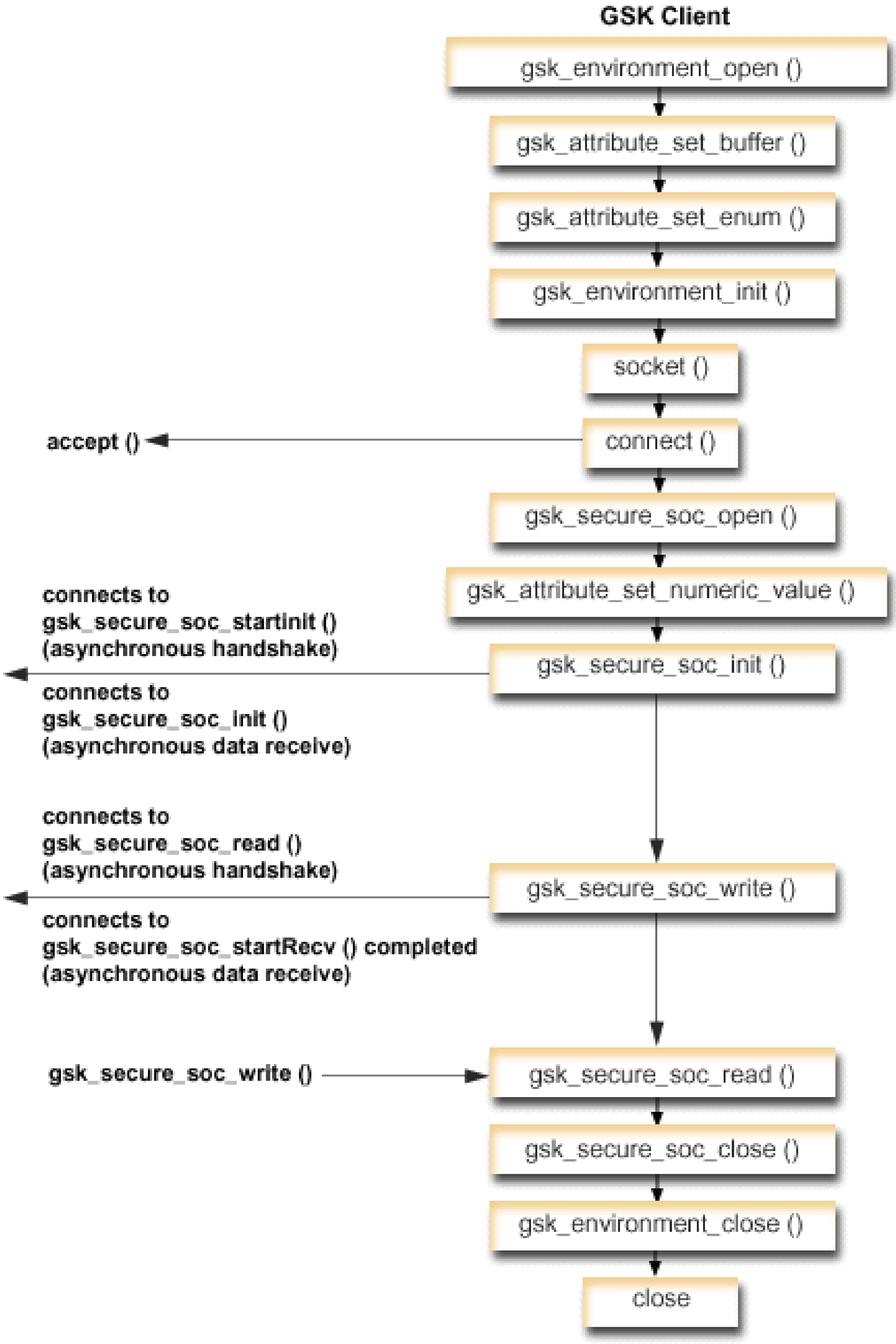
```
listen(lsd, 5);
if (rc < 0)
{
  perror("listen() failed");
  break;
}

/* 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 */
```

Example: Establish a secure client with Global Secure ToolKit (GSKit) APIs

The following code sample provides an example of a client that uses the GSKit APIs. For information on the use of code examples, see the code disclaimer.

The following graphic shows the function calls on a secure client using the GSKit APIs:


GSK Client
gsk_environment_open ()
gsk_attribute_set_buffer ()
gsk_attribute_set_enum ()
gsk_environment_init ()
socket ()
accept ()
connect ()
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
connects to
gsk_secure_soc_startInit ()
(asynchronous handshake)
gsk_secure_soc_init ()
connects to
gsk_secure_soc_init ()
(asynchronous data receive)
connects to
gsk_secure_soc_read ()
(asynchronous handshake)
gsk_secure_soc_write ()
connects to
gsk_secure_soc_startRecv () completed
(asynchronous data receive)
gsk_secure_soc_write ()
gsk_secure_soc_read ()
gsk_secure_soc_close ()
gsk_environment_close ()
close

Socket flow of events: GSKit client

This flow describes the socket calls in the following sample application. Use this client example with the the GSKit server example and the Example: GSKit secure server with asynchronous handshake .

1. The **gsk_environment_open()** function obtains a handle to an SSL environment.
2. One or more calls to gsk_attribute_set_xxxxx() to set attributes of the SSL environment. At a minimum, either a call to **gsk_attribute_set_buffer()** to set the GSK_OS400_APPLICATION_ID value or to set the GSK_KEYRING_FILE value. Only one of these should be set. It is prefered that you use the GSK_OS400_APPLICATION_ID value. Also ensure you set the type of application (client or server), GSK_SESSION_TYPE, using **gsk_attribute_set_enum()**.
3. A call to **gsk_environment_init()** to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that will run using this environment.
4. The **socket** function creates a socket descriptor. The client then issues the **connect()** to connect to the server application.
5. The **gsk_secure_soc_open()** function obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related function calls.
6. The **gsk_attribute_set_numeric_value()** function associates a specific socket with this secure session.
7. The **gsk_secure_soc_init()** function starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session.
8. The **gsk_secure_soc_write()** function writes data on a secure session to the worker thread.

   Note: For the GSKit server example, this function writes data to the worker thread where the **gsk_secure_soc_startRecv()** function completes. In the asynchronous example, it writes to the completed **gsk_secure_soc_startInit()** .
9. The **gsk_secure_soc_read()** function receives a message from the worker thread using the secure session.
10. The **gsk_secure_soc_close()** function ends the secure session.
11. The **gsk_environment_close()** function closes the SSL environment.
12. The **close()** function ends the connection.

```
/* 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 */
```

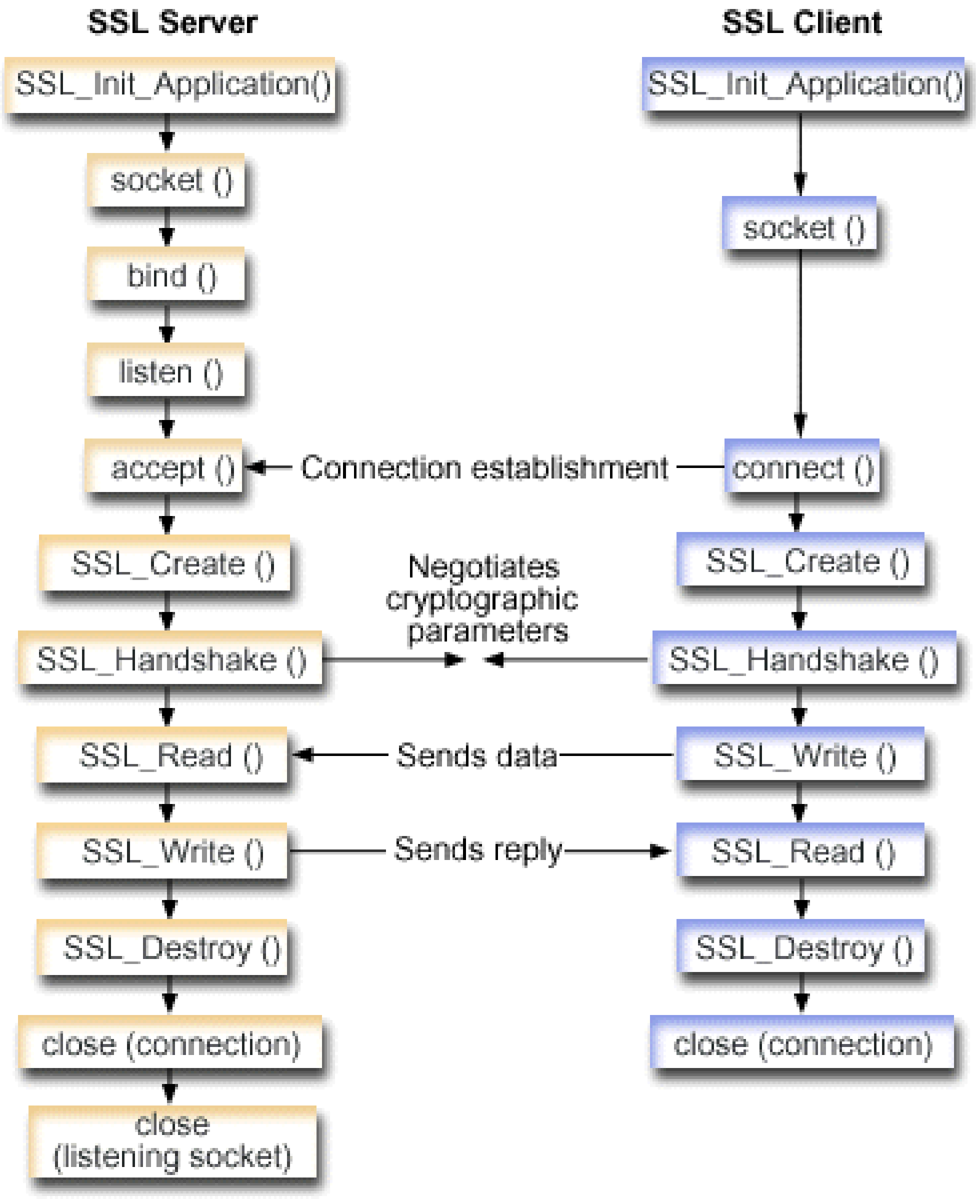
```
struct sockaddr_in address;
int buf_len, rc = 0, sd = -1;
int amtWritten, amtRead;
char buff1[1024];
char buff2[1024];

/*  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;
}
```

Example: Establish a secure server with SSL_ APIs

In addition to creating secure applications using the GSKit APIs, you can also use the SSL_ APIs. These APIs are native to iSeries operating system. As with the GSKit APIs, a server must provide a valid certificate to exchange data securely. The following graphic shows the socket and SSL_ APIs used to create a secure server. If you are writing secure applications across IBM @server platforms, use the GSKit APIs.



Socket flow of events: Secure server that uses SSL_ APIs

The following description shows the relationship between the APIs that enable an SSL server to perform and communicate with an SSL client:

1. Either call **SSL_Init()** or **SSL_Init_Application()** to initialize the job environment for SSL processing and to establish the SSL security information for all SSL sessions that will run in the current job. Only one of these APIs should be used. It is preferred that you use the **SSL_Init_Application()** API.

   Note: The following example program uses the **SSL_Init_Application** API.
2. The server calls **socket()** to obtain a socket descriptor.
3. The server calls **bind()**, **listen()**, and **accept()** to activate a connection for a server program.
4. The server calls **SSL_Create()** to enable SSL support for the connected socket.

5. The server calls **SSL_Handshake()** to initiate the SSL handshake negotiation of the cryptographic parameters.
6. The server calls **SSL_Write()** and **SSL_Read()** to send and receive data.
7. The server calls **SSL_Destroy()** to disable SSL support for the socket.
8. The server calls **close()** to destroy the connected sockets.

Socket flow of events: Secure client that uses SSL_ APIs

1. Either call **SSL_Init()** or **SSL_Init_Application()** to initialize the job environment for SSL processing and to establish the SSL security information for all SSL sessions that will run in the current job. Only one of these APIs should be used. It is preferred that you use the **SSL_Init_Application** API.

   Note: The following example program uses the **SSL_Init_Application** API.
2. The client calls **socket()** to obtain a socket descriptor.
3. The client calls **connect()** to activate a connection for a client program.
4. The client calls **SSL_Create()** to enable SSL support for the connected socket.
5. The client calls **SSL_Handshake()** to initiate the SSL handshake negotiation of the cryptographic parameters.
6. The client calls **SSL_Read()** and **SSL_Write()** to receive and send data.
7. The client calls **SSL_Destroy()** to disable SSL support for the socket.
8. The client calls **close()** to destroy the connected sockets.

Note: The sample uses AF_INET address family; however, it can be modified to use the AF_INET6 address family.

For information on the use of code examples, see the code disclaimer.

```
/* 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;
}
```

Example: Establish a secure client with SSL_ APIs

In addition to the GSKit APIs, OS/400 sockets also supports the traditional SSL_ APIs. These APIs establish a secure connection between a iSeries native server and client applications. For graphic that describes the socket flow of events for this program and its corresponding server application, see Example: Establish a secure server with SSL_ APIs. The following example enables a client application using the SSL_ APIs to communicate with a server application that uses the SSL_APIs:

For information on the use of code examples, see the code disclaimer.

```
/* 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;
}
```

Example: Use gethostbyaddr_r() for threadsafe network routines

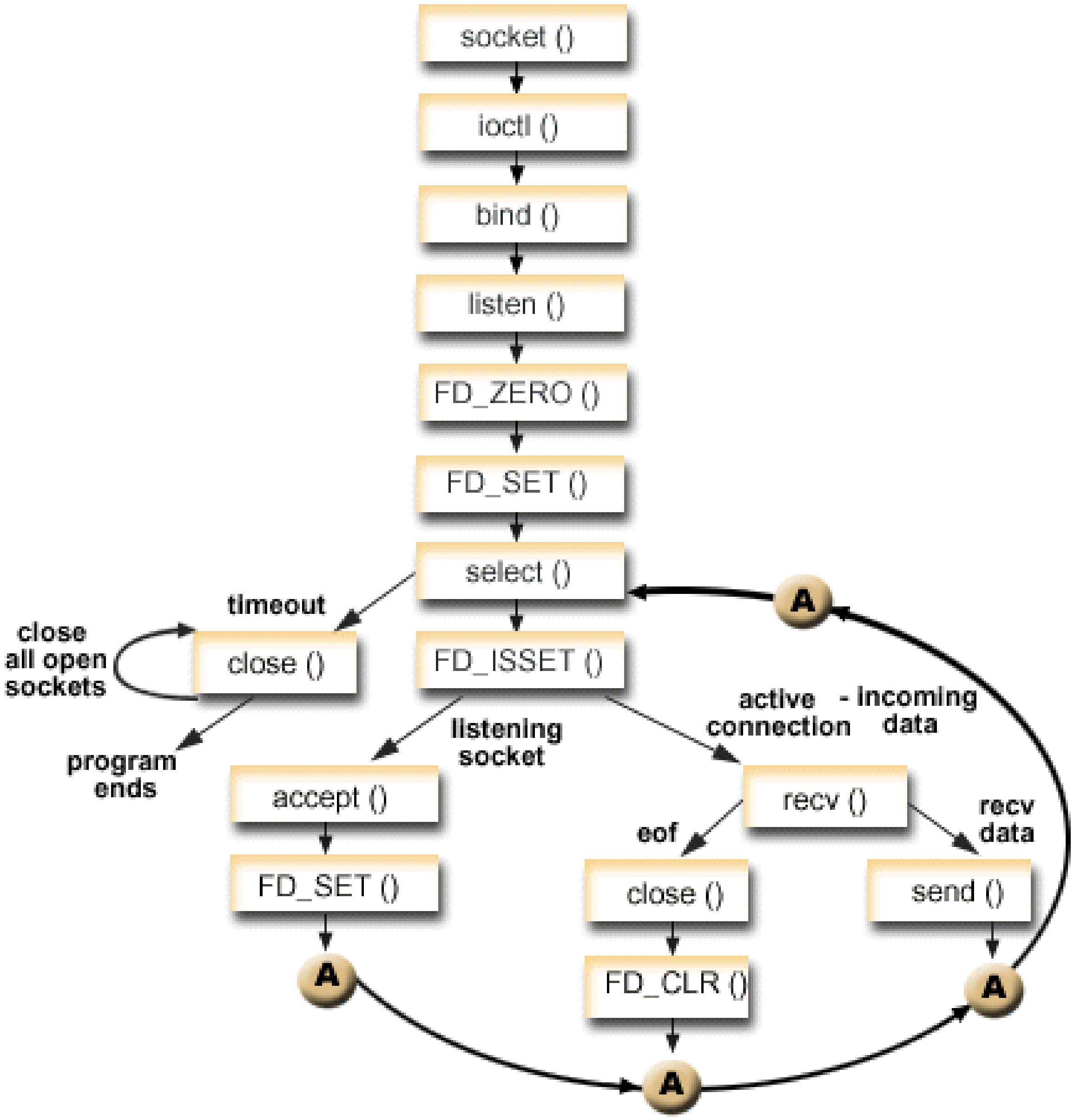
Following is an example of a program that uses **gethostbyaddr_r()**. All other routines with names that end in ″_r″ have similar semantics and are also threadsafe. This example program takes an IP address in the dotted-decimal notation and prints the host name.

For information on the use of code examples, see the code disclaimer.

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

Example: Non-blocking I/O and select()

The following sample program uses non-blocking and the select() API. See Example: Generic client for an example that contains the code for a common client job that can be used with this example.



Socket flow of events: Server that uses non-blocking I/O and select()
The following calls are used in the example:

1. 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.
2. The **ioctl()** function allows the local address to be reused when the server is restarted before the required wait time expires. In this example, it sets the 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.
3. After the socket descriptor is created, the **bind()** function gets a unique name for the socket.
4. The **listen()** allows the server to accept incoming client connections.
5. 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.

6. 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 **select ()** function returns a number that represents the socket descriptors that are ready to be processed.

   0 Indicates that the process will timeout. In this example, the timeout is set for 30 seconds.

   –1 Indicates that the process has failed.

   1 Indicates only one descriptor is ready to be processed. In this example, when a 1 is returned the FD_ISSET and the subsequent socket calls complete only once.

   n Indicates multiple descriptors waiting to be processed. In this example, when an n is returned the FD_ISSET and subsequent code loops and completes the requests in the order they are received by the server.
7. The **accept()** and **recv()** functions are completed when the EWOULDBLOCK is returned.
8. The **send()** function echoes the data back to the client.
9. The **close()** function closes any open socket descriptors.

For information on the use of code examples, see the code disclaimer.

```
#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)
            {
```

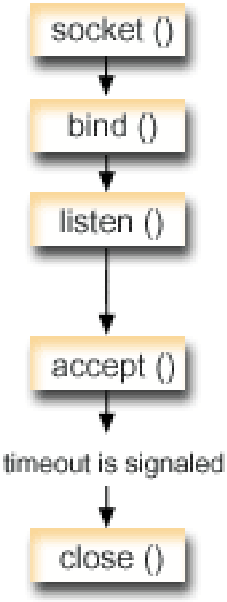
```
         if (errno != EWOULDBLOCK)
         {
            perror("  accept() failed");
            end_server = TRUE;
         }
         break;
      }

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

Example: Use signals with blocking socket APIs

Signals allow you to be notified when a process or application becomes blocked. Signals provide a time limit for blocking processes. In this example the signal occurs after five seconds on the **accept()** call. This call will normally block indefinitely, but because we have an alarm set, it will only block for five seconds. Because blocked programs can hinder performance of an application or server, they can be used to diminish this impact. The following example shows you how to use signals with blocking socket APIs.

Note: Asynchronous I/O used in a threaded server model is preferable over the more conventional model. For more information on advantages of using Asynchronous I/O, see Asynchronous I/O. For a sample program that uses the asynchronous I/O APIs, see Example: Using asynchronous I/O.



Socket flow of events: Use signals with blocking socket

The following sequence of function calls shows how you can use signals to alert the application when the socket has been inactive:

1. 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.
2. After the socket descriptor is created, a **bind()** function gets a unique name for the socket. In this example, a port number is not specified because client application will not connect to this socket. This code snippet can be used within other server programs that uses blocking APIs, such as **accept()**.
3. The **listen()** function indicates a willingness to accept client connection requests. After the **listen()** function is issued an alarm is set to go off in five seconds. This alarm or signal will alert you when the **accept()** call blocks.
4. The **accept()** function accepts a client connection request. This call will normally block indefinitely, but because we have an alarm set, it will only block for five seconds. When the alarm goes off, the accept call will complete with -1 and an errno value of EINTR.
5. The **close()** function ends any open socket descriptors.

For information on the use of code examples, see the code disclaimer.

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

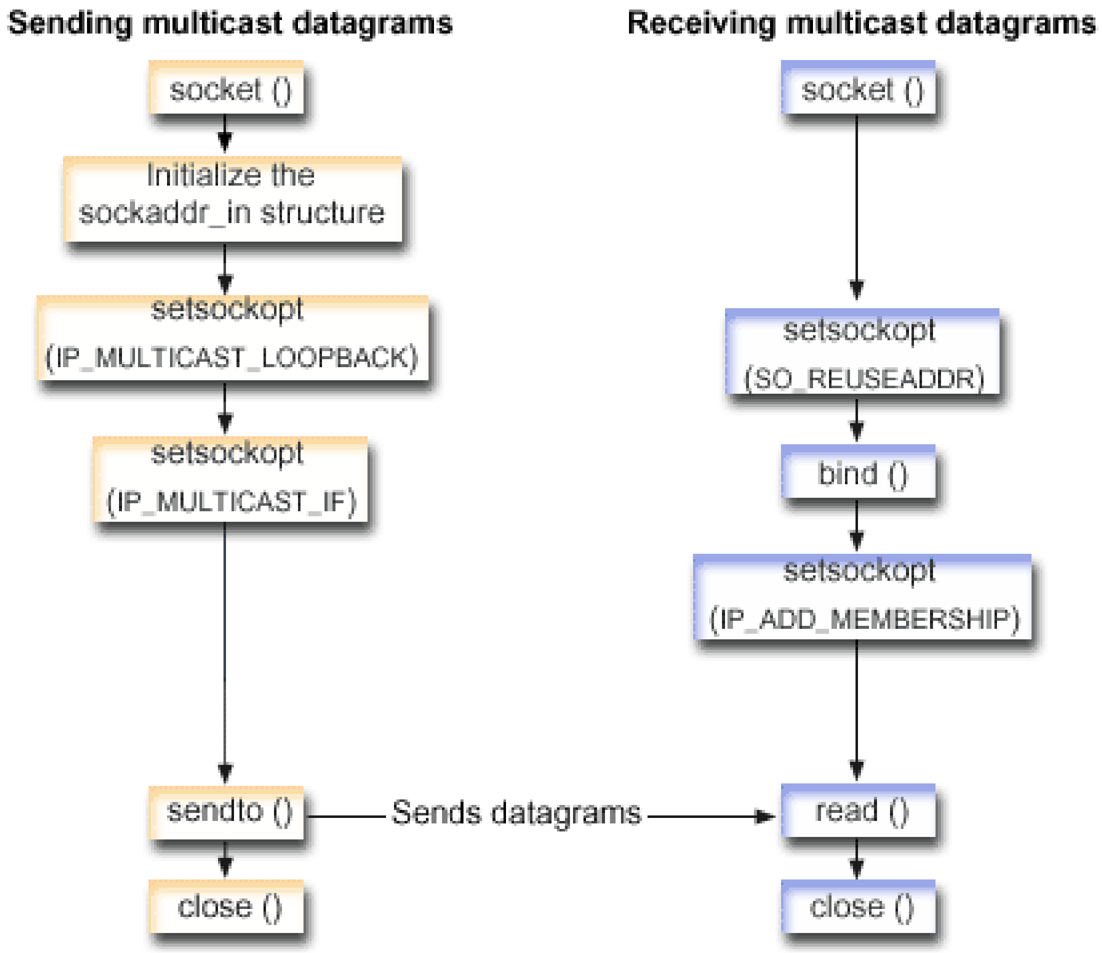
Examples: Use multicasting

IP multicasting provides the capability for an application to send a single IP datagram that a group of hosts in a network can receive. The hosts that are in the group may reside on a single subnet or may be on different subnets that connect multicast-capable routers. Hosts may join and leave groups at any time. There are no restrictions on the location or number of members in a host group. A class D Internet address in the range 224.0.0.1 to 239.255.255.255 identifies a host group.

An application program can send or receive multicast datagrams by using the **socket()** API and connectionless SOCK_DGRAM type sockets. Multicasting is a one-to-many transmission method. You cannot use connection-oriented sockets of type SOCK_STREAM for multicasting. When a socket of type SOCK_DGRAM is created, an application can use the **setsockopt()** function to control the multicast characteristics associated with that socket. The setsockopt() function accepts the following IPPROTO_IP level flags:

- IP_ADD_MEMBERSHIP: Joins the multicast group specified.
- IP_DROP_MEMBERSHIP: Leaves the multicast group specified.
- IP_MULTICAST_IF: Sets the interface over which outgoing multicast datagrams are sent.
- IP_MULTICAST_TTL: Sets the Time To Live (TTL) in the IP header for outgoing multicast datagrams.
- IP_MULTICAST_LOOP: Specifies whether or not a copy of an outgoing multicast datagram is delivered to the sending host as long as it is a member of the multicast group.

Note: OS/400 sockets supports IP multicasting for AF_INET address family.


Sending multicast datagrams
socket ()
Initialize the sockaddr_in structure
setsockopt (IP_MULTICAST_LOOPBACK)
setsockopt (IP_MULTICAST_IF)
sendto ()
close ()
Sends datagrams
Receiving multicast datagrams
socket ()
setsockopt (SO_REUSEADDR)
bind ()
setsockopt (IP_ADD_MEMBERSHIP)
read ()
close ()

Socket flow of events: Sending multicast datagrams
The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between two applications that send and receive multicast datagrams. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The send a multicast datagram uses the following sequence of function calls:

1. 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_DGRAM) will be used for this socket. This socket with send datagrams to another application.
2. The sockaddr_in structure specifies the destination IP address and port number. In this example, the address is 225.1.1.1 and the port number is 5555.
3. The **setsockopt()** function sets the IP_MULTICAST_LOOP socket option so that the sending system will not recieve a copy of the multicast datagrams it transmit.
4. The **setsockopt()** function uses the IP_MULTICAST_IF socket option which defines the local interface over which the multicast datagrams are sent.
5. The **sendto()** function sends multicast datagrams to the specified group IP addresses.
6. The **close()** function closes any open socket descriptors.

Socket flow of events: Receive multicast datagrams
The receive a multicast datagram uses the following sequence of function calls:

1. 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_DGRAM) will be used for this socket. This socket with send datagrams to another application.
2. The **setsockopt()** function sets the SO_REUSEADDR socket option to allow multiple applications to receive datagrams that are destined to the same local port number.
3. The **bind()** function specifies the local port number. In this example, the IP address is specifed as INADDR_ANY to recieve datagrams that are addressed to the multicast group.
4. The **setsockopt()** function uses the IP_ADD_MEMBERSHIP socket option which joins the multicast group that receives the datagrams. When joining a group, specify the class D group address along with the IP address of a local interface. The system must call the IP_ADD_MEMBERSHIP socket option for each local interface receiving the multicast datagrams. In this example, the multicast group (225.1.1.1) is joined on the local 9.5.1.1 interface.

   Note: IP_ADD_MEMBERSHIP option must be called for each local interface over which the multicast datagrams are to be received.
5. The **read()** function reads multicast datagrams that are being sent..
6. The **close()** function closes any open socket descriptors.

Example: Send multicast datagrams

The following example enables a socket to perform the steps listed below and to send multicast datagrams. If you would like to review a description of the socket flow of events for this program, see Examples: Use multicasting .

For information on the use of code examples, see the code disclaimer.

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

Example: Receive multicast datagrams

The following example enables a socket to perform the steps listed below and to receive multicast datagrams:If you would like to review a description of the socket flow of events for this program, see Examples: Use multicasting .

For information on the use of code examples, see the code disclaimer.

```
#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);
  }

}
```

Example: Update and query DNS

The following example show how to query and update Domain Name System (DNS) records.

For information on the use of code examples, see the code disclaimer.

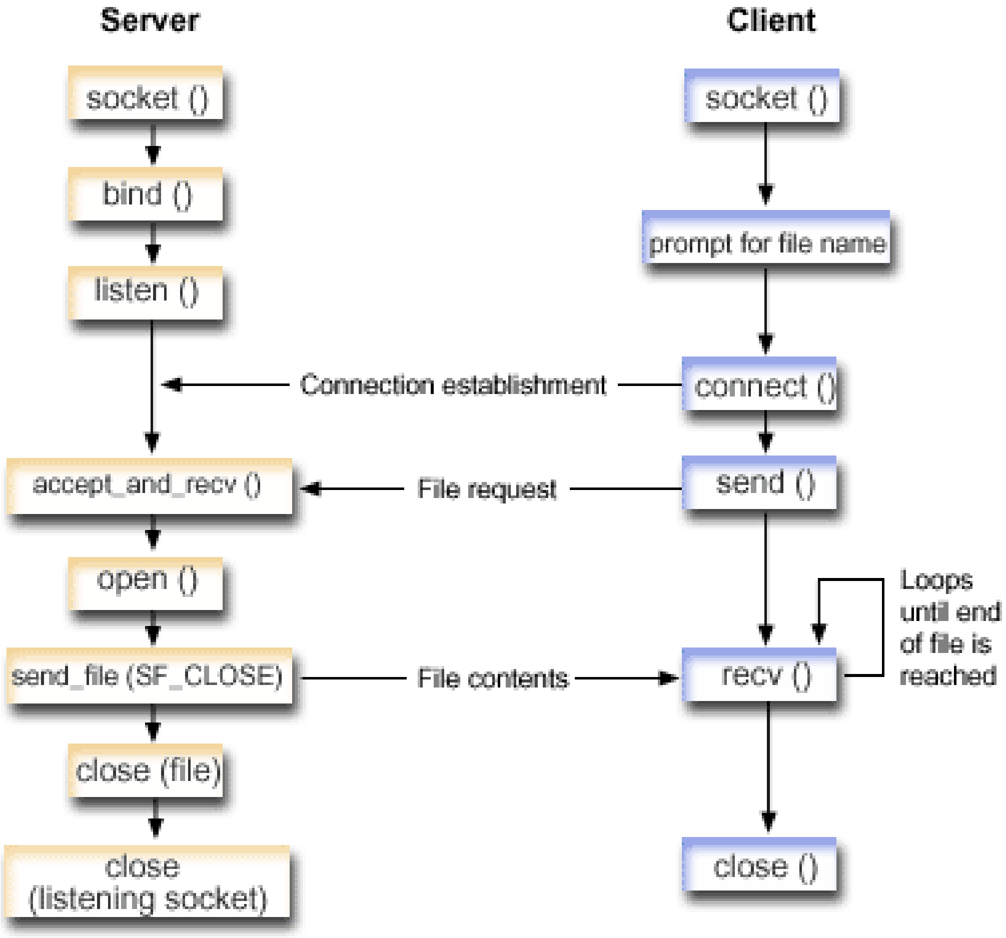
```
/**************************************************************************/
/* This program updates a DNS using a transaction signature (TSIG) to     */
/* sign the update packet.  It then queries the DNS to verify success.    */
/**************************************************************************/

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

Example: Transfer file data using send_file() and accept_and_recv() APIs

The following examples enable a server to communicate with a client by using the **send_file()** and **accept_and_recv()** APIs.



Socket flow of events: Server send contents of a file
The following sequence of the socket calls provide a description of the graphic. It also describes the relationship between two applications that send and receive files. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The Example: Use accept_and_recv() and send_file() APIs to send contents of a file uses the following sequence of function calls:

1. The server calls **socket()**, **bind()**, and **listen()** to create a listening socket.
2. The server initializes the local and remote address structures.
3. The server calls **accept_and_recv()** to wait for an incoming connection and to wait for the first data buffer to arrive over this connection. This call returns the number of bytes that is received and the local and remote addresses that are associated with this connection. This call is a combination of the **accept()**, **getsockname()**, and **recv()** APIs.
4. The server calls **open()** to open the file whose name was obtained as data on the **accept_and_recv()** from the client application.
5. The **memset()** function is used to set all of the fields of the sf_parms structure to an initial value of 0. The server sets the file descriptor field to the value that **open()** returned. The server then sets the file bytes field to -1 to indicate that the server should send the entire file. The system is sending the entire file, so you do not need to assign the file offset field.
6. The server calls **send_file()** to transmit the contents of the file. **send_file()** does not complete until the entire file has been sent or an interrupt occurs. **send_file()** is more efficient because the application does not have to go into a **read()** and **send()** loop until the file finishes.

7. The server specifies the SF_CLOSE flag on the **send_file()** API. The SF_CLOSE flag informs the **send_file()** API that it should automatically close the socket connection when the last byte of the file and the trailer buffer (if specified) have been sent successfully. The application does not need to call **close()** if the SF_CLOSE flag is specified.

Socket flow of events: Client request for file
The Example: Client request for a file uses the following sequence of function calls:

1. This client program takes from zero to two parameters.

   The first parameter (if specified) is the dotted-decimal IP address or the host name where the server application is located.

   The second parameter (if specified) is the name of the file that the client attempts to obtain from the server. A server application sends the contents of the specified file to the client. If the user does not specify any parameters, then the client uses INADDR_ANY for the server's IP address. If the user does not specify a second parameter, the program prompts the user to enter a file name.
2. The client calls **socket()** to create a socket descriptor.
3. The client calls **connect()** to establish a connection to the server. Step one obtained the IP address of the server.
4. The client calls **send()** to inform the server what file name it wants to obtain. Step one obtained the name of the file.
5. The client goes into a "do" loop calling **recv()** until the end of the file is reached. A return code of 0 on the **recv()** means that the server closed the connection.
6. The client calls **close()** to close the socket.

Example: Use accept_and_recv() and send_file() APIs to send contents of a file

The following example enables a server to perform the steps listed below to communicate with a client by using the **send_file()** and **accept_and_recv()** APIs.

For information on the use of code examples, see the code disclaimer.

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

Example: Client request for a file

The following example enables a client to request a file from the server and to wait for the server to send the contents of that file back.

For information on the use of code examples, see the code disclaimer.

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

Xsockets tool

The Xsocket tool is one of many tools that are shipped with the iSeries. All tools are stored in the QUSRTOOL library. Xsockets allows programmers to interactively work with socket APIs. The Xsocket tool allows you to do the following tasks:

- learn about the socket APIs
- recreate specific scenarios interactively to help debug

Note: Xsockets tool is shipped in an ″as-is″ format.

Prerequisites for using Xsockets

- The ILE C/400® language is installed.
- The License Program 5722–SS1 ′Openness Include′ is installed.
- The License Program 5722–DG1 ′IBM HTTP Server′ is installed.

  Note: This is needed if you plan to use Xsockets in a web browser.
- The License Program 5722–JV1 ′Developer Kit for Java′ is installed.

  Note: This is needed if you plan to use Xsockets in a web browser.

Use the following topics to install and use the Xsocket tool:

Configure Xsockets
This topic explains how to create the Xsockets tool, which helps you design and compile your socket programs.

Use Xsockets
This topic explains how to use the Xsockets tool.

Customize Xsockets
This topic explains how to customize the Xsockets tool.

Configure Xsockets

There are two versions of the tool that can be created. The first version is native iSeries client. The native version is completely created by the first set of instructions. The second version uses a web browser as the client. If you want to use the web browser client, you must complete native setup instructions for the native version first.

To create the Xsockets tool, complete the following steps:

1. To unpackage the tool, enter
   ```
   CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)
   ```
   on a command line.

   Note: There must be 10 characters between the openning ′ and closing′.
2. To add the QUSRTOOL library to your library list, enter
   ```
   ADDLIBLE QUSRTOOL
   ```
   on a command line.
3. Create a library in which to create the Xsocket program files by entering
   ```
   CRTLIB <library-name>
   ```
   on a command line. The `<library-name>` is the the library in which you want the Xsocket tool objects created. For example,
   ```
   CRTLIB MYXSOCKET
   ```
   would be a valid library name.

   Note: Do not add Xsocket tool objects to the QUSRTOOL library. This could interfere with the use of other tools within that directory.
4. To add this library to the library list, enter
   ```
   ADDLIBLE <library-name>
   ```
   on the command line. The`<library-name>` is the the library that you created in Step 3. For example, we used MYXSOCKET as the library name, then we would enter:
   ```
   ADDLIBLE MYXSOCKET
   ```
5. Create the install program TSOCRT that will automatically install the Xsockets tool by entering:
   ```
   CRTCLPGM <library-name>/TSOCRT QUSRTOOL/QATTCL
   ```
   on the command line. The`<library-name>` is the the library that you created in Step 3. For example, we used MYXSOCKET as the library name, then we would enter:
   ```
   CRTCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
   ```
6. To call the install program, enter:
   ```
   CALL TSOCRT library-name
   ```
   on the command line. In the place of library-name, use the library you created in Step 3. For example, to create the tool in the MYXSOCKET library, enter:
   ```
   CALL TSOCRT MYXSOCKET
   ```

Note: This may take a few minutes to complete.

If you do not have job control (*JOBCTL) special authority when you call TSOCRT to create the sockets tool the **givedescriptor()** socket function will return errors when an attempt is made to pass a descriptor to a job which is not the one you are running.

TSOCRT will create a CL program, an ILE C/400 program (2 modules are created), 2 ILE C/400 service programs (2 modules are created), and three display files.Whenever you want to use the tool, you should add the library to your library list. All objects created by the tool will have a name that is prefixed by TSO.

What to do next:

Use native Xsockets
If you want to use Xsocket natively, you can go to this step. This topic covers the basics on using the native Xsockets tool.

Note: The native version does not support GSKit secure socket APIs. If you want to write socket programs that use these APIs, you should use the browser-based version of the tool.

Configure Xsockets to use a web browser

Note: This step is optional.

What is created by native Xsocket setup

Below is a table listing the objects created by the install program. All the objects that are created will reside in the specified library.

Table 20. Objects created during Xsocket install

Object name	Member name	Source file name	Object type	Extension	Description
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	Module that is used for interfacing between JSP and TSOSTSOC
TSODLT	TSODLT	QATTCL	*PGM	CLP	CL program to delete the tool objects and/or source file members.
TSOXSOCK	N/A	N/A	*PGM	C	Main program that is used for the SOCKETS interactive tool.
TSOXGJOB	N/A	N/A	*SRVPGM	C	Service program that is used in support of the SOCKETS interactive tool

Table 20. Objects created during Xsocket install (continued)

TSOJNI	N/A	N/A	*SRVPGM	C	Service program that is used for interfacing between JSP and TSOSTSOC in support of the SOCKETS interactive tool.
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXSOCK program. The source file contains the main() routine.
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXSOCK program. The source file contains the routines that actually invoke the socket functions.
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXGJOB service program. The source file contains the routine that identifies the internal job. This internal job identifier is made up of the job name, user ID, and job number.
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	Display file used by the Xsockets tool for the main window that contains the sockets functions.
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	Display file used by the XSockets tool in support of the various socket functions.

Table 20. Objects created during Xsocket install (continued)

TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	Display file used by the Xsockets tool that supports the menu bar.
QATTIFS2	N/A	N/A	*FILE	PF-DTA	Contains the JAR file used by Tomcat Server.

Configure Xsockets to use a web browser

The following set of instructions allow your to enable the Xsockets tool to be accessed through a web browser. You can implement these instructions multiple times on the same system to create different server instances. This allows multiple versions to run at the same time on different listening ports. To configure Xsockets to use a web browser, you must complete the following tasks:

1. Configure HTTP server (powered by Apache)
2. Configure Tomcat
3. Update configuration files
4. Test Xsockets tool in browser

Configure HTTP server (powered by Apache)

Before you can configure a web browser to work with the Xsockets tool, you must first complete native Xsockets configuration. The following steps configure a HTTP server (powered by Apache) so you can use the Xsockets tool in a web browser.

1. Verify the HTTP admin instance is running under the QHTTPSVR subsystem. You can start it with CL command it not running:
   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```
2. In a web browser, enter :
   ```
   http://<system_name>:2001/.
   ```

   where <system_name>is the machine name of the iSeries. For example: http://myiSeries:2001/.
3. On the iSeries Tasks page, select **IBM HTTP Server for iSeries**.
4. From the top menu, select the **Setup** tab.
5. Click **Create New HTTP Server**.
6. Select **HTTP server (powered by Apache)**, and click **Next**.
7. Enter the name for the server instance. For example, since this instance will serve the Xsocket tool in a browser, you could use the name `xsocket`. Click **Next**.
8. Select **No**. This will create a new server instance that is not based on an existing server. Click **Next**.
9. Click **Next** to accept the default for the server root directory.
10. Click **Next** to accept the default for the document root directory.
11. Select the IP address and an available port that you would like to use. Use a port number greater than 1024. Click **Next**.

    Note: Do not select the default port number 80.
12. Choose either **yes** or **no** to indicated whether or not you want an access log created for this server. Click **Next**.
13. The next page displays the HTTP server (powered by Apache) configuration settings. If these settings are correct, click **Finish**.
14. Click **Manage newly created server**. You are now done with Apache configuration.

What to do next:

Configure Tomcat

Configure Tomcat

After you have configured HTTP server (powered by Apache) server instance, you must configure Tomcat to launch the Xsockets tool in a web browser.

1. Under the **Dynamic content** heading, select **ASF Tomcat Setup Server Task** .
2. Select **Enable servlets for this HTTP Server**. This will fill in a Workers definition file for you. Click **Next**.
3. Accept the defaults on the **Workers Definition** page by clicking **Next**.
4. On the **URL to Worker Mapping** page, click **Add**.
5. In the **URL(Mount Point)** column, enter `/xsock`. Click **Continue**.
6. Click **Add.**
7. In the **URL(Mount Point)** column, enter `/xsock/*`. Click **Continue**.
8. Click **Next**.
9. On the **In-Process Application Context Definition** page, click **Add.**
10. In the **URL Path** column, enter `/xsock`.
11. In the **Application Base Directory** column, enter `webapps/xsock`.
12. Click **Continue**. You will see a warning message that you need to configure more information.
13. Click **Configure** under **Configure Application**.
14. In the new browser window that opens, select 3 days in the **Session Object timeout** field.

    Note: This is the recommended value; however, you can specify another value for the **Session Object timeout**.
15. Click **Add** to add Servlet definition and complete the following steps:
    a. For **Servlet class name**, enter `com.ibm.iseries.xsocket.XSocketServlet`.
    b. For **URL patterns**, enter `/*`.
    c. Set the **Startup load sequence** to 3.
    d. Click **Continue**.
    e. Click **OK**. This will close the browser window.
16. In the main Tomcat setup window, click **Next**.
17. Click **Finish**.
18. Click **OK**. You have now completed Tomcat configuration for the Xsockets tool.

What to do next:

Update configuration files

Update configuration files

After you have configured Tomcat on the HTTP server (powered by Apache) to server the Xsockets tool you must complete manual changes to several configuration files for the instance. There are three files that you needed to update: the web.xml file, the JAR file, and the httpd.conf. To complete these steps you will need to know the following information:

- The library name that contains the Xsockets application files. You created this file during initial Xsockets configuration for a native client.
- The server name that you created during HTTP server (powered by Apache) configuration.

1. **Update web.xml file**
   a. From a command line, type

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

where `<server_name>` is the name of the server instance you created during Apache configuration. For example, if you chose `xsocks` for the server name, you would enter:

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

b. Press 2 to edit the file.

c. Find the `</servlet-class>` line in the web.xml file.

d. Insert the following code after this line:

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

In place of the XXXX, insert the library name that you created during Xsockets configuration.

e. Save the file and exit the edit session.

2. **Move JAR file**

a. From a command line, enter this command:

```
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)
```

where XXXX is the library name that you created during Xsockets configuration and <server_name> is the name of the server instance you created during HTTP server (powered by Apache) configuration.

3. **Add authority check to httpd.conf file** (This step is optional.)
This will force Apache to authenticate users trying to access the Xsockets web application.

Note: It is also necessary for getting write access to create UNIX sockets.

a. From a command line, type

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

where <server_name> is the name of the server instance you created during Apache configuration. For example, if you chose xsocks for the server name, you would enter:

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

b. Press 2 to edit the file.

c. Insert the following lines at the end of the file.

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

d. Save the file and exit the edit session.

What to do next

Test Xsockets tool in web browser

Test Xsockets tool in web browser

After you have completed manual updates to the configuration files, you are ready to test the Xsocket tool within a browser.

1. To start the server instance, enter this command on a command line:

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

where <server_name> is the name of the server instance you created during Apache configuration.

Note: This takes awhile.

2. Check on its status by issuing WRKACTJOB command from the command line interface.. If all jobs with your server_name have SIGW status, then you can precede to next step.
3. In a browser, enter the following URL:

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

   where <system_name> and <port> are the server instance name and port number that you chose during Apache configuration.
4. When prompted, enter your username and password for the server. The web client for Xsocket should appear.

Use Xsockets

There are now two ways to work with the Xsockets tool. You can work with the tool from the native client or work with the tool in a web browser. To work with a native version of Xsocket, you must configure the Xsockets tool. In addition to configuring the Xsockets tool for a native client, you must also complete the steps in Configure Xsockets to use a web browser if you prefer to work with the tool in a browser environment. Many of the concepts are similar between the two versions of the tools. Both tools allow you to issue socket calls interactively and both tools provide errnos for issued socket calls; however, the interfaces do have some differences. The following sets of instructions show you how to work with the Xsocket tool in both environments.

Note: If you want to work with socket programs that use the GSKit secure socket APIs, you must use the web version of the tool.

The following sets of instruction describes how to work with each of these tools:

- Use native Xsockets
- Use browser-based Xsockets

Use native Xsockets

Ensure that you have completed all the configuration steps for native Xsockets before completing these steps. To use Xsocket on a native client, complete these steps:

1. At a command line, add the library in which the Xsocket tool exist to your library lists by issuing this command

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

   where the <library-name> is the name of the library you created during native Xsockets configuration. For example, if the name of the library is MYXSOCKET, then enter:

   ```
   ADDLIBLE MYXSOCKET
   ```
2. On a command line interface, type

   ```
   CALL TSOXSOCK
   ```
3. In the Xsocket window that displays allows you to access all socket routines through its menu bar and selection field. This window always displays after you choose a socket function. You can use this interface to select socket programs that already exists. To work with a new socket, complete the following:
   a. In the list of socket functions, select **socket** and press **Enter**.
   b. In the **socket() prompt** window that displays, select the appropriate Address Family, Socket Type, and Protocol for the socket, and press **Enter**.
   c. Select **Descriptor** and select **Select descriptor**.

      Note: If other socket descriptors already exist, this will display a list of active socket descriptors.

d. From the list that displays, select the socket descriptor that you created

   Note: If other socket descriptors exist, the tool will automatically apply a socket function to the latest socket descriptor.

4. From the list of socket functions, select a socket function with which you would like to work. Whatever socket descriptor you chose in Step 3c will be used on that socket function. Once you select a socket function, a series of windows display that you can provide specific information on the socket function. For example, if you select **connect()**, you will need to provide the address length, address family, and address data in the resulting windows. The socket function chosen is then called with this information that you provided. Any errors that occur on a socket function will be displayed back to the user as an errno.

Notes:

1. The Xsockets tool uses the graphical look support for DDS.Thus, how data is entered and how selections are made from the windows/panels you see will be dependent on whether you are using a graphical display station or a nongraphical display station. For example, on a graphical display station, you will see the selection field for the socket functions as a check box; otherwise, you might see a single field.
2. Be aware that there are **ioctl()** requests that are available on a socket which have not been implemented in the tool.

Use Xsockets in a web browser

Ensure that you have completed all the native configuration of Xsockets and all the necessaryweb browser configuration before working with the Xsockets tool in a web browser. To work with the Xsockets tool in a web browser, complete these steps:

1. In a web browser, type:

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

   where server-name is the name of the iSeries that contains the server instance.
2. Select **Administration**.
3. From the left navigation, select **Manage HTTP Servers**.
4. Select your instance name, and click **Start**. You can also start the server instance from a command line by entering:

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

   where <instance_name> is the name of your HTTP server created in the Apache configuration. For example, you could use the server instance name xsocks.
5. To access the Xsockets web application, enter this URL in a browser:

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

   where <system_name> is the machine name of the iSeries and <port> is the port specified when you created the HTTP instance. For example, if the system name is myiSeries and the HTTP server instance listens on port 1025, you would enter:

   ```
   http://myiSeries:1025/xsock/index
   ```
6. Once the Xsocket tool loads in the web browser, you can work with existing socket descriptor or create new onces. Many of the concepts are similar between the two versions of the tools Both tools allow you to issue socket calls interactively and both tools provide errnos for issued socket calls, however the interfaces do have some differences. To create a new socket descriptor, you can do the following:
   a. From the **Xsocket Menu**, select **socket**.
   b. In the **Xsocket Query** window that displays, select the appropriate Address Family, Socket Type, and Protocol for this socket discriptor. Click **Submit**.
   c. Once the page reloads, the new socket descriptor will be displayed in the **Socket** pull-down menu.

d. From the **Xsocket Menu**, select function calls that you would like applied to this socket descriptor. As with the native version of the Xsockets tool, the tool will automatically apply function calls to the latest socket descriptor if you do not select a socket descriptor.

Delete objects created by the Xsocket tool

You may need to delete objects that are created by the Xsockets tool. The program named TSODLT is created by the install program to remove the objects created by the tool (except the library and the program TSODLT) and/or remove the source members used by the Xsocket tool. The following set of commands allow you to delete these objects:

To delete ONLY the source members used by the tool, enter the following command :

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

To delete ONLY objects that the tool creates, enter the following command:.

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

To delete BOTH source members and objects created by the tool, enter the following command:

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

Customize Xsockets

You can change the Xsockets tool by adding additional support for the socket network routines, for example **inet_addr()**. If you choose to customize this tool to meet your own needs, we recommend that you do not make changes in the QUSRTOOL library. Instead, copy the source files into a separate library and make the changes there. This will preserve the original files in the QUSRTOOL library so they will be available if needed in the future. You may use the TSOCRT program to recompile the tool after making your changes (note that if the source files are copied to a separate library, you will also need to make changes in TSOCRT to use it). Use the TSODLT program to remove old versions of the tool objects before creating the tool.

Serviceability tools

As the use of sockets and secure sockets continues to grow to accommodate e-business applications and servers, the current serviceability tools need to keep up with this demand. Enhanced serviceability tools allow you to complete traces on socket programs to find solutions to errors within socket and SSL-enabled applications. These tools help you and support center personnel to determine where socket problems are by selecting socket traits, such as IP address or port information.

The following table gives an overview for the each of these service tools.

Table 21. Serviceability tools for Socket and Secure sockets

Serviceability tool	Description
LIC Trace Filtering (TRCINT and TRCCNN)	Provides selective trace on sockets. You can now restrict sockets trace on address family, socket type, protocol, IP address, and port information. You can also limit traces to only certain categories of socket APIs and also to only those sockets that have the SO_DEBUG socket option set. New for V5R2, a LIC trace can now be filtered by thread, task, user profile, job name, or server name.

Table 21. Serviceability tools for Socket and Secure sockets (continued)

Trace job with STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR))	New for V5R2, STRTRC command provides additional parameters which generates output that is separated from all other non-socket related trace points. This output contains return code and errno information when an error is encountered during a socket operation. See the STRTRC (Start Trace) Command Description in the Information Center for details.
Flight recorder tracing	Sockets LIC component traces will now include a dump of the flight recorder entries for each socket operation performed.
Associated job information	Allows service personnel and programmers to find all jobs that are associated to a connected or listening socket. This information can be viewed using NETSTAT for those socket applications using an address family of AF_INET or AF_INET6.
NETSTAT Connection Status(option ′3′) to enable SO_DEBUG	Provides enhanced low-level debug information when the SO_DEBUG socket option is set on a socket application.
Secure socket return code and message processing	Presents standardized secure socket return code messages through two SSL_ APIs. These APIs are **SSL_Strerror()** and **SSL_Perror()**. In addition, the **gsk_strerror()** provides similar function for GSKit APIs. There is also the **hstrerror()** API that provides return code information from resolver routines.
Performance Data Collection (PDC) tracepoints	Provides a trace for the data flow from an application through Sockets and the TCP/IP stack.

Related information

Listed below are IBM Redbooks™ (in PDF format), Web sites, and Information Center topics that provide more information socket programming. You can view or print any of the PDFs.

IBM Redbooks

- Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More
- IBM @server iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements

Request For Comments

- **IPv6**
  - RFC 2553: ″Basic Socket Interface Extensions for IPv6″
  - RFC 2292: ″Advanced Sockets API for IPv6″
- **Domain Name System**
  - 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)″
- **Secure Sockets Layer/Transport Layer Security**
  - RFC 2246: ″The TLS Protocol Version 1.0 ″

Other Web Resources

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

Code disclaimer information

This document contains programming examples.

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.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

© Copyright IBM Corp. 2001, 2005

Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (IBM) (2004). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2001, 2004. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Anynet
C/400
IBM
iSeries
Language Environment
Operating System/400
OS/400
Redbooks

Microsoft®, Windows®, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Terms and conditions for downloading and printing publications

Permissions for the use of the publications you have selected for download are granted subject to the following terms and conditions and your indication of acceptance thereof.

Personal Use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these Publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

All material copyrighted by IBM Corporation.

By downloading or printing a publication from this site, you have indicated your agreement with these terms and conditions.

Printed in USA