CHAPTER 3: WINSOCK PROGRAMMING

Compiled by:

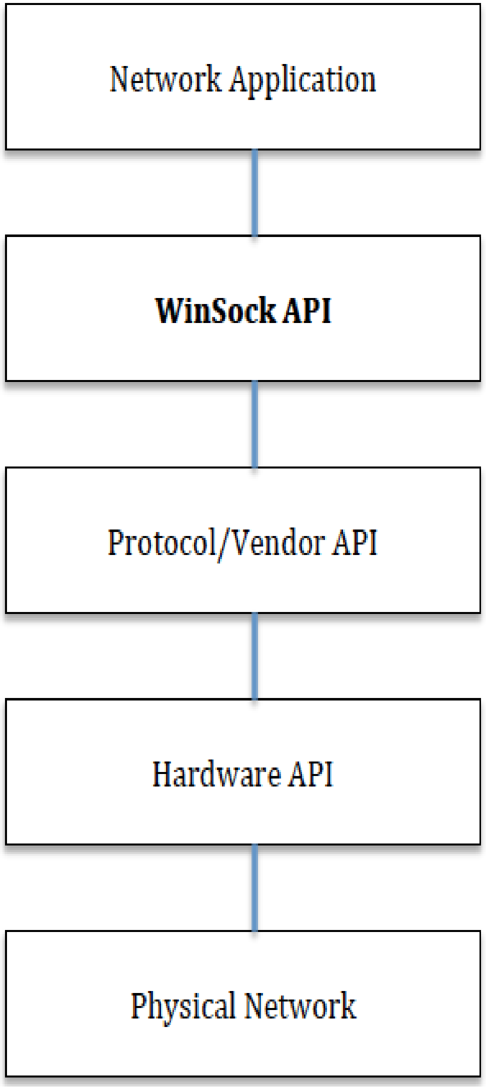
Madan Kadariya

NCIT

INTRODUCTION TO WINSOCK PROGRAMMING

- The Windows Sockets Application Programming Interface (WinSock API) is a library of functions that implements the socket interface.
- Winsock augments the Berkeley socket implementation by adding Windows-specific extension to support the message driven nature of the Windows operating system.
- WinSock is a network application programming interface (API) for Microsoft Windows; a well-defined set of data structures and function calls implemented as a dynamic link library (DLL).
- An application make function calls requesting generic network services (like send() and receive()); Winsock translates these generic requests into protocol- specific requests and performs the necessary tasks.
- Residing between the application and the network implementation, Winsock shields you from the details of low-level network protocols.

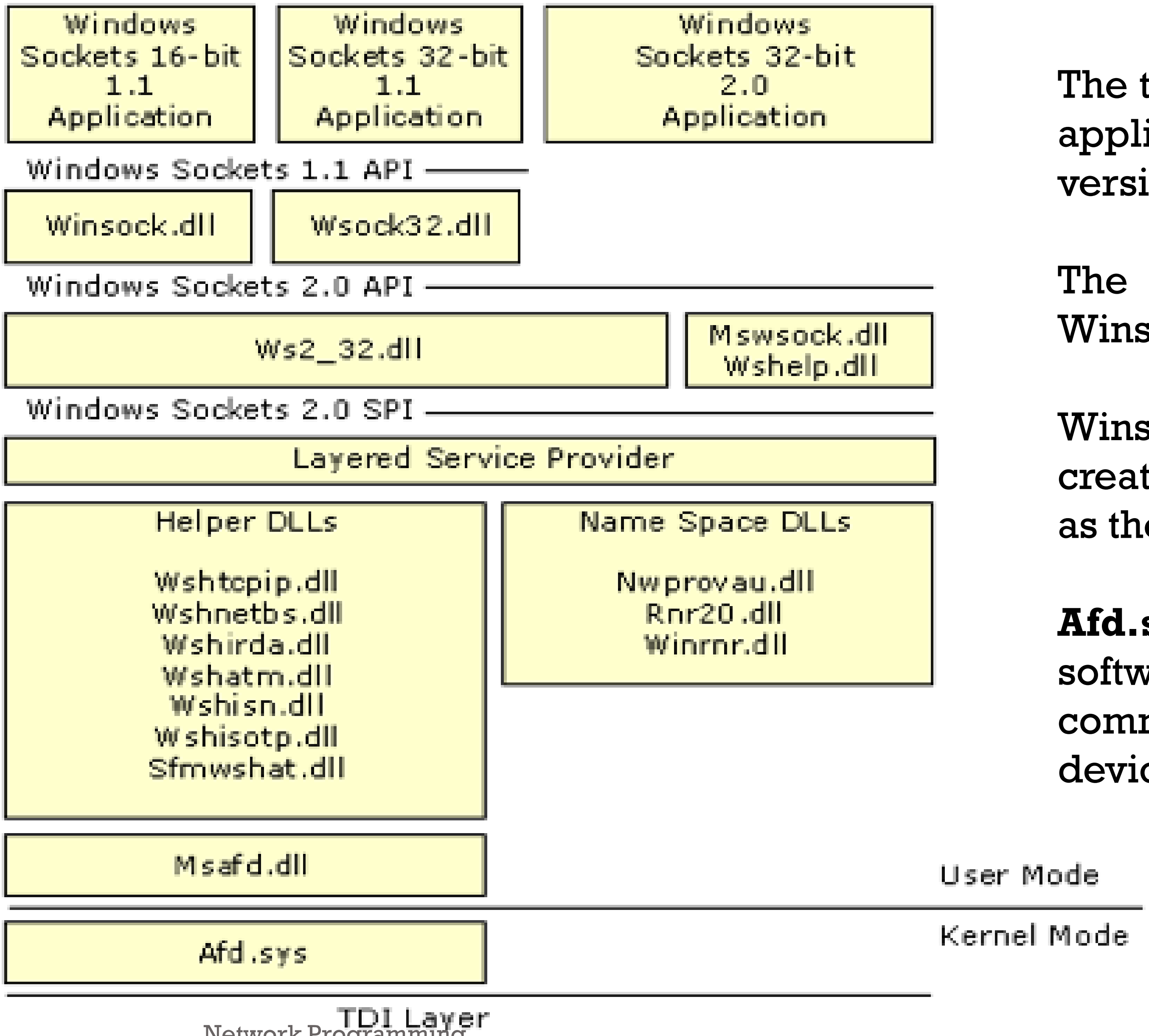
INTRODUCTION TO WINSOCK PROGRAMMING



The WinSock specification allows TCP/IP stack vendors to provide a consistent interface of their stacks so that application developers can write an application to the WinSock specification and have that application run on any vendor's WinSock-compatible TCP/IP protocol stack.

This is contrast to the days before the WinSock standard when software developers had to link their applications with libraries to each TCP/IP vendor's implementation. This limited the number of stacks that most applications ran on because of the difficulty in maintaining an application that used several different implementations of Berkeley sockets.

WINSOCK ARCHITECTURE



The top layer shows different network applications pertaining to different Winsock API versions and Windows architectures.

The two layers below it specify DLLs used for Winsock API version 1.1 and 2.0, respectively.

Winsock provides a Service Provider Interface for creating Winsock services, commonly referred to as the Winsock SPI.

Afd.sys is a Windows driver. A driver is a small software program that allows your computer to communicate with hardware or connected devices.

WINSOCK DLL (CURRENT IS WS2_32.DLL)

- **WINSOCK.DLL** is a dynamic-link library that provides a common application programming interface (API) for developers of network applications that use the Transmission Control Protocol/Internet Protocol (TCP/IP) stack.
- This means that a programmer who develops a Windows-based TCP/IP application, such as an FTP or Telenet client, can write one program that works with any TCP/IP protocol stack that provides Windows Socket Services (**WINSOCK.DLL**).

Some Winsock DLLs and their descriptions.

- **Winsock.dll**
  - 16-bit Winsock 1.1
- **Wsock32.dll**
  - 32-bit Winsock 1.1
- **Ws2_32.dll (this is latest)**
  - Main Winsock 2.0
- **Mswsock.dll**
  - Microsoft extensions to Winsock. Mswsock.dll is an API that supplies services that are not part of Winsock.
- **Ws2help.dll**
  - Platform-specific utilities. Ws2help.dll supplies operating system–specific code that is not part of Winsock.
- **Wshtcpip.dll**
  - Helper for TCP
- Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

TYPES OF DYNAMIC LINKING

There are two methods for calling a function in a DLL:

- In **load-time dynamic linking**, a module makes explicit calls to exported DLL functions as if they were local functions. This requires to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.
- In **run-time dynamic linking**, a module uses the **LoadLibrary** or **LoadLibraryEx** function to load the DLL at run time. After the DLL is loaded, the module calls the **GetProcAddress** function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by **GetProcAddress**. This eliminates the need for an import library.

DLLS AND MEMORY MANAGEMENT

- Every process that loads the DLL maps it into its virtual address space.
- After the process loads the DLL into its virtual address, it can call the exported DLL functions.
- The system maintains a per-process reference count for each DLL.
- When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero (run-time dynamic linking only), the DLL is unloaded from the virtual address space of the process.
- Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:
  - The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
  - The DLL uses the stack of the calling thread and the virtual address space of the calling process.
  - The DLL allocates memory from the virtual address space of the calling process.

ADVANTAGES OF DYNAMIC LINKING

- Multiple processes that load the same DLL at the same base address share a single copy of the DLL in physical memory. Doing this saves system memory and reduces swapping.
- When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
- Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers.
- A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the missing DLL are not available to the program.

Berkeley Socket Versus WinSock

- WinSock supports the TCP/IP domain for inter-process communication on the same computer as well as network communication. In addition to the TCP/IP domain, sockets in most UNIX implementations support the UNIX domain for the inter-process communication on the same computer.

- The return values of certain Berkeley functions are different. For example, the **socket()** function returns **-1** on failure in the UNIX environment; the WinSock implementation returns **INVALID_SOCKET**.

- Certain Berkeley functions have different names in WinSock. For example, in UNIX the **close()** system call is used to close the socket connection. In WinSock, the function is called **closesocket().** It is so because WinSock socket handles may not be UNIX-style file descriptors.

WinSock Extension to Berkeley Sockets

- WinSock has several extensions to Berkeley sockets. Most of these extensions are due to the message-driven architecture of Microsoft Windows. Some extensions are also required to support the non-preemptive nature of the 16-bit Windows operating environment.

Blocking I/O

- The simplest form of I/O in Windows Sockets 2 is blocking I/O. Sockets are created in blocking mode by default.
- Any I/O operation with a blocking socket will not return until the operation has been fully completed.
- Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer. Although this is simple, it is not necessarily the most efficient way to do I/O.

WinSock Asynchronous Functions

- WinSock was originally designed for the non-preemptive Windows architecture. For this reason, several extensions were added to traditional Berkeley sockets.

Blocking Versus Non-blocking

- Many of the Berkeley socket functions take an indeterminate amount of time to execute. When a function exhibits this behavior, it is said to block; calling the function blocks the further execution of the calling program.
- In the Berkeley UNIX environment, for which sockets were originally developed, this didn't pose a serious problem because the UNIX operating system would simply preempt the blocking program and begin running another program.
- Windows (unlike Windows NT) can't preempt a task, so all other programs are put on hold until the blocking call returns.
- The designers of WinSock knew this posed a serious problem, so they added special code in the blocking functions to force the message loop of other applications to be checked. But this was still not the most efficient technique.
- Berkeley sockets already have the notion of blocking versus non-blocking for some operations. For example, the send() function used to send data to a remote host may not return immediately, so the programmer is given the option of creating the socket with blocking or non-blocking sends.
- If the socket is created in blocking mode, it won't return until the data has been delivered. If it's created in non-blocking mode, the call to the send() function returns immediately and the program must call another function called select() to determine the status of the send.
- Windows and Windows NT can use the select() method of non-blocking calls, but the best thing to do in a Windows program is to use the special Windows asynchronous functions.

- The special Windows asynchronous functions begin with the prefix **WSAAsync**. These functions were added to WinSock to make Berkeley sockets better fit the message-driven paradigm of Windows.
- The most common events to use the asynchronous functions for are the sending and receiving of data. Sending data might not happen instantly, and receiving data most certainly will cause a program to wait unless it is receiving a constant stream of bytes.
- By creating a socket for nonblocking sends and receives and using the **WSAAsyncSelect()** function call, an application will receive event notification messages to inform it when it can send data or when data has arrived and needs to be read. In the mean time, when there is no data communications occurring, the rest of the program remains fully responsive to the user's actions.
- WinSock even extends Berkeley's nonblocking support to functions that could still cause a Berkeley UNIX program to block.
- The name server's job is to take as input the plain text representation of a computer's name and return that computer's IP address.
- GetXbyY is used to refer to database functions; function names take the form of get X by Y, or put another way: "Given Y, what is the corresponding X?"
- gethostbyname(): given the computer's name, what is its host information? In Berkeley UNIX, the getXbyY functions may block.
- WinSock adds asynchronous versions of the getXbyY functions called **WSAAsyncGetXbyY**. The gethostbyname() function is complimented by the nonblocking WinSock function called **WSAAsyncGetHostByName**(), for example. A call to a WSAAsyncGetXbyY function returns immediately with an identifying handle. When the actual work performed by the function has completed, a message is sent to the application notifying completion of the function with the specified handle.

- The WinSock asynchronous functions were added primarily for the benefit of the nonpreemptive Windows environment.
- The **WSAAsync** functions have an important use even in Windows NT. They allow an applications to remain responsive to the user.
- Users won't enjoy working with any program if it forces them to wait for completion of a long event.
- Most users expect a way to cancel operations that take a long time. For example, suppose that you have a program that takes as input a computer's plain-text name. The user enters the name and then presses a button labeled Look Up, which causes the gethostbyname() function to be called. Using gethostbyname() will cause the program to hang for an indeterminate amount of time until the request is carried out.
- Under Windows NT other programs would still run, but under Windows the performance of all programs would be degraded. This program could be modified to use **WSAAsyncGetHostByName**() instead of **gethostbyname**(). As soon as users press the Look Up button, the **WSAAsyncGetHostByName**() function is called and returns an identifying handle.
- If users wish to cancel the search, they can press the Cancel button, which terminates the request with that identifying handle. Users would maintain full control instead of being at the mercy of the program.

WINDOWS SOCKET EXTENSION; SETUP AND CLEANUP FUNCTION

- The WinSock functions the application needs are located in the dynamic library named **WINSOCK.DLL** or **WSOCK32.DLL** depending on whether the 16-bit or 32- bit version of Windows is being targeted.
- The application is linked with either **WINSOCK.LIB** or **WSOCK32.LIB** as appropriate.
- The include file where the WinSock functions and structures are defined is named **WINSOCK.H** for both the 16-bit and 32-bit environments.
- Before the application uses any WinSock functions, the application must call an initialization routine called **WSAStartup()**.
- Before the application terminates, it should call the **WSACleanup()** function.

WSASTARTUP

- The **WSAStartup**() function initializes the underlying Windows Sockets Dynamic Link Library (WinSock DLL).
- The **WSAStartup**() function gives the TCP/IP stack vendor a chance to do any application-specific initialization that may be necessary.
- **WSAStartup()** is also used to confirm that the version of the WinSock DLL is compatible with the requirements of the application.

The prototype of the WSAStartup() functions follows:

- ***int WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);***

- The **wVersionRequired** parameter is the highest version of the WinSock API the calling application can use. The high-order byte specifies the minor version and the low-order byte specifies the major version number. The **lpWSAData** parameter is a pointer to a WSAData structure that receives details of the WinSock implementation.

- **WinSock implementation.**

```
typedef struct WSAData {
        WORD wVersion; // version supplied
        WORD wHighVersion; // highest version that can be supplied
        char szDescription[WSADESCRIPTION_LEN+1]; /* a null terminated
                ASCII string that describes the WinSock implementation*/
        char szSystemStatus[WSASYS_STATUS_LEN+1]; /* a null terminated
                ASCII string that contains relevant status or configuration information */
        unsigned short iMaxSockets; /* the maximum number of sockets that a
                single process can potentially open */
        unsigned short iMaxUdpDg; /* the size, in bytes, of the largest UDP
                datagram that can be sent or received */
        char FAR * lpVendorInfo; // a pointer to a vendor specific data structures
} WSADATA;
```

WSACLEANUP

- The **WSACleanup**() function is used to terminate an application's use of WinSock.
- For every call to **WSAStartup**() there has to be a matching call to **WSACleanup**().
- **WSACleanup**() is usually called after the application's message loop has terminated.
- In an MFC application, the **ExitInstance**() member function of the **CWinApp** class provides a convenient location to call **WSACleanup**().
- The prototype follows:

int PASCAL FAR WSACleanup(void);

WSAGETLASTERROR

- The **WSAGetLastError**() function doesn't deal exclusively with startup or shutdown procedures, but it needs to be addressed early. Its function prototype looks like

int WSAGetLastError(void);

- **WSAGetLastError**() returns the last WinSock error that occurred. Because WinSock isn't really part of the operating system but is instead a later add-on, errno (like in UNIX and MS-DOS) couldn't be used.
- As soon as a WinSock API call fails, the application should call **WSAGetLastError**() to retrieve specific details of the error.

FUNCTION FOR HANDLING BLOCKED I/O

- The Berkeley method of using non-blocking sockets involves two functions: ioctl() and select().
- ioctl() is the UNIX function to perform input/output control on a file descriptor or socket.
- Because a WinSock socket descriptor may not be a true operating system file descriptor, ioctl() can't be used, so ioctlsocket() is provided instead. select() is used to determine the status of one or more sockets. The use of ioctlsocket() to convert a socket to nonblocking mode looks like this:

// put socket s into nonblocking mode

```
u_long ulCmdArg = 1; // 1 for non-blocking, 0 for blocking

ioctlsocket(s, FIONBIO, &ulCmdArg);
```

- Once a socket is in its non-blocking mode, calling a normally blocking function simply returns **WSAEWOULDBLOCK** if the function can't immediately complete.

```
SOCKET clientS;
clientS = accept(s, NULL, NULL);
if (clientS == INVALID_SOCKET)
{
int nError = WSAGetLastError();
// if there is no client waiting to connect to this server,
nError will be WSAEWOULDBLOCK
}
```

- Your server application could simply call accept() periodically until the call succeeded, or you could use the select() call to query the status of the socket.
- The select() function checks the readability, writeability, and exception status of one or more sockets.
- WinSock provides a function called **WSAAsyncSelect**() to solve the problem of blocking socket function calls. It is a much more natural solution to the problem than using **ioctlsocket**() and select().
- It works by sending a Windows message to notify a window of a socket event. Its prototype is as follows:

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, u_int wMsg, long lEvent);
```

- **s** is the socket descriptor for which event notification is required. **hWnd** is the Window handle that should receive a message when an event occurs on the socket. **wMsg** is the message to be received by **hWnd** when a socket event occurs on socket **s**. It is usually a user-defined message (WM_USER + n). **lEvent** is a bitmask that specifies the events in which the application is interested.
- **WSAAsyncSelect**() returns 0 (zero) on success and **SOCKET_ERROR** on failure. On failure, **WSAGetLastError**() should be called. **WSAAsyncSelect**() is capable of monitoring several socket events.

Event	**Meaning**
FD_READ	Socket ready for reading
FD_WRITE	Socket ready for writing
FD_OOB	Out-of-band data ready for reading on socket
FD_ACCEPT	Socket ready for accepting a new incoming connection
FD_CONNECT	Connection on socket completed
FD_CLOSE	Connection on socket has been closed

-

- The lEvent parameter is constructed by doing a logical OR on the events in which you're interested. To cancel all event notifications, call WSAAsyncSelect() with wMsg and lEvent set to 0.

ASYNCHRONOUS DATABASE FUNCTION

- WinSock provides a set of procedures commonly referred to as the database functions. The duty of these database functions is to convert the host and service names that are used by humans into a format usable by the computer.
- The computers on an internetwork also require that certain data transmitted between them be in a common format. WinSock provides several conversion routines to fulfill this requirement.

Conversion Routines and Network Byte Ordering

- There are four primary byte-order conversion routines. They handle the conversions to and from unsigned short integers and unsigned long integers.

Unsigned Short Integer Conversion

- The **htons()** and **ntohs()** functions convert an unsigned short from host-to- network order and from network-to-host order. The prototype looks like.

```
u_short htons(u_short hostshort);

u_short ntohs(u_short netshort);
```

Unsigned Long Integer Conversion

- The **htonl()** and **ntohl()** functions work like **htons()** and **ntohs()** except that they operate on four-byte unsigned longs rather than unsigned shorts. The prototype look like the following:

```
u_long PASCAL htons(u_long hostlong);

u_long PASCAL ntohs(u_long netlong);
```

Converting IP Addresses

- WinSock provides another set of conversion functions that provide a translation between the ASCII representation of a dotted-decimal IP address and the internal 32-bit, byte-ordered number required by other WinSock functions.

Converting an IP Address String to Binary

- **inet_addr()** converts a dotted-decimal IP address string into a number suitable for use as an Internet address. Its function prototype is as follows:

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

- cp is a pointer to a string representing an IP address in dotted-decimal notation. The **inet_addr()** function returns a binary representation of the Internet address given. This value is already in network byte order, so there is no need to call **htonl()**. If the cp string doesn't contain a valid IP address, inet_addr() returns **INADDR_NONE**. E.g.

```
u_long ulIPAddress = inet_addr("166.78.16.148");
```

Converting a Binary IP Address to a String

- **inet_ntoa()** performs the opposite job of **inet_addr()**. Its function prototype is as follows:

```
char *  inet_ntoa(struct in_addr in);
```

- **in** is a structure that contains an Internet host addess. On success, the **inet_ntoa()** function returns a pointer to a string with a dotted-decimal representation of the IP address. On error, NULL is returned. A NULL value means that the IP address passed as the in parameter is invalid.E.g.

- // first get an unsigned long with a valid IP address

```
u_long ulIPAddress = inet_addr("166.78.16.148");
```

- // copy the four bytes of the IP address into an in_addr structure IN_ADDR in;

```
memcpy(&in, &ulIPAddress, 4);
```

- // convert the IP address back into a string

```
char lpszIPAddress[16];
lstrcpy(lpszIPAddress, inet_ntoa(in));
```

What’s My Name?

- Some applications need to know the name of the computer on which they are running. The **gethostname()** function provides this functionality. The function’s prototype looks like the following.

```
int gethostname(char  * name, int namelen);
```

- name is a pointer to a character array that will accept the null-terminated host name, and namelen is the size of that character array. The **gethostname()** function returns 0 (zero) on success and **SOCKET_ERROR** on failure. On a return value of **SOCKET_ERROR**, the application can call **WSAGetLastError()** to determine the specifics of the problem.

```
#define HOST_NAME_LEN 50
char lpszHostName[HOST_NAME_LEN]; // will accept the host name
char lpszMessage[100]; // informational message
if (gethostname(lpszHostName, HOST_NAME_LEN) == 0) {
printf(lpszMessage, “This computer’s name is %s”, lpszHostName);
} else {
      printf(lpszMessage, “gethostname() generated error %d”, WSAGetLastError());
}
```

Finding a Host’s IP Address

- The function of **gethostbyname()** is to take a host name and return its IP address. This function, and its asynchronous counterpart named **WSAAsyncGetHostByName()**, may perform a simple table lookup on a host file local to the computer on which the program us running, or it may send the request across the network to a name server.
- The function’s prototype looks like the following:

```
struct hostent * gethostbyname(const char * name);
```

- name is a pointer to a null-terminated character array that contains the name of the computer about which you want host information. The **hostent** structure returned has the following format:

```
struct hostent {
  char  * h_name; // official name of host
  char ** h_aliases; // alias list
  short h_addrtype; // host address type
  short h_length; // length of address
  char  * * h_addr_list; // list of addresses
  #define h_addr h_addr_list[0] // address, for backward compatibility
};
```

Asynchronously Finding a Host’s IP Address

- In the getXbyY functions, one of which is gethostbyname(), the data retrieved might come from the local host or might come by way of a request over the network to a server of some kind.
- Consequently, the application has to be concerned with response times and the responsiveness of the application to the user while those network requests are taking place.
- The **WSAAsyncGetHostByName**() function is the asynchronous version of gethostbyname().
- The function prototype for WSAAsyncGetHostByName() is as follows:

```
HANDLE WSAAsyncGetHostByName(HWND hwnd, u_int wMsg, const char *name, char
*buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetHostByName**() has completed its asynchronous operation.
- **wMsg** is the message that will be posted to **hWnd** when the asynchronous operation is complete.
- **name** is a pointer to a string that contains the host name for which information is being requested.
- **buf** is a pointer to an area of memory that, on successful completion of the host name lookup, will contain the **hostent** structure for the desired host.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety’s sake.
- If the asynchronous operation is initiated successfully, the return value of **WSAAsyncGetHostByName**() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero), and **WSAGetLastError**() should be called to find out the reason for the error.

Canceling an Outstanding Asynchronous Request

- The handle returned by the asynchronous database functions, such as **WSAAsyncGetHostByName**(), can be used to terminate the database lookup.
- The **WSACancelAsyncRequest**() function performs this task. Its prototype is the following:

```
int WSACancelAsyncRequest(HANDLE hAsyncTaskHandle);
```

- **hAsyncTaskHandle** is the handle to the asynchronous task you wish to abort.
- On success, this function returns 0 (zero).
- On failure, it returns SOCKET_ERROR, and **WSAGetLastError**() can be called.

Finding a Host Name When You Know Its IP Address

- The function of **gethostbyaddr**() is to take the IP address of a host and return its name. This function, and its asynchronous counterpart named **WSAAsyncGetHostByAddr**(), might perform a simple table lookup on a host file local to the computer on which the program is running, or it might send the request across the network to a name server. The function's prototype looks like the following:

```
struct hostent * PASCAL gethostbyaddr(const char * addr, int len, int type);
```

- **addr** is a pointer to the IP address, in network byte order, of the computer about which you want host information.
- **len** is the length of the address to which addr points. In WinSock 1.1, the length is always four because this version of the specification supports only Internet style addressing.
- **type** must always be **PF_INET** for the same reason.
- The **gethostbyaddr**() function returns a pointer to a **hostent** host entry structure on success and NULL on failure.
- Upon a return value of NULL, you can call **WSAGetLastError**() to determine the specifics of the problem.

Asynchronously Finding a Host Name When You Know Its IP Address

- The **WSAAsyncGetHostByAddr**() function is the asynchronous version of gethostbyaddr(). Its function prototype is as follows:

```
HANDLE WSAAsyncGetHostByAddr(HWND hWnd, u_int wMsg, const char * addr,
int len, int type, char * buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetHostByAddr**() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to **hWnd** when the asynchronous operation is complete.
- **addr** is a pointer to the IP address, in network byte order, of the computer about which you want host information.
- **len** is the length of the address to which **addr** points and is always 4 (four) for Internet addresses. type must always be PF_INET because WinSock 1.1 supports only Internet-style addressing.
- **buf** is a pointer to an area of memory that, upon successful completion of the address lookup, will contain the hostent structure for the desired host.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value of **WSAAsyncGetHostByAddr**() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero) and WSAGetLastError() should be called to find out the reason for the error.

Finding a Service's Port Number

- The **getservbyname**() function gets service information corresponding to a specific service name and protocol. Its function prototype looks like the following:

```
struct servent *  getservbyname(const char * name, const char * proto);
```

- name is a pointer to a string that contains the service for which you are searching. proto is a pointer to a string that contains the transport protocol to use; it's either "udp", "tcp", or NULL. A NULL proto will match on the first service in the services table that has the specified name, regardless of the protocol. The **servent** structure returned has the following format:

```
struct servent {
char * s_name; // official service name
char ** s_aliases; // alias list
short s_port; // port #
char * s_proto; // protocol to use
};
```

- The **getservbyname**() function returns a pointer to a servent structure on success and NULL on failure. On a return value of NULL, you can call WSAGetLastError() to determine the specifics of the problem.

Asynchronously Finding a Service’s Port Number

- **WSAAsyncGetServByName**() is the asynchronous counterpart to getservbyname(). Its function prototype is as follows:
- *HANDLE WSAAsyncGetServByName(HWND hWnd, u_int wMsg, const char * name, const char * proto, char * buf, int buflen);*
- **hWnd** is the handle to the window to which a message will be sent when WSAAsyncGetServByName() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to hWnd when the asynchronous operation is complete.
- **name** is a pointer to a service name about which you want service information. proto is a pointer to a protocol name; it is “tcp”, “udp”, or NULL.
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory that, on successful completion of the service lookup, will contain the servent structure for the desired service.
- **buflen** is the size of the buf buffer.
- It should be MAXGETHOSTSTRUCT for safety’s sake.

Finding a Service Name When You Know Its Port Number

- The **getservbyport**() function gets service information corresponding to a specific port and protocol. Its function prototype looks like the following:

```
struct servent FAR * PASCAL FAR getservbyport(int port, const char FAR *
proto);
```

- **port** is the service port, in network byte order.
- **proto** is a pointer to a protocol name; it is “tcp”, “udp”, or NULL.
- If **proto** is NULL, the first matching service is returned.
- The **getservbyport**() function returns a pointer to a servent structure on success and NULL on failure.
- On a return value of NULL, you can call WSAGetLastError() to determine the specifics of the problem.

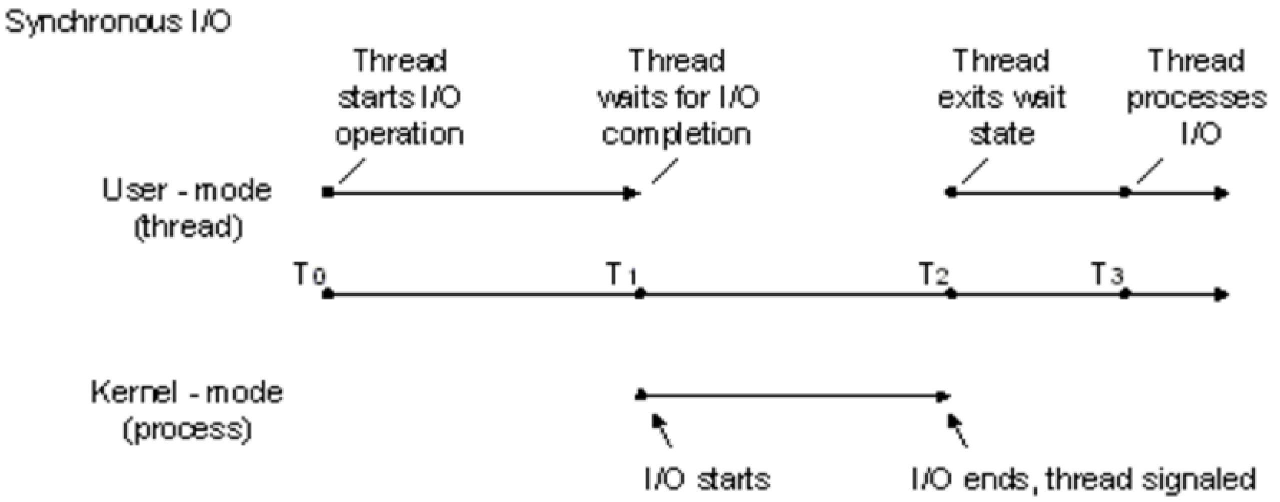
Asynchronously Finding a Service Name When You Know Its Port Number

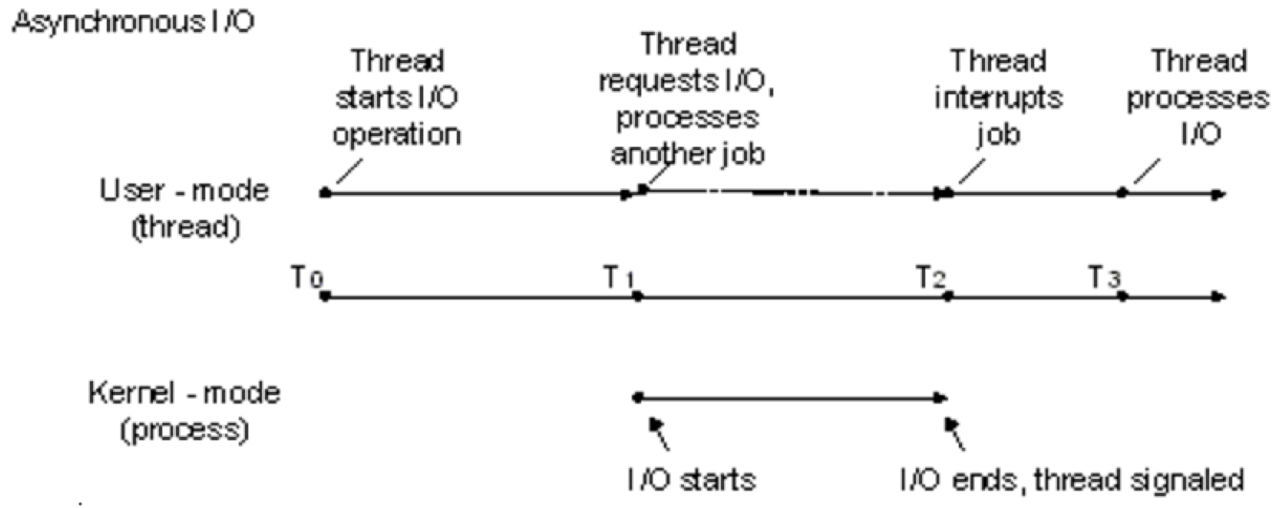
- WSAAsyncGetServByPort() is the asynchronous counterpart to getservbyport(). Its function prototype is as follows:

```
HANDLE PASCAL FAR WSAAsyncGetServByPort(HWND hWnd, u_int wMsg, int port,
const char FAR * proto, char FAR * buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent when WSAAsyncGetServByName() has completed its asynchronous operation.
- **wMsg** is the user defined message that will be posted to hWnd when the asynchronous operation is complete.
- **port** is the service port, in network byte order, of the service about which you want information. proto is a pointer to a protocol name; it is "tcp", "udp", or NULL.
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory that, on successful completion of the service lookup, will contain the servent structure for the desired service.
- **buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value of WSAAsyncGetServByName() is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero) and WSAGetLastError() should be called to find out the reason for the error.

SYNCHRONOUS AND ASYNCHRONOUS I/O





ASYNCHRONOUS I/O FUNCTIONS

- There are two types of input/output (I/O) synchronization: **synchronous I/O** and **asynchronous I/O**.
- Asynchronous I/O is also referred to as overlapped I/O.
- In synchronous file I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.
- A thread performing asynchronous file I/O sends an I/O request to the kernel by calling an appropriate function.
- If the request is accepted by the kernel, the calling thread continues processing another job until the kernel signals to the thread that the I/O operation is complete.
- It then interrupts its current job and processes the data from the I/O operation as necessary.
- In situations where an I/O request is expected to take a large amount of time, such as a refresh or backup of a large database or a slow communications link, asynchronous I/O is generally a good way to optimize processing efficiency.
- However, for relatively fast I/O operations, the overhead of processing kernel I/O requests and kernel signals may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made. In this case, synchronous I/O would be better.
- WinSock provides functions (**WSASend**, **WSASendTo**, **WSARecv**, **WSARecvFrom**, **WSAIoctl**, etc.) that support both synchronous and asynchronous I/O.
- **WSASocket** is used to create overlapped socket.

```
SOCKET WSASocket(
_In_ int                 af,
_In_ int                 type,
_In_ int                 protocol,
_In_ LPWSAPROTOCOL_INFO lpProtocolInfo,
_In_ GROUP               g,
_In_ DWORD               dwFlags);   // Used to create overlapped socket
```

- **af[in]** : address family; AF_INET, AF_INET6, AF_UNSPEC
- **type[in]** : type of the new socket; SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- **protocol[in]** : the protocol to be used. If a value of 0 is specified, the caller does not wish to specify a protocol and the service provider will choose the protocol to use; else IPPROTO_TCP, IPPROTO_UDP, IPROTO_ICMP
- **lpProtocolInfo[in]** : A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created.
- **g [in]** : An existing socket group ID.
- **dwFlags[in]** : A set of flags used to specify additional socket attributes.

WSASEND

```
int WsaSend(
_In_ SOCKET                                  s,
_In_ LPWSABUF                                lpBuffers,
_In_  DWORD                                  dwBufferCount,
_Out_ LPDWORD                                lpNumberOfBytesSent,
_In_  DWORD                                  dwFlags,
_In_  LPWSAOVERLAPPED                        lpOverlapped,
_In_  LPWSAOVERLAPPED_COMPLETION_ROUTINE   lpCompletionRoutine );
```

- **s[in]** : A descriptor that identifies a connected socket.
- **lpBuffers[in]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length, in bytes, of the buffer.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberOfBytesSent[out]** : The pointer to the number, in bytes, sent by this call if the I/O operation completes immediately. NULL for overlapped socket.
- **dwFlags[in]** : The flags used to modify the behavior of the WSASend function call.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.
- **lpCompletionRoutine[in]** : A pointer to the completion routine (function) called when the send operation has been completed. This parameter is ignored for non- overlapped sockets.

WSASENDTO

```
int WSASendTo(
_In_   SOCKET                                   s,
_In_   LPWSABUF                                 lpBuffers,
_In_   DWORD                                    dwBufferCount,
_Out_  LPDWORD                                  lpNumberOfBytesSent,
_In_   DWORD                                    dwFlags,
_In_   const struct sockaddr                   *lpTo,
_In_   int                                      iToLen,
_In_   LPWSAOVERLAPPED                          lpOverlapped,
_In_   LPWSAOVERLAPPED_COMPLETION_ROUTINE       lpCompletionRoutine );
```

- **s[in]** : A descriptor identifying a socket.
- **lpBuffers[in]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length of the buffer, in bytes. This array must remain valid for the duration of the send operation.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).
- **lpNumberofBytesSent[out]** : The pointer to the number, in bytes, sent by this call if the I/O operation completes immediately. NULL for overlapped socket.
- **dwFlags[in]** : The flags used to modify the behavior of the WSASendTo call.
- **lpTo[in]** : An optional pointer to the address of the target socket in the SOCKADDR structure.
- **iToLen** [**in**] : The size, in bytes, of the address in the IpTo parameter.

WSARECV

```
int WSARecv(
_In_    SOCKET                             s,
_Inout_ LPWSABUF                           lpBuffers,
_In_    DWORD                              dwBufferCount,
_Out_   LPDWORD                            lpNumberOfBytesRecvd,
_Inout_ LPDWORD                            lpFlags,
_In_    LPWSAOVERLAPPED                    lpOverlapped,
_In_    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

- **s[in]** : A descriptor identifying a connected socket.
- **lpBuffers[in, out]** : A pointer to an array of WSABUF strucutres. Each WSABUF structure contains a pointer to a buffer and the length, in bytes, of the buffer.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** : A pointer to the number, in bytes, of data received by this call if the receive operation competes immediately. (NULL for overlapped socket)
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of the WSARecv function call.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).
- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets).

WSARECVFROM

```
int WSARecvFrom(
  _In_    SOCKET                             s,
  _Inout_ LPWSABUF                           lpBuffers,
  _In_    DWORD                              dwBufferCount,
  _Out_   LPDWORD                            lpNumberOfBytesRecvd,
  _Inout_ LPDWORD                            lpFlags,
  _Out_   struct sockaddr                    *lpFrom,
  _Inout_ LPINT                              lpFromlen,
  _In_    LPWSAOVERLAPPED                    lpOverlapped,
  _In_    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- **s [in]** : A descriptor identifying a socket.
- **lpBuffers [in, out]** : A pointer to an array of WSABUF structures. Each WSABUF structure contains a pointer to a buffer and the length of the buffer.
- **dwBufferCount [in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** : A pointer to the number of bytes received by this call if the WSARecvFrom operation completes immediately. (NULL for overlapped socket)
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of the WSARecvFrom function call.
- **lpFrom [out]** : An optional pointer to a buffer that will hold the source address upon the completion of the overlapped operation.
- **lpFromlen [in, out]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the WSARecvFrom operation has been completed (ignored for nonoverlapped sockets).

WSAIOCTL

```
int WSAIoctl(
_In_  SOCKET                              s,
_In_  DWORD                               dwIoControlCode,
_In_  LPVOID                              lpvInBuffer,
_In_  DWORD                               cbInBuffer,
_Out_ LPVOID                              lpvOutBuffer,
_In_  DWORD                               cbOutBuffer,
_Out_ LPDWORD                             lpcbBytesReturned,
_In_  LPWSAOVERLAPPED                     lpOverlapped,
_In_  LPWSAOVERLAPPED_COMPLETION_ROUTINE  lpCompletionRoutine );
```

- **s [in]** : A descriptor identifying a socket.
- **dwIoControlCode [in**] : The control code of operation to perform
- **lpInBuffer [in]** : A pointer to the input buffer
- **cbInBuffer [in]** : The size, in bytes, of the input buffer
- **lpOutBuffer [out**] : A pointer to the output buffer
- **cbOutBuffer [in]** : The size, in bytes, of the output buffer
- **lpcbBytesReturned [out]** :A pointer to actual number of bytes of output
- **lpOverlapped [in]** : A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets)
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets)

ERROR HANDLING FUNCTIONS; ASYNCHRONOUS OPERATION

- If asynchronous function returns SOCKET_ERROR, and the specific error code retrieved by calling WSAGetLastError() is WSA_ISO_PENDING, then it means the operlapped operation has been successfully initiated and the completion will be indicated at a later time.
- Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.
- When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine(if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**.
- ***BOOL WSAAPI WSAGetOverlappedResult(_In_ SOCKET s,_In_ LPWSAOVERLAPPED lpOverlapped,_Out_ LPDWORD lpcbTransfer,_In_ BOOL fWait,_Out_ LPDWORD lpdwFlags);***
- **s [ın]**: A descriptor identifying the socket.
- **lpOverlapped [in]**: A pointer to a WSAOVERLAPPED structure that was specified when the overlapped operation was started. This parameter must not be a NULL pointer.
- **lpcbTransfer [out]**: A pointer to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by the WSAIoctl function.

- **fWait [in]**: A flag that specifies whether the function should wait for the pending overlapped operation to complete.
- **lpdwFlags [out]**: A pointer to a 32-bit variable that will receive one or more flags that supplement the completion status.
- If **WSAGetOverlappedResult** succeeds, the return value is **TRUE**. This means that the overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated.
- If **WSAGetOverlappedResult** returns **FALSE**, this means that either the overlapped operation has not completed, the overlapped operation completed but with errors, or the overlapped operation's completion status could not be determined due to errors in one or more parameters to **WSAGetOverlappedResult**.
- On failure, the value pointed to by *lpcbTransfer* will not be updated. Use WSAGetLastError to determine the cause of the failure (either by the WSAGetOverlappedResult function or by the associated overlapped operation).

USING NON-BLOCKING SOCKET, NON-BLOCKING WITH CONNECT

- When connect is called on a blocking socket, the function blocks i.e. the function doesn't return until the TCP's 3-way handshake is completed (actually, until SYN, ACK is received from remote end).
- The **ioctlsocket** function can be used to set socket in non-blocking mode. With a non-blocking socket, the connect returns immediately without completion.
- In this case, connect will return SOCKET_ERROR, and WSAGetLastError will return WSAEWOULDBLOCK.
- In this case, there are three possible scenarios.
  - Use the select function to determine the completion of the connection request by checking to see if the socket is writeable. If connect fails, failure of the connect attempt is indicated in exceptfds (application must then call getsockopt SO_ERROR to determine the error value to describe why the failure occurred).
  - If the application is using WSAAsyncSelect to indicate interest in connect events, then the application will receive an FD_CONNECT notification indicating that the connect operation is complete (successfully or not).
  - If the application is using WSAEventSelect to indicate interest in connection events, then the associated event object will be signaled indicating that the connect operation is complete (successfully or not).

SELECT IN CONJUNCTION WITH ACCEPT, SELECT WITH RECV/RECVFROM AND SEND/SENDTO

Select with accept

- The accept function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking.
- If the socket is marked as non-blocking and no pending connections are present on the queue, accept returns an error.
- When using non-blocking socket, select function can be used to determine if the connecting is present by checking to see if the socket is readable. Then, accept returns successfully, immediately.

Select with recv/recvfrom

- With blocking socket, recv/recvfrom blocks if no data is available. When using non-blocking socket, select function can be used to determine if the data is available by checking to see if the socket is readable. Then, recv/recvfrom returns successfully, immediately.

Select with send/sendto

- With blocking socket, send/sendto blocks if data can't be written to the kernel. When using non-blocking socket, select function can be used to determine if the data can be written to the kernel by checking to see if the socket is writable. Then, send/sendto returns successfully, immediately.

Getting Started with WinSock

General model for creating a streaming TCP/IP Server and Client.

Client

1. Initialize Winsock.
2. Create a socket.
3. Connect to the server.
4. Send and receive data.
5. Disconnect.

Server

1. Initialize Winsock.
2. Create a socket.
3. Bind the socket.
4. Listen on the socket for a client.
5. Accept a connection from a client.
6. Receive and send data.
7. Disconnect.

Initializing Winsock

- All processes (applications or DLLs) that call Winsock functions must initialize the use of the Windows Sockets DLL before making other Winsock functions calls. This also makes certain that Winsock is supported on the system.

To initialize Winsock

1. Create a WSADATA object called wsaData.

```
WSADATA wsaData;
```

2. Call WSAStartup and return its value as an integer and check for errors.

```
int iResult;
// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
printf("WSAStartup failed: %d\n", iResult);
return 1;
}
```

- The **WSAStartup** function is called to initiate use of WS2_32.dll. The WSADATA structure contains information about the Windows Sockets implementation. The MAKEWORD(2,2) parameter of WSAStartup makes a request for version 2.2 of Winsock on the system, and sets the passed version as the highest version of Windows Sockets support that the caller can use.

CREATING A SOCKET FOR THE CLIENT

- After initialization, a SOCKET object must be instantiated for use by the client.

To create a socket

1. Declare an **addrinfo** object that contains a **sockaddr** structure and initialize these values. For this application, the Internet address family is unspecified so that either an IPv6 or IPv4 address can be returned. The application requests the socket type to be a stream socket for the TCP protocol.

```
struct addrinfo *result = NULL, *ptr = NULL, hints;

ZeroMemory( &hints, sizeof(hints) );

hints.ai_family = AF_UNSPEC;

hints.ai_socktype = SOCK_STREAM;

hints.ai_protocol = IPPROTO_TCP;
```

2. Call the **getaddrinfo** function requesting the IP address for the server name passed on the command line. The TCP port on the server that the client will connect to is defined by DEFAULT_PORT as 27015 in this sample. The **getaddrinfo** function returns its value as an integer that is checked for errors.

```
#define DEFAULT_PORT "27015"
// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
printf("getaddrinfo failed: %d\n", iResult);
WSACleanup();
return 1;
}
```

3. Create a SOCKET object called ConnectSocket.

- SOCKET ConnectSocket = INVALID_SOCKET;

4. Call the socket function and return its value to the ConnectSocket variable. For this application, use the first IP address returned by the call to getaddrinfo that matched the address family, socket type, and protocol specified in the hints parameter. In this example, a TCP stream socket was specified with a socket type of SOCK_STREAM and a protocol of IPPROTO_TCP. The address family was left unspecified (AF_UNSPEC), so the returned IP address could be either an IPv6 or IPv4 address for the server. If the client application wants to connect using only IPv6 or IPv4, then the address family needs to be set to AF_INET6 for IPv6 or AF_INET for IPv4 in the hints parameter.

```
// Attempt to connect to the first address returned by

// the call to getaddrinfo

ptr=result;

// Create a SOCKET for connecting to server

ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
```

5. Check for errors to ensure that the socket is a valid socket.

```
if (ConnectSocket == INVALID_SOCKET) {
printf("Error at socket(): %ld\n",WSAGetLastError());
freeaddrinfo(result);
WSACleanup();
return 1;
}
```

- The parameters passed to the socket function can be changed for different implementations. Error detection is a key part of successful networking code. If the socket call fails, it returns INVALID_SOCKET. The if statement in the previous code is used to catch any errors that may have occurred while creating the socket. WSAGetLastError returns an error number associated with the last error that occurred.

Connecting to a Socket

- For a client to communicate on a network, it must connect to a server.
- **To connect to a socket**
- Call the connect function, passing the created socket and the sockaddr structure as parameters. Check for general errors.

```
// Connect to server.

iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);

if (iResult == SOCKET_ERROR) {

closesocket(ConnectSocket);

ConnectSocket = INVALID_SOCKET;

}

// Should really try the next address returned by getaddrinfo

// if the connect call failed

freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {

printf("Unable to connect to server!\n");

WSACleanup();

return 1;

}
```

- The getaddrinfo function is used to determine the values in the sockaddr structure.
- In this example, the first IP address returned by the getaddrinfo function is used to specify the sockaddr structure passed to the connect.
- If the connect call fails to the first IP address, then try the next addrinfo structure in the linked list returned from the getaddrinfo function.
- The information specified in the sockaddr structure includes:
  - the IP address of the server that the client will try to connect to.
  - the port number on the server that the client will connect to. This port was specified as port 27015 when the client called the getaddrinfo function.

SENDING AND RECEIVING DATA ON THE CLIENT

Client

```
#define DEFAULT_BUFLEN 512

int recvbuflen = DEFAULT_BUFLEN;

char *sendbuf = "this is a test";

char recvbuf[DEFAULT_BUFLEN];

int iResult;

// Send an initial buffer

iResult = send(ConnectSocket, sendbuf, (int)
strlen(sendbuf), 0);

if (iResult == SOCKET_ERROR) {

printf("send failed: %d\n", WSAGetLastError());

closesocket(ConnectSocket);

WSACleanup();

return 1;

}

printf("Bytes Sent: %ld\n", iResult);
```

```
// shutdown the connection for sending since no more data will be sent
// the client can still use the ConnectSocket for receiving data
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n", WSAGetLastError());
closesocket(ConnectSocket);
WSACleanup();
return 1;
}
// Receive data until the server closes the connection
do {
iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
if (iResult > 0)
printf("Bytes received: %d\n", iResult);
else if (iResult == 0)
printf("Connection closed\n");
else
printf("recv failed: %d\n", WSAGetLastError());
} while (iResult > 0);
```

Disconnecting the Client

- Once the client is completed sending and receiving data, the client disconnects from the server and shutdowns the socket.

To disconnect and shutdown a socket

1. When the client is done sending data to the server, the shutdown function can be called specifying SD_SEND to shutdown the sending side of the socket. This allows the server to release some of the resources for this socket. The client application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n", WSAGetLastError());
closesocket(ConnectSocket);
WSACleanup();
return 1;
}
```

2. When the client application is done receiving data, the closesocket function is called to close the socket. When the client application is completed using the Windows Sockets DLL, the WSACleanup function is called to release resources.

```
// cleanup

closesocket(ConnectSocket);

WSACleanup();

return 0;
```

CREATING A SOCKET FOR THE SERVER

- After initialization, a SOCKET object must be instantiated for use by the server.

To create a socket for the server

1. The getaddrinfo function is used to determine the values in the sockaddr structure:
   - AF_INET is used to specify the IPv4 address family.
   - SOCK_STREAM is used to specify a stream socket.
   - IPPROTO_TCP is used to specify the TCP protocol.
   - AI_PASSIVE flag indicates the caller intends to use the returned socket address structure in a call to the bind function.
   - When the AI_PASSIVE flag is set and nodename parameter to the getaddrinfo function is a NULL pointer, the IP address portion of the socket address structure is set to INADDR_ANY for IPv4 addresses or IN6ADDR_ANY_INIT for IPv6 addresses.
   - 27015 is the port number associated with the server that the client will connect to.

- The addrinfo structure is used by the getaddrinfo function.

```
#define DEFAULT_PORT "27015"
struct addrinfo *result = NULL, *ptr = NULL, hints;
```

Compiled by Chandra Mohan Jayaswal

```
ZeroMemory(&hints, sizeof (hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
// Resolve the local address and port to be used by the server
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
printf("getaddrinfo failed: %d\n", iResult);
WSACleanup();
return 1;
}
```

2. Create a SOCKET object called ListenSocket for the server to listen for client connections.

SOCKET ListenSocket = INVALID_SOCKET;

3. Call the socket function and return its value to the ListenSocket variable. For this server application, use the first IP address returned by the call to getaddrinfo that matched the address family, socket type, and protocol specified in the hints parameter.
   - If the server application wants to listen on IPv6, then the address family needs to be set to AF_INET6 in the hints parameter.
   - If a server wants to listen on both IPv6 and IPv4, two listen sockets must be created, one for IPv6 and one for IPv4.
   - These two sockets must be handled separately by the application. Windows Vista and later offer the ability to create a single IPv6 socket that is put in dual stack mode to listen on both IPv6 and IPv4. For more information on this feature.

// Create a SOCKET for the server to listen for client connections

ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);

4. Check for errors to ensure that the socket is a valid socket.

```
if (ListenSocket == INVALID_SOCKET) {
printf("Error at socket(): %ld\n", WSAGetLastError());
freeaddrinfo(result);
WSACleanup();
return 1;
}
```

Binding a Socket

- For a server to accept client connections, it must be bound to a network address within the system. The following code demonstrates how to bind a socket that has already been created to an IP address and port. Client applications use the IP address and port to connect to the host network.

To bind a socket

- The sockaddr structure holds information regarding the address family, IP address, and port number. Call the bind function, passing the created socket and sockaddr structure returned from the getaddrinfo function as parameters. Check for general errors.

```
// Setup the TCP listening socket

iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);

if (iResult == SOCKET_ERROR) {

printf("bind failed with error: %d\n", WSAGetLastError());

freeaddrinfo(result);

closesocket(ListenSocket);

WSACleanup();

return 1;

}
```

- Once the bind function is called, the address information returned by the getaddrinfo function is no longer needed. The freeaddrinfo function is called to free the memory allocated by the getaddrinfo function for this address information.

```
freeaddrinfo(result);
```

Listening on a Socket

- After the socket is bound to an IP address and port on the system, the server must then listen on that IP address and port for incoming connection requests.

To listen on a socket

- Call the listen function, passing as parameters the created socket and a value for the backlog, maximum length of the queue of pending connections to accept. In this example, the backlog parameter was set to SOMAXCONN. This value is a special constant that instructs the Winsock provider for this socket to allow a maximum reasonable number of pending connections in the queue.

```
if ( listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR ) {
printf( "Listen failed with error: %ld\n", WSAGetLastError() );
closesocket(ListenSocket);
WSACleanup();
return 1;
}
```

Accepting a Connection

- Once the socket is listening for a connection, the program must handle connection requests on that socket.

To accept a connection on a socket

1. Create a temporary SOCKET object called ClientSocket for accepting connections from clients.

```
SOCKET ClientSocket;
```

2. Normally a server application would be designed to listen for connections from multiple clients. For high-performance servers, multiple threads are commonly used to handle the multiple client connections. One programming technique is to create a continuous loop that checks for connection requests using the listen function. If a connection request occurs, the application calls the accept, AcceptEx, or WSAAccept function and passes the work to another thread to handle the request.

```
ClientSocket = INVALID_SOCKET;

// Accept a client socket

ClientSocket = accept(ListenSocket, NULL, NULL);

if (ClientSocket == INVALID_SOCKET) {

printf("accept failed: %d\n",WSAGetLastError());

closesocket(ListenSocket);

WSACleanup();

return 1;

}
```

3. When the client connection has been accepted, a server application would normally pass the accepted client socket to a worker thread or an I/O completion port and continue accepting additional connections.
   - There are a number of other programming techniques that can be used to listen for and accept multiple connections.
   - These include using the select or WSAPoll functions.

Note: On Unix systems, a common programming technique for servers was for an application to listen for connections. When a connection was accepted, the parent process would call the fork function to create a new child process to handle the client connection, inheriting the socket from the parent. This programming technique is not supported on Windows, since the fork function is not supported. This technique is also not usually suitable for high-performance servers, since the resources needed to create a new process are much greater than those needed for a thread.

RECEIVING AND SENDING DATA ON THE SERVER

To receive and send data on a socket

```
#define DEFAULT_BUFLEN 512
char recvbuf[DEFAULT_BUFLEN];
int iResult, iSendResult;
int recvbuflen = DEFAULT_BUFLEN;
// Receive until the peer shuts down the connection
do {
iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
if (iResult > 0) {
printf("Bytes received: %d\n", iResult);
// Echo the buffer back to the sender
iSendResult = send(ClientSocket, recvbuf, iResult,
0);
if (iSendResult == SOCKET_ERROR) {
printf("send failed: %d\n", WSAGetLastError());
closesocket(ClientSocket);
WSACleanup();
return 1;
}
printf("Bytes sent: %d\n", iSendResult);
} else if (iResult == 0)
printf("Connection closing...\n");
else {
printf("recv failed: %d\n", WSAGetLastError());
closesocket(ClientSocket);
WSACleanup();
return 1;
}
} while (iResult > 0);
```

The send and recv functions both return an integer value of the number of bytes sent or received, respectively, or an error.

DISCONNECTING THE SERVER

- Once the server is completed receiving data from the client and sending data back to the client, the server disconnects from the client and shutdowns the socket.

To disconnect and shutdown a socket

1. When the server is done sending data to the client, the shutdown function can be called specifying SD_SEND to shutdown the sending side of the socket. This allows the client to release some of the resources for this socket. The server application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ClientSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
printf("shutdown failed: %d\n",WSAGetLastError());
closesocket(ClientSocket);
Compiled by Chandra Mohan Jayaswal
WSACleanup();
return 1;
}
```

2. When the client application is done receiving data, the closesocket function is called to close the socket. When the client application is completed using the Windows Sockets DLL, the WSACleanup function is called to release resources.

```
// cleanup

closesocket(ClientSocket);

WSACleanup();

return 0;
```

END OF CHAPTER 3