Socket Programming

1. Introduction

In the classic client-server model, the client sends out requests to the server, and the server does some processing with the request(s) received, and returns a reply (or replies) to the client. The terms request and reply here may take on different meanings depending upon the context, and method of operation. An example of a simple and ubiquitous client-server application would be that of a Web-server. A client (Internet Explorer, or Netscape) sends out a request for a particular web page, and the web-server (which may be geographically distant, often in a different continent!) receives and processes this request, and sends out a reply, which in this case, is the web page that was requested. The web page is then displayed on the browser (client).

Further, servers may be broadly classified into two types based on the way they serve requests from clients. Iterative servers can serve only one client at a time. If two or more clients send in their requests at the same time, one of them has to wait until the other client has received service. On the other hand, concurrent servers can serve multiple clients at the same time. Typically, this is done by spawning off a new server process on the receipt of a request – the original process goes back to listening to new connections, and the newly spawned off process serves the request received.

We can realize the client-server communication described above with a set of network protocols, like the TCP/IP protocol suite, for instance. The lower-level details, like the operation of the protocol itself must be clear to you by now. In this tutorial, we will look at the issue of developing applications for realizing such communication over a network. In order to write such applications, we need to understand sockets.

2. What are sockets?

Sockets (also called Berkeley Sockets, owing to their origin) can simply be defined as end-points for communication. To provide a rather crude visualization, we could imagine the client and server hosts in Figure 1 being connected by a pipe through which data-flow takes place, and each end of the pipe can now be construed as an “end-point”. Thus, a socket provides us with an abstraction – or a logical end point for communication. There are different types of sockets. Stream sockets, of type SOCK_STREAM are used for connection oriented, TCP connections, whereas datagram sockets of type SOCK_DGRAM are used for UDP based applications. Apart from these two, there are other socket types defined, like SOCK_RAW and SOCK_SEQPACKET.
2.1 Socket layer, and the Berkeley Socket API

Figure 2 shows the TCP/IP protocol stack, and shows where the “Socket layer” may be placed. Again, please be advised that this is just a representation to indicate the level at which we operate when we write network programs using sockets. As shown in the figure, sockets make use of the lower level network protocols, and provide the application developer with an interface to the lower level network protocols. A library of system calls are provided by the socket layer, and are termed as the “Socket API”. These system calls can be used in writing socket programs. In the sections that ensue, we will study those system calls in detail.

![Figure 2. TCP/IP protocol stack](image)

2.2 Programming environment, and the WinSock library

We will study socket programming on the windows environment (Windows 9x/NT/2000/XP). Socket programming on UNIX based environments is similar, and most of the system calls used are the same. Windows adds its own windows-specific extensions. Windows provides the WinSock library which contains the system calls used for socket programming on the windows environment. Compilation issues on windows machines is relegated to Appendix-I.

3 Basic Socket system calls

Figure 3 shows the sequence of system calls between a client and server for a connection-oriented protocol.

3.1 The `socket()` system call

The `socket()` system call creates a socket and returns a handle to the socket created. The handle is of data type SOCKET. In UNIX based environments, the socket descriptor is a non-negative integer, but Windows uses the SOCKET data type for the same.
Socket handles may take any value in the range 0 to INVALID_SOCKET - 1.

**socket() system call syntax**

Figure 3. Socket system calls for connection-oriented case
Here, *af* is the Address family specification, *type* is the socket type and the protocol field is used to specify the protocol to be used with the address family specified. The address family can be one of AF_INET (for Internet protocols like TCP, UDP) or AF_UNIX (for Unix internal protocols), AF_NS, for Xerox network protocols or AF_IMPLINK for the IMP link layer. The type field is the socket type – which may be SOCK_STREAM for stream sockets (TCP connections), or SOCK_DGRAM (for datagram connections). Other socket types are defined too. SOCK_RAW is used for raw sockets, and SOCK_SEQPACKET is used for a sequenced packet socket. The protocol argument is typically set to 0. You may also specify a protocol argument to use a specific protocol for your application.

```c
SOCKET sd = socket(...);
if (sd == INVALID_SOCKET) /* error condition */
    {...
```

### 3.2 The **bind()** system call

The **bind()** system call is used to specify the association <Local-Address—Local-Port>. It is used to bind either connection oriented or connectionless sockets. The **bind()** function basically associates a name to an unnamed socket. “name”, here refers to three components – The address family, the host address, and the port number at which the application will provide its service. The syntax and arguments taken by the **bind** system call is given below:

#### **bind()** syntax

```c
int result = bind (SOCKET sd, const struct sockaddr* name, int namelen);
```

Here, *sd* is the SOCKET handle returned by the **socket()** system call before, and *name* points to the SOCKADDR structure, and *namelen* is the length of the parameter *name* in bytes.

```c
int res = bind (...);
if (res == SOCKET_ERROR) /* error condition */
    {...
```
Upon success, bind() returns 0. In case of error, bind() returns SOCKET_ERROR. So the return value of the bind system call must be checked against SOCKET_ERROR to check if the call succeeded.

3.3 The listen() system call

After creation of the socket, and binding to a local port, the server has to wait on incoming connection requests. The listen() system call is used to specify the queue or backlog of waiting (or incomplete) connections. The syntax and arguments taken by the listen system call is given below:

**listen() syntax**

```c
int result = listen (SOCKET sd, int backlog);
```

Here, `sd` is the SOCKET handle returned by the socket() system call before, and `backlog` is the number of incoming connections that can be queued.

**listen system call**

```c
int result = listen (...);
if (result == SOCKET_ERROR) /* error condition */
   {...}
```

Upon success, listen() returns 0. In case of error, listen() returns SOCKET_ERROR.

3.4 The accept() system call

After executing the listen() system call, a server waits for incoming connections. An actual connection setup is completed by a call to accept(). accept() takes the first connection request on the queue, and creates another socket with the same properties as `sd` (the socket descriptor returned earlier by the socket() system call). In case of blocking sockets, this call would block the caller until a connection request arrives. accept() assumes a concurrent server, and automatically creates a new socket descriptor for the connection request received. In case of concurrent servers, a new server process is spawned (or “fork”ed in UNIX terminology), and the newly created server process serves this connection on this socket. The earlier socket goes back to listening on new connections. In a sense, the accept() system call completes the connection, and at the end of a successful accept(), all elements of the four tuple (or the five tuple – if you consider “protocol” as one of the elements) of a connection are filled. The “four-tuple” that we talk about here is <Local Addr, Local Port, Remote Addr, Remote Port>. This combination of fields is used to uniquely identify a flow or a connection. The fifth tuple
element can be the protocol field. No two connections can have the same values for all the four (or five) fields of the tuple.

**accept() syntax**

`SOCKET newsd = accept (SOCKET sd, struct sockaddr* addr, int* addrlen);`

Here, `sd` is the `SOCKET` handle returned by the `socket()` system call before, and `addr` is a pointer to a buffer that receives the address of the connecting entity, and `addrlen` is the length of the parameter `addr`.

**accept system call**

`SOCKET newsd = accept (...);`

If `(newsd == INVALID_SOCKET) /* error condition */`

{...}

Upon success, `accept()` returns a handle to the new socket created. In case of error, `accept()` returns `SOCKET_ERROR`.

**3.5 The connect() system call**

A client process also starts out by creating a socket by calling the `socket()` system call. It uses `connect()` to connect that socket descriptor to establish a connection with a server. In the case of a connection oriented protocol (like TCP/IP), the `connect()` system call results in the actual connection establishment of a connection between the two hosts. In case of TCP, following this call, the three-way handshake to establish a connection is completed. Note that the client does not have to bind to a local port in order to call `connect()`. Clients typically choose ephemeral port numbers for their end of the connection. Servers, on the other hand, have to provide service on well-known (premeditated) port numbers.

**connect() syntax**

`int result = connect (SOCKET sd, const struct sockaddr* servaddr, int addrlen);`

Here, `sd` is the `SOCKET` handle returned by the `socket()` system call before, `servaddr` is a pointer to the server’s address structure. `addrlen` holds the length of this parameter.

**connect system call**

`int result = connect (...);`

If `(result == SOCKET_ERROR) /* error condition */`

{...}

Upon success, `connect()` returns 0. In case of error, `connect()` returns `SOCKET_ERROR`. 
3.6 send(), recv(), sendto() and recvfrom() system calls

After connection establishment, data is exchanged between the server and client using the system calls send(), recv(), sendto() and recvfrom(). The syntax of the system calls are as below:

send(), recv(), sendto() and recvfrom() syntax

```c
int nbytes = send (SOCKET sd, const char* buf, int len, int flags);  
int nbytes = recv (SOCKET sd, const char* buf, int len, int flags);  
int nbytes = sendto (SOCKET sd, const char* buf, int len, int flags,  
                     struct sockaddr* to, int tolen);  
int nbytes = recvfrom (SOCKET sd, const char* buf, int len, int flags,  
                       struct sockaddr* from, int fromlen);
```

Here, sd is the SOCKET handle returned by the socket() system call before, buf is the char buffer to be sent or received, flags is the indicator specifying the way the call is to be made. sendto() and recvfrom() are used in case of connectionless sockets, and they do the same function as send() and recv() except that they take more arguments (the “to” and “from” addresses – as the socket is connectionless)

Upon success, all these calls return the number of bytes written or read. Upon failure, all the above calls return SOCKET_ERROR

3.7 The closesocket() system call

The closesocket() system call is used to close the connection. In some cases, any remaining data that is queued is sent out before the closesocket() is executed.

closesocket() syntax

```c
int result = closesocket (SOCKET sd);
```

Here, sd is the SOCKET handle returned by the socket() system call before.

closesocket system call

```c
int result = closesocket (...);
if (result == SOCKET_ERROR) /* error condition */  
{...}
```

Upon success, closesocket() returns 0. In case of error, closesocket() returns SOCKET_ERROR
3.8 The shutdown() system call

The shutdown() system call is used to disable sends or receives on a socket.

shutdown() syntax

```c
int result = shutdown (SOCKET sd, int how);
```

Here, `sd` is the SOCKET handle returned by the socket() system call before. The parameter `how` determines how the shutdown is achieved. If the value of the parameter `how` is `SD_RECEIVE`, further receives on the socket will be disallowed. If `how` is `SD_SEND`, further sends are disallowed, and finally, if `how` is `SD_BOTH`, both sends and receives on the socket are disallowed. Remember that the shutdown() function does not close the socket. The socket is closed and all the associated resources are freed only after a closesocket() call.

shutdown system call

```c
int result = shutdown (...);
if (result == SOCKET_ERROR) /* error condition */
   {...}
```

Upon success, shutdown() returns 0. In case of error, shutdown() returns SOCKET_ERROR.

4 Byte ordering, and byte ordering routines

When data is transmitted on networks, the byte ordering of data becomes an issue. There are predominantly two kinds of byte orderings – Little-Endian byte ordering, and Big-Endian byte ordering. Different processor architectures use different kinds of byte orderings. For example, the x86 family of processors uses Little-Endian byte ordering, and the Motorola 68K machines use Big-Endian byte ordering. Because of disparities like the above, when data is transmitted over a network, we need to change the byte ordering to the “Network byte order”, and change it back to “Host byte order” when data is read in.

The following routines help in changing the byte order of the data.

```c
u_short result = ntohs (u_short netshort);
```

```c
u_short result = htons (u_short hostshort);
```

```c
u_long result = ntohl (u_long netlong);
```

```c
u_long result = htonl (u_long hostlong);
```

The htonl* routines convert host-byte-order to the network-byte-order and return the value in TCP/IP network byte order. The ntoh* routines do the opposite.
5 Important structs

This section has definitions for the structs used in the socket system calls.

```
struct sockaddr {
    unsigned short   sa_family;    /* address family, AF_xxx */
    char              sa_data[14];  /* 14 bytes of protocol address */
};
```

The `sockaddr` structure holds the socket address information for all types of sockets. The `sa_family` field can point to many address families. Refer from section 3.1 that the Internet address family is denoted by AF_INET, and that encompasses most of the popular protocols we use (TCP/UDP), and so the Internet address specific parallel `sockaddr` structure is called `sockaddr_in`. The fields are self-explanatory. The `sin_zero` field serves as padding, and is typically set to all zeroes using `memset()`.

```
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```
Appendix: Compilation instructions

The code provided for both the EchoServer and EchoClient are written for the Windows platform, using the Winsock Library. You can use any compiler that provides the windows libraries, like the Microsoft VC++ compiler, or the CodeWarrior IDE. The usage of the IDE itself is clearly explained in the tutorials that come with CodeWarrior or MSVC++. In either case, you need to open a new project, and add the given source files to the project, and use the build option for getting the executables for both the Client and Server.