Introduction to Socket Programming

Almost all Winsock functions operate on a socket, as it is our handle to the connection. Both sides of the connection use a socket, and they are not platform-specific (i.e. a Windows and UNIX machine can talk to each other using sockets). Sockets are also two-way; data can be both sent and received on a socket.

There are two common types for socket, one is a streaming socket (SOCK_STREAM), the other is a datagram socket (SOCK_DGRAM). The streaming variant is designed for applications that need a reliable connection, often using continuous streams of data. The protocol used for this type of socket is TCP.

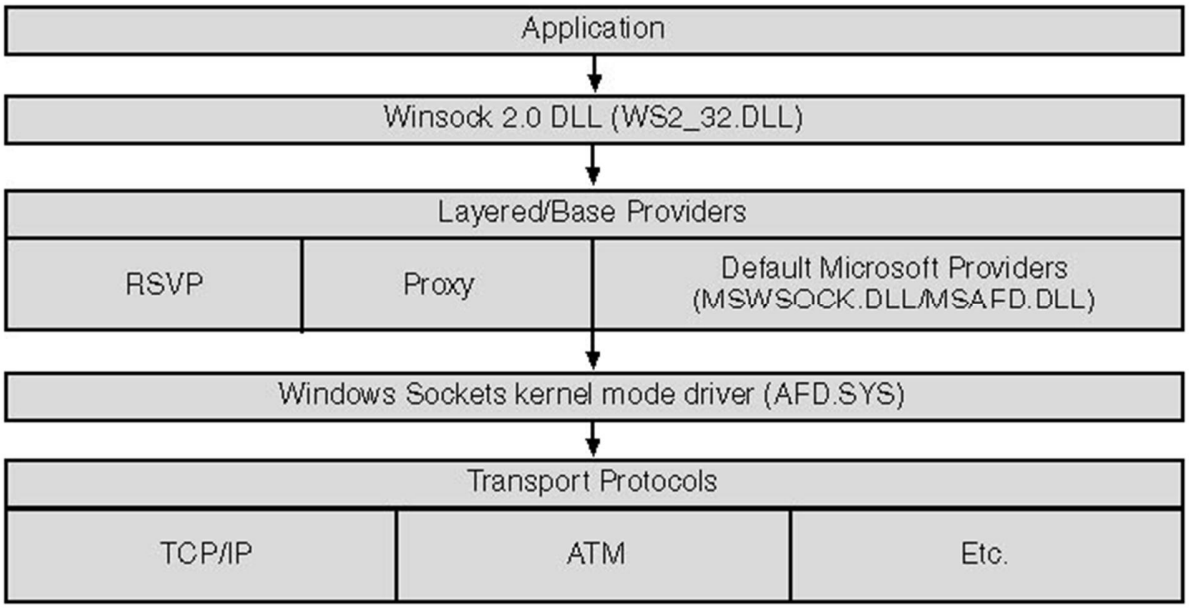
Datagram sockets use UDP as underlying protocol, are connectionless, and have a maximum buffer size. They are intended for applications that send data in small packages and that do not require perfect reliability. Unlike streaming sockets, datagram sockets do not guarantee data will reach its destination or that it comes in the right order. Datagram sockets can be slightly faster and useful for applications like streaming audio or video, where reliability is not as high on the priority list as speed and latency. Where the reliability is required, streaming sockets are used.

Winsock is a standard application programming interface (API) that allows two or more applications (or processes) to communicate either on the same machine or across a network and is primarily designed to foster data communication over a network. It should be noted that Winsock is a network programming interface and not a protocol. Winsock provides the programming interface for applications to communicate using popular network protocols such as Transmission Control Protocol/Internet Protocol (TCP/IP) and Internetwork Packet Exchange (IPX). The Winsock interface inherits a great deal from the BSD Sockets implementation on UNIX platforms. In Windows environments, the interface has evolved into a truly protocol-independent interface, especially with the release of Winsock 2.

Winsock is available in two major versions—Winsock 1 and Winsock 2—on all Windows platforms except Windows CE (Windows CE, e.g. Windows mobile, has only Winsock 1). When developing new applications we should target the Winsock 2 specification by including WINSOCK2.H in our application. For compatibility with older Winsock applications and when developing on Windows CE platforms, WINSOCK.H is available. There is also an additional header file: MSWSOCK.H, which targets Microsoft-specific programming extensions that are normally used for developing high performance Winsock applications.

The Winsock 2 Service Provider Interface (SPI) is the complement to the Winsock. As the name implies, the SPI is a service to applications and is not an application. It is written and exposes itself to applications that can load the service either knowingly or unknowingly. The SPI is a part of the Winsock 2 specification and therefore requires the Winsock 2 update if running in Windows 95. The following figure illustrates the relationship between Winsock applications and the SPI.

There are two parts to the SPI: transport service providers and name space providers. Each part provides distinctly different functionalities. There are two types of transport service providers: layered and base. A layered service provider installs itself into the Winsock catalog above base providers and possibly between other layered providers and intercepts Winsock API calls from applications. A base provider exposes a Winsock interface that directly implements a protocol such as the Microsoft TCP/IP provider.



SPI Architecture

When compiling our application with WINSOCK2.H, we should link with WS2_32.LIB library. When using WINSOCK.H (as on Windows CE) we should use WSOCK32.LIB. If we use extension APIs from MSWSOCK.H, we must also link with MSWSOCK.DLL. Once we have included the necessary header files and link environment, we are ready to begin coding our application, which requires initializing Winsock.

Windows Socket Programming in C

Now that the environment is ready, let's see the necessary steps in creating a working network application. Every Winsock application must load the appropriate version of the Winsock DLL. If we fail to load the Winsock library before calling a Winsock function, the function returns a `SOCKET_ERROR`; the error that we get after the call to `WSAGetLastError` will be `WSANOTINITIALISED`. Loading the Winsock library is accomplished by calling the `WSAStartup` function, which is defined as

```
int WSAStartup(
    WORD wVersionRequested,
    LPWSADATA lpWSAData
);
```

The wVersionRequested parameter is used to specify the version of the Winsock library we want to load. The high-order byte specifies the minor version of the requested Winsock library, while the low-order byte is the major version. We can use the handy macro MAKEWORD(x, y), in which x is the high byte and y is the low byte, to obtain the correct value for wVersionRequested. The following example demonstrates how we initialize a specific version of the Winsock library.

```
WSADATA wsadata;

if (WSAStartup(MAKEWORD(2, 0), &wsadata) != 0)
{
  printf("WSA Initialization failed.");
}
```

Note that after a successful call to WSAStartup, the wsadata variable will be filled with information about the loaded library.

When our application is completely finished using the Winsock interface, we should call WSACleanup, which allows Winsock to free up any resources allocated by Winsock and cancel any pending Winsock calls that our application made. WSACleanup is defined as

```
int WSACleanup(void);
```

Failure to call WSACleanup when our application exits is not harmful because the operating system will free up resources automatically. However, our application will not be following the Winsock specification. Also, we should call WSACleanup for each call that is made to WSAStartup.

IPv4 Issues

In IPv4, computers are assigned an address that is represented as a 32-bit quantity. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Also, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP addresses and service port information through the SOCKADDR_IN structure, which is defined as

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

The sin_family field must be set to AF_INET, which tells Winsock we are using the IP address family.

The sin_port field defines which TCP or UDP communication port will be used to identify a server service. Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services, such as File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP).

The sin_addr field of the SOCKADDR_IN structure is used for storing an IPv4 address as a four-byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or a remote IP address. IP addresses are normally specified in Internet standard dotted notation as "a.b.c.d." Each letter represents a number (in decimal, octal, or hexadecimal format) for each byte and is assigned, from left to right, to the four bytes of the unsigned long integer. The final field, sin_zero, functions only as padding to make the SOCKADDR_IN structure the same size as the SOCKADDR structure.

A useful support function named inet_addr converts a dotted IP address to a 32-bit unsigned long integer quantity. The inet_addr function is defined as

```
unsigned long inet_addr(
    const char FAR *cp
);
```

The cp field is a null-terminated character string that accepts an IP address in dotted notation. Note that this function returns an IP address as a 32-bit unsigned long integer in network-byte order.

Byte Ordering

Different computer processors represent numbers in big-endian and little-endian form, depending on how they are designed. For example, on Intel x86 processors, multi-byte numbers are represented in little-endian form: the bytes are ordered from least significant to most significant. When an IP address and port number are specified as multi-byte quantities in a computer, they are represented in host-byte order. However, when IP addresses and port numbers are specified over a network, Internet networking standards specify that multi-byte values must be represented in big-endian form (most significant byte to least significant), normally referred to as network-byte order.

A series of functions can be used to convert a multi-byte number from host-byte order to network-byte order and vice versa. The following four API functions convert a number from host-byte to network-byte order:

```
u_long htonl(u_long hostlong);

int WSAHtonl(SOCKET s, u_long hostlong, u_long FAR * lpnetlong);

u_short htons(u_short hostshort);

int WSAHtons(SOCKET s, u_short hostshort, u_short FAR * lpnetshort);
```

The next four functions are the opposite of the preceding four functions; they convert network-byte order to host-byte order.

```
u_long ntohl(u_long netlong);

int WSANtohl(SOCKET s, u_long netlong, u_long FAR * lphostlong);

u_short ntohs(u_short netshort);

int WSANtohs(SOCKET s, u_short netshort, u_short FAR * lphostshort);
```

The following code demonstrates how to address IPv4 by creating a SOCKADDR_IN structure using the `inet_addr` and `htons` functions described previously.

```
SOCKADDR_IN addr;
int nPortId = 80;

addr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a four-byte integer, and assign it to sin_addr

addr.sin_addr.s_addr = inet_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert
// nPortId to network-byte order, and assign it to sin_port.

addr.sin_port = htons(nPortId);
```

Creating Sockets

The prototype for the `socket` function is as follows:

```
SOCKET socket(int af, int type, int protocol);
```

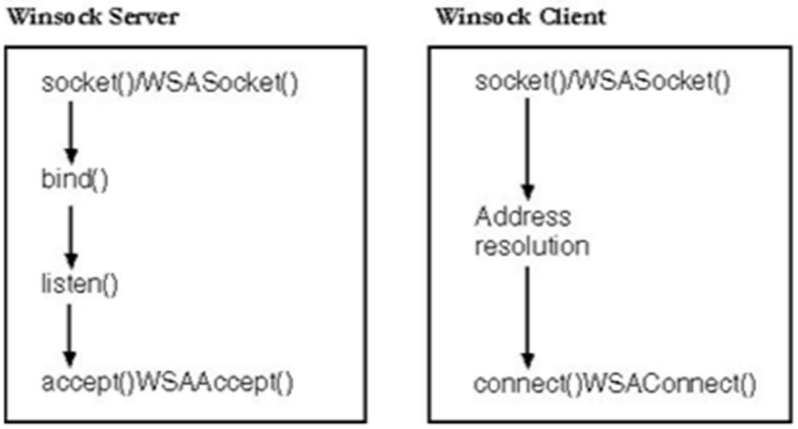
The first parameter, `af`, is the protocol's address family. Since we are working on IPv4 protocol, we should set this field to AF_INET. The second parameter, `type`, is the protocol's socket type. When we are creating a socket to use TCP/IP, we set this field to SOCK_STREAM,

and for UDP/IP we use SOCK_DGRAM. The third parameter is `protocol` and is used to qualify a specific transport if there are multiple entries for the given address family and socket type. For TCP we should set this field to IPPROTO_TCP; for UDP we use IPPROTO_UDP.

Unbound Socket

Now that a socket is created, based on the protocol the socket was created, we can either of the following: connect to a remote host (TCP), listen a connection on a specific port (TCP) or send data to a remote host without connecting (UDP). As an example, let's create a Winsock client/server application.

A server is a process that waits for any number of client connections with the purpose of servicing their requests. A server must listen for connections on a well-known name. In TCP/IP, this name is the IP address of the local interface and a port number. Every protocol has a different addressing scheme and therefore a different naming method. The first step in Winsock is to create a socket with the `socket` call and bind the socket of the given protocol to its well-known name, which is accomplished with the `bind` API call. The next step is to put the socket into listening mode, which is performed (appropriately enough) with the `listen` API function. Finally, when a client attempts a connection, the server must accept the connection with the `accept` call.



Client/Server Architecture

Binding

Once the socket of a particular protocol is created, we must bind it to a well-known address. The `bind` function associates the given socket with a well-known address. This function is declared as

```
int bind(
    SOCKET                      s,
    const struct sockaddr FAR* name,
    int                         namelen
);
```

The first parameter, `s`, is the socket on which we want to wait for client connections. The second parameter is of type `struct sockaddr`, which is simply a generic buffer. We must actually fill out an address buffer specific to the protocol we are using and cast that as a `struct sockaddr` when calling bind. The third parameter is simply the size of the protocol-

specific address structure being passed. The following code snippet creates a TCP socket and binds it to a well-known address (IP + port).

```
SOCKET              s;
SOCKADDR_IN         addr; // == struct sockaddr addr;
int                 port = 80;

s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&addr, sizeof(addr));
```

From the code snippet, we'll see a stream socket being created, followed by setting up the TCP/IP address structure on which client connections will be accepted. In this case, the socket is being bound to the default IP interface by using a special address, `INADDR_ANY` (which mean `0.0.0.0`), and occupies port number `80`. We could have specified an explicit IP address available on the system, but `INADDR_ANY` allows us to bind to all available interfaces on the system so that any incoming client connection on any interface (but the correct port) will be accepted by our listening socket. The call to `bind` formally establishes this association of the socket with the local IP interface and port.

On error, `bind` returns `SOCKET_ERROR`. The most common error encountered with `bind` is `WSAEADDRINUSE`. With TCP/IP, the `WSAEADDRINUSE` error indicates that another process is already bound to the local IP interface and port number or that the IP interface and port number are in the `TIME_WAIT` state. If we call `bind` again on a socket that is already bound, `WSAEFAULT` will be returned.

Bound Socket

Listening

The next step is to put the socket into listening mode. The `bind` function merely associates the socket with a given address. The API function that tells a socket to wait for incoming connections is `listen`, which is defined as

```
int listen(
    SOCKET s,
    int    backlog
```

```
);
```

Again, the first parameter is a bound socket. The `backlog` parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For example, let's say the `backlog` parameter is set to two. If three client requests are made at the same time, the first two will be placed in a "pending" queue so that the application can service their requests. The third connection request will fail with `WSAECONNREFUSED`. Note that once the server accepts a connection, the request is removed from the queue so that others can make a request. The `backlog` parameter is silently limited to a value that the underlying protocol provider determines. Illegal values are replaced with their nearest legal values. The following code snippet makes the socket created above listen on the specified port.

```
listen(s, 5); // where 5 is the maximum pending connections
```

The errors associated with `listen` are fairly straightforward. By far the most common is `WSAEINVAL`, which usually indicates that we forgot to call `bind` before `listen`.

Listening Socket

Accepting

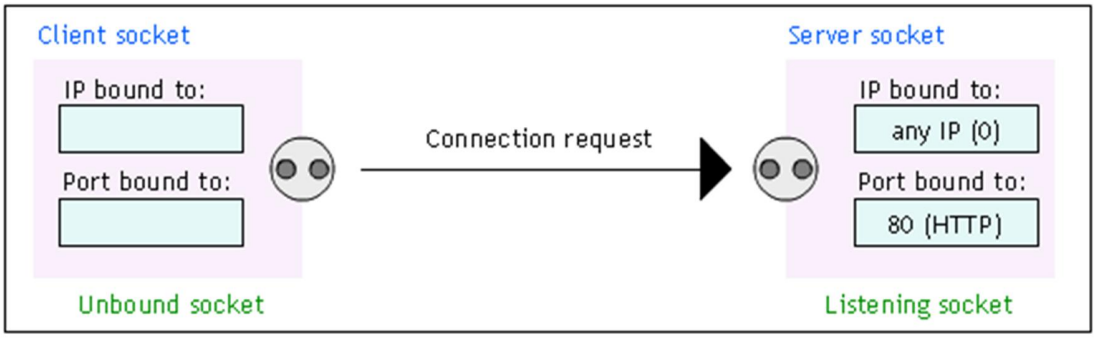
Now we're ready to accept client connections. This is accomplished with the `accept` function. The prototype for `accept` is

```
SOCKET accept(
    SOCKET s,
    struct sockaddr FAR* addr,
    int FAR* addrlen
);
```

Parameter `s` is the bound socket that is in a listening state. The second parameter should be the address of a valid `SOCKADDR_IN` structure, while `addrlen` should be a reference to the length of the `SOCKADDR_IN` structure. For a socket of another protocol, we substitute the `SOCKADDR_IN` with the `SOCKADDR` structure corresponding to that protocol. A call to `accept` services the first connection request in the queue of pending connections. When the `accept` function returns, the `addr` structure contains the IPv4 address information of the client making the connection request, and the `addrlen` parameter indicates the size of

the structure. In addition, `accept` returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still open to accept other client connections and is still in listening mode.

If an error occurs, `INVALID_SOCKET` is returned. The most common error encountered is `WSAEWOULDBLOCK` if the listening socket is in asynchronous or non-blocking mode and there is no connection to be accepted.



Client Attempting to Connect on the Listening port (80)

The following program fragment demonstrates a simple server that accepts one TCP/IP connection.

```
#include <winsock2.h>

void main(void)
{
   WSADATA              wsadata;
   SOCKET               sock;
   SOCKET               new_sock;
   SOCKADDR_IN          server_addr;
   SOCKADDR_IN          cleit_addr;
   int                  port = 80;

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsadata);

   // Create a new socket to listen for client connections.

   sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Set up a SOCKADDR_IN structure that will tell bind that we
   // want to listen for connections on all interfaces using port
   // 80. Notice how we convert the port variable from host byte
   // order to network byte order.

   server_addr.sin_family = AF_INET;
   server_addr.sin_port = htons(Port);
   server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

   // Associate the address information with the socket using bind.

   bind(sock, (SOCKADDR *)&server_addr, sizeof(server_addr));

   // Listen for client connections. We used a backlog of 5, which
   // is normal for many applications.

   listen(sock, 5);

   // Accept a new connection when one arrives.

   new_sock = accept(sock, (SOCKADDR*)&client_addr,& client_addr));

   // At this point we can do two things with these sockets. Wait
   // for more connections by calling accept again on sock
   // and start sending or receiving data on new_sock.

   // When we are finished sending and receiving data on the
   // new_sock socket and are finished accepting new connections
   // on sock, we should close the sockets using the
   // closesocket API.

   closesocket(new_sock);
   closesocket(sock);

   // When our application is finished handling the connections,
   // we call WSACleanup.

   WSACleanup();
}
```

Note that the above fragment does not involve error handling. Now that we have seen the server side of client/server architecture, let's now see the client side.

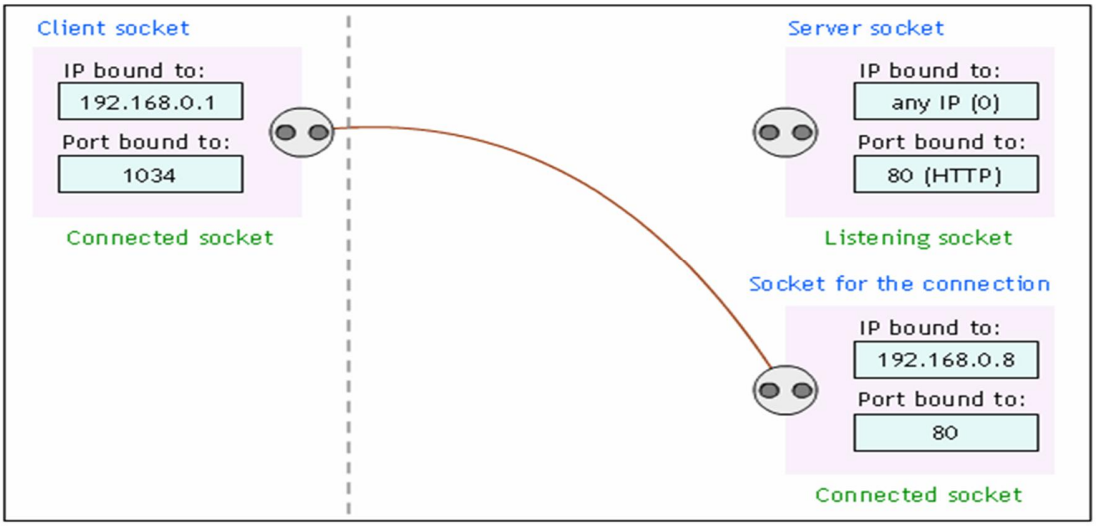
Connecting

Connecting a socket is accomplished by calling the `connect` function. The prototype for `connect` is as follows

```
int connect(
    SOCKET s,
    const struct sockaddr FAR* name,
    int namelen
);
```

The parameters are fairly self-explanatory: `s` is the valid TCP socket on which to establish the connection, `name` is the socket address structure (`SOCKADDR_IN`) for TCP that describes the server to connect to, and `namelen` is the length of the `name` variable.

If the computer we're attempting to connect to does not have a process listening on the given port, the `connect` call fails with the `WSAECONNREFUSED` error. The other error we might encounter is `WSAETIMEDOUT`, which occurs if the destination we're trying to reach is unavailable (either because of a communication-hardware failure on the route to the host or because the host is not currently on the network).



A Connected Socket

The following program fragment demonstrates a client connecting to a server listening on port 80.

```
#include <winsock2.h>

void main(void)
{
   WSADATA              wsadata;
   SOCKET               s;
   SOCKADDR_IN          server_addr;
   int                  port = 80, status;

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsadata);

   // Create a new socket to make a client connection.

      s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Set up a SOCKADDR_IN structure that will be used to connect
   // to a listening server on port 80. For demonstration
   // purposes, let's assume our server's IP address is 192.168.0.8.
   // Obviously, we will want to prompt the user for an IP address
   // and fill in this field with the user's data.

      server_addr.sin_family = AF_INET;
      server_addr.sin_port = htons(port);
      server_addr.sin_addr.s_addr = inet_addr("192.168.0.8");

   // Make a connection to the server with socket s.

      status = connect(s, (SOCKADDR *)&server_addr, sizeof(server_addr));
      if (status == 0)
      {
          // now we're connected, do sending and receiving here
      }

   // When we are finished sending and receiving data on socket s,
   // we should close the socket using the closesocket API.

      closesocket(s);

   // When our application is finished handling the connection, call
   // WSACleanup.

      WSACleanup();
}
```

We have seen how to set up a server and how to make a connection to a server from a client using C. But we haven't done anything other than handshaking so far. Let's now see how we can handle data transmission.

Data Transmission

Sending and receiving data is what network programming is all about. For sending data on a connected socket we use the `send` function. Likewise, we use the `recv` function for receiving data on a connected socket.

Sending

The API function to send data on a connected socket is `send`, which is prototyped as

```
int send(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags
);
```

The `SOCKET` parameter is the connected socket to send the data on. The second parameter, `buf`, is a pointer to the character buffer that contains the data to be sent. The third parameter, `len`, specifies the number of characters in the buffer to send. Finally, the `flags` parameter can be either `0`, `MSG_DONTROUTE`, or `MSG_OOB`. Alternatively, the flags parameter can be a bitwise `OR` any of those flags. The `MSG_DONTROUTE` flag tells the transport not to route the packets it sends. It is up to the underlying transport to honor this request (for example, if the transport protocol doesn't support this option, it will be ignored). The `MSG_OOB` flag signifies that the data should be sent out of band.

On a good return, `send` returns the number of bytes sent; otherwise, if an error occurs, `SOCKET_ERROR` will be returned. A common error is `WSAECO-NNABORTED`, which occurs when the virtual circuit terminates because of a timeout failure or a protocol error. When this occurs, the socket should be closed, as it is no longer usable.

Receiving

The `recv` function is the most basic way to accept incoming data on a connected socket. This function is defined as

```
int recv(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags
);
```

The first parameter, `s`, is the socket on which data will be received. The second parameter, `buf`, is the character buffer that will receive the data, and `len` is either the number of bytes we want to receive or the size of the buffer, `buf`. Finally, the `flags` parameter can be one of the following values: `0`, `MSG_PEEK`, or `MSG_OOB`. In addition, we can bitwise `OR` any one of these flags together. Of course, `0` specifies no special actions. `MSG_PEEK` causes the data that is available to be copied into the supplied receive buffer, but this data is not removed from the system's buffer. The number of bytes pending is also returned.

A streaming protocol is one that the sender and receiver may break up or coalesce data into smaller or larger groups. The main thing to be aware of with any function that sends or

receives data on a stream socket is that we are not guaranteed to read or write the amount of data we request. The following code snippets ensure that all our bytes are either sent or received.

```
// Make sure everything is sent
char sendbuff[2048];
int  nBytes = 2048, nLeft, idx;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
nLeft = nBytes;
idx = 0;

while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    nLeft -= ret;
    idx += ret;
}

// we are done sending.
```

Sending snippet

```
// Make sure everything is received
char    recvbuff[1024];
int     ret, nLeft, idx;

nLeft = 512; // say, we expect a 512byte of data…
idx = 0;

while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    idx += ret;
    nLeft -= ret;
}

// we are done receiving.
```

Receiving snippet

Breaking the Connection

Once we are finished with a socket connection, we must close it and release any resources associated with that socket handle. To actually release the resources associated with an open socket handle, we use the `closesocket` call. We should be aware, however, that `closesocket` can have some adverse effects—depending on how it is called—that can

lead to data loss. For this reason, a connection should be gracefully terminated with the `shutdown` function before a call to the `closesocket` function.

To ensure that all data an application sends is received by the peer, a well-written application should notify the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a graceful close and is performed by the `shutdown` function, defined as

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

The `how` parameter can be `SD_RECEIVE`, `SD_SEND`, or `SD_BOTH`. For `SD_RECEIVE`, subsequent calls to any receive function on the socket are disallowed. This has no effect on the lower protocol layers. And for TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. However, on UDP sockets incoming data is still accepted and queued (because `shutdown` has no meaning for connectionless protocols). For `SD_SEND`, subsequent calls to any send function are disallowed. For TCP sockets, this causes a `FIN` packet to be generated after all data is sent and acknowledged by the receiver. Finally, specifying `SD_BOTH` disables both sends and receives.

Note that not all connection-oriented protocols support graceful closure, which is what the `shutdown` API performs. For these protocols (such as ATM), only `closesocket` needs to be called to terminate the session.

Closing a Socket

The `closesocket` function closes a socket and is defined as

```
int closesocket (SOCKET s);
```

Calling `closesocket` releases the socket descriptor and any further calls using the socket fail with `WSAENOTSOCK`. If there are no other references to this socket, all resources associated with the descriptor are released. This includes discarding any queued data.

Connectionless Communication

Connectionless communication behaves differently than connection-oriented communication, so the method for sending and receiving data is substantially different. In IP, connectionless communication is accomplished through UDP/IP. UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving it from multiple sources. For example, if a client sends data to a server, the data is transmitted immediately regardless of whether the server is ready to receive it. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams, which are discrete message packets.

A UDP socket is created as follows:

```
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Receiving on Connectionless Communication

The steps in the process of receiving data on a connectionless socket are simple. First, we create the socket with either. Next, bind the socket to the interface on which we wish to receive data. This is done with the `bind` function (exactly like the session-oriented example). The difference with connectionless sockets is that we do not call `listen` or `accept`. Instead, we simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagrams originating from any machine on the network. The simplest of the receive functions is `recvfrom`, which is defined as

```
int recvfrom(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

Sending on Connectionless Communication

The simplest way to send data on a connectionless communication is to create a UDP socket and call `sendto` which is defined as

```
int sendto(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
    int tolen
);
```

Useful APIs

We have covered some of the general APIs needed to create a client/server either on a connection-oriented or connectionless communication. Let's now discuss some common useful APIs.

getpeername

This function is used to obtain the peer's socket address information on a connected socket. The function is defined as

```
int getpeername(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The first parameter is the socket for the connection; the last two parameters are a pointer to a SOCKADDR structure of the underlying protocol type and its length. For datagram sockets, this function returns the address passed to a connect call; however, it will not return the address passed to a `sendto` call.

getsockname

This function is the opposite of `getpeername`. It returns the address information for the local interface of a given socket. The function is defined as follows:

```
int getsockname(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The parameters are the same as the `getpeername` parameters except that the address information returned for socket `s` is the local address information. In the case of TCP, the address is the same as the server socket listening on a specific port and IP interface.

gethostbyname

This function translates a hostname to its network IPv4 address. It's prototype is

```
HOSTENT gethostbyname(const char FAR * hostname);
```

The parameter `hostname` is the name of the host we want the IP for.

Conclusion

In the above discussion, we have seen core Winsock APIs that are required for connection-oriented and connectionless communication specifically using the TCP and UDP protocols. For connection-oriented communication, we have seen how to accept a client connection and how to establish a client connection to a server. We have also seen the semantics for session-oriented data-send operations and data-receive operations. For connectionless communication, we have seen how to send and receive data. The Windows Socket API has numerous useful functions to work on network programing. I have tried to go through the most important APIs to perform simple network communication. The accompanying chat programs (CS461 Chat Client and Server programs) demonstrate the concepts discussed in this assignment. The chat program is a multithreaded program written entirely in C. Visual C 6.0 (part of the Visual Studio 6.0 suite, 1998) is needed to compile the code. The other platforms (.NET with C# or VB, VB 6) hide most of the "under the hood" concepts about

socket programming and for that reason I hate them. In addition to that the advantage with C is that we can directly talk to the APIs the operating system provide. When dealing with the other frameworks, they normally provide an interface to the API. Therefore our application needs to include the framework DLL in the installation package. For example, Visual Basic 6.0 provides an interface to the Windows Socket API (ws2_32.dll functions) through MSWINSCK.OCX ActiveX Control. That means, in order to make our application work, we need to include this OCX in the distribution. Note also that we can actually write a Winsock application with VB 6 that does not use the MSWINSCK.OCX ActiveX Control. But it needs too much effort and we will not escape from the MSVBVM6.0.DLL library, the Visual Basic 6 Runtime Library, which is not guaranteed to exist on Windows versions earlier than 2000 and it is around 2MB in size, which might be a headache for distribution through the Internet. But in C, we have the luxury of writing a neat and light weighted application that does not need any additional library other than the C Runtime Library which is guaranteed to exist on every Windows system because the Windows operating system itself relies on it.