Network Programming

Lecture slide III

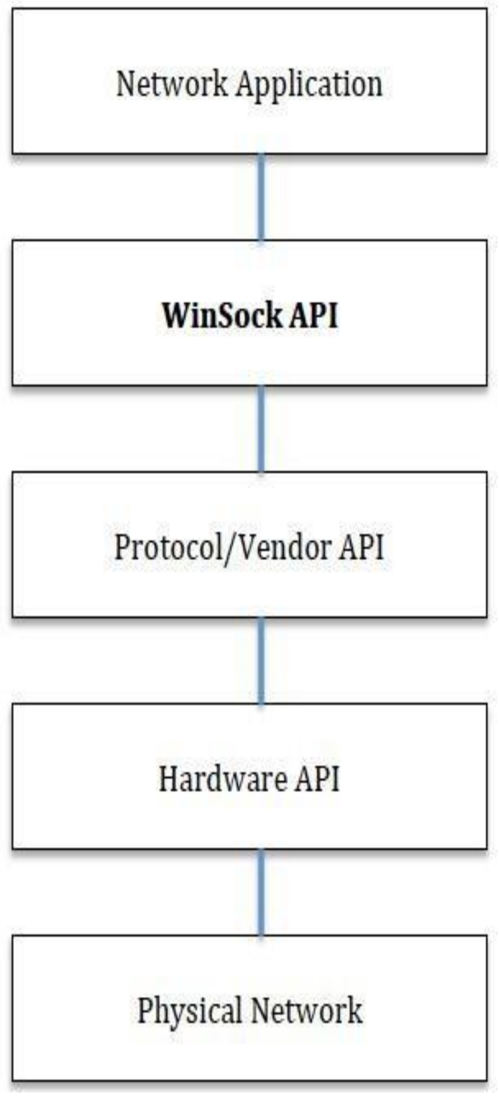
Introduction to Winsock

- In UNIX, sockets follow the **Berkeley (BSD**) model closely
- Windows adapted the BSD socket API into WinSock:
- Winsock 1.x was a close adaptation of the BSD API
- Winsock 2.x added new features
  - ❑ Asynchronous calls & callbacks
  - ❑ Overlapped I/O
  - ❑ The layered service provider (LSP) architecture

Introduction to Winsock

- **WinSock** is short for **Windows Sockets**, and is used as the interface between TCP/IP and Windows
- 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).
- WinSock is a .DLL (Dynamic Link Library) and runs under Windows.
- **WINSOCK.DLL** is the interface to TCP/IP and, from there, on out to the Internet.

Introduction to Winsock

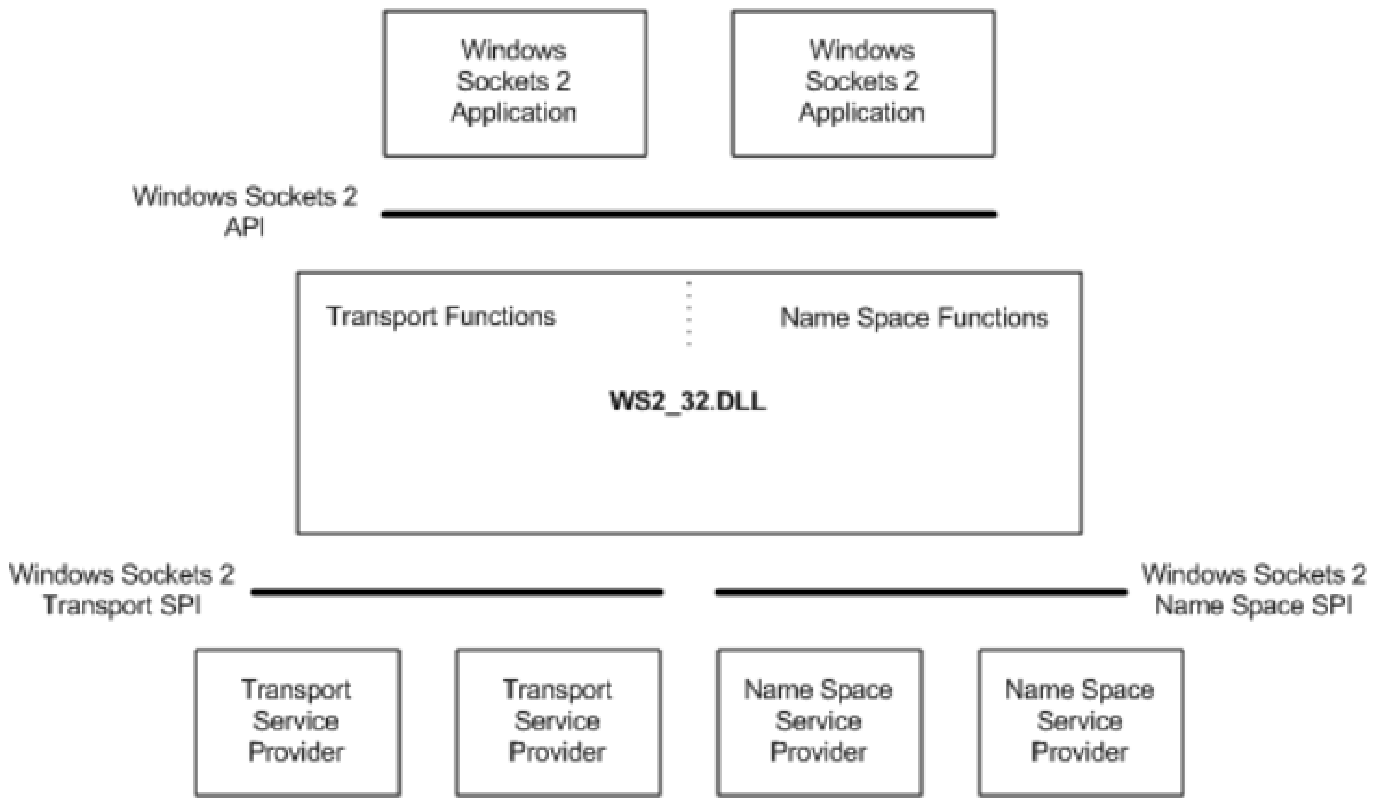


- The WinSock specification allows TCP/IP stack vendors to provide a consistent interface of their stacks so that application developers can write an application to the **WinSock specification** and have that application run on any vendor's WinSock-compatible TCP/IP protocol stack.

- 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 2 Architecture

- The Windows Sockets 2 architecture is compliant with the Windows Open System Architecture (WOSA), as illustrated below:


Windows Sockets 2 Application
Windows Sockets 2 Application
Windows Sockets 2 API
Transport Functions
Name Space Functions
WS2_32.DLL
Windows Sockets 2 Transport SPI
Windows Sockets 2 Name Space SPI
Transport Service Provider
Transport Service Provider
Name Space Service Provider
Name Space Service Provider

- Winsock defines a **standard service provider interface (SPI)** between the application programming interface (API), with its functions exported from WS2_32.dll and the protocol stacks. Consequently, Winsock support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1.

- With the Windows Sockets 2 architecture, it is not necessary or desirable, for stack vendors to supply their own implementation of WS2_32.dll, since a single WS2_32.dll must work across all stacks.

- The WS2_32.dll and compatibility shims should be viewed in the same way as an operating system component.

WinSock 2 Architecture

New features that WinSock 2 provides

- **Multiple Protocol support:** WOSA architecture let's service providers "plug-in" and "pile-on"
- **Transport Protocol Independence:** Choose protocol by the services they provide
- **Multiple Namespaces:** Select the protocol you want to resolve hostnames, or locate services
- **Scatter and Gather:** Receive and send, to and from multiple buffers
- **Overlapped I/O and Event Objects:** Utilize Win32 paradigms for enhanced throughput
- **Quality of Service*:** Negotiate and keep track of bandwidth per socket or socket group**
- **Socket Groups*:** Group sockets within an application, and prioritize them
- **Multipoint and Multicast:** Protocol independent APIs and protocol specific APIs
- **Conditional Acceptance*:** Ability to reject or defer a connect request before it occurs
- **Connect and Disconnect data*:** For transport protocols that support it
- **Socket Sharing:** Two or more processes can share a socket handle
- **Vendor IDs and a mechanism or vendor extensions:** Vendor specific APIs can be added
- **Layered Service Providers:** The ability to add services to existing transport providers

* not implemented yet

** no QOS templates defined yet (for WSAGetQOSByName())

WinSock 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/Interne 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

Type 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.
- he 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.

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

Blocking : https://learn.microsoft.com/en-us/windows/win32/winsock/blocking-i-o-2

Non Blocking https://learn.microsoft.com/en-us/windows/win32/winsock/nonblocking-i-o-2

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.

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);

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.

Function for Handling Blocked I/O

```
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:

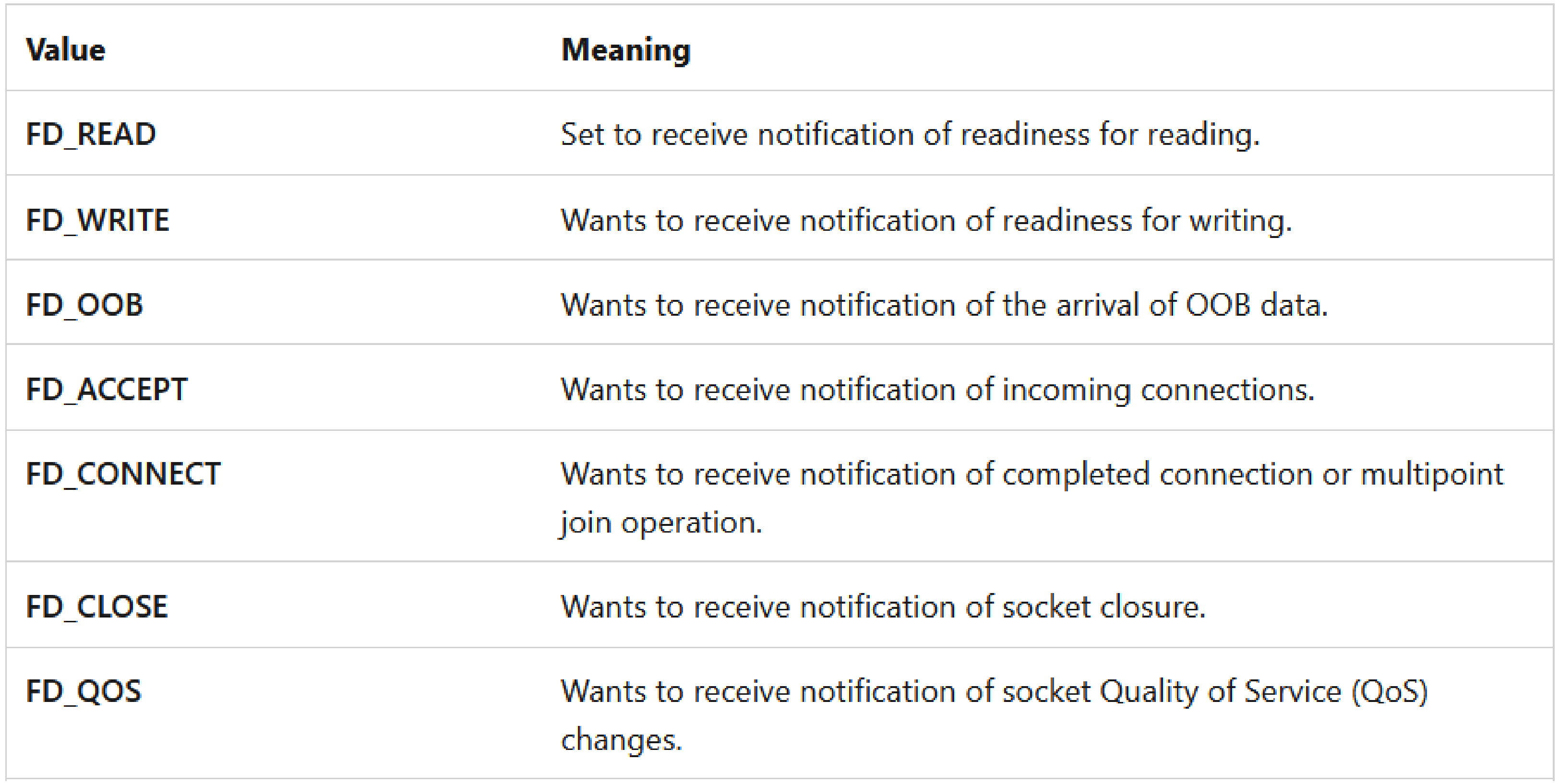
Function for Handling Blocked I/O

```
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 . **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.

Function for Handling Blocked I/O

The ***lEvent*** parameter is constructed by using the bitwise OR operator with any value listed in the following table



Value	Meaning
FD_READ	Set to receive notification of readiness for reading.
FD_WRITE	Wants to receive notification of readiness for writing.
FD_OOB	Wants to receive notification of the arrival of OOB data.
FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket Quality of Service (QoS) changes.

More : https://learn.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsaasyncselect

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);
```

Asynchronous Database function

- **Unsigned Long Integer Conversion**
- The **htonl()** and **ntohl()** functions work like htons() and ntohs() except that they operate on four-bytemunsigned 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 an IP Address String to Binary**

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

  - **Converting a Binary IP Address to a String**

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

Asynchronous Database function

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));
```

Asynchronous Database function

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());
}
```

Asynchronous Database function

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);
```

Asynchronous Database function

- **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 **hosten**t structure for the desired host.
- **buflen** is the size of the **buf** buffer. It should be MAXGETHOSTSTRUCT for safety's sake.

Asynchronous Database function

Canceling an Outstanding Asynchronous Request

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);
```

Asynchronous Database function

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);
```

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.

Asynchronous Database function

Asynchronously Finding a Service’s Port Number

WSAAsyncGetServByName() is the asynchronous counterpart to **getservbyname().** Its function prototype is as follows:

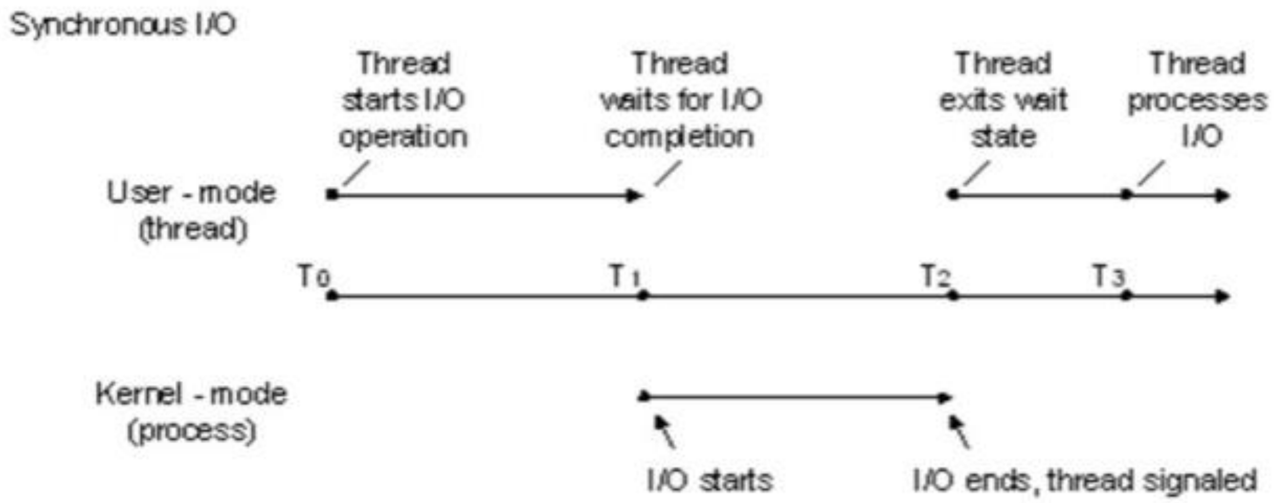
```
HANDLE WSAAsyncGetServByName(HWND hWnd, u_int wMsg, const char
* name, const char * proto, char * buf, int buflen);
```

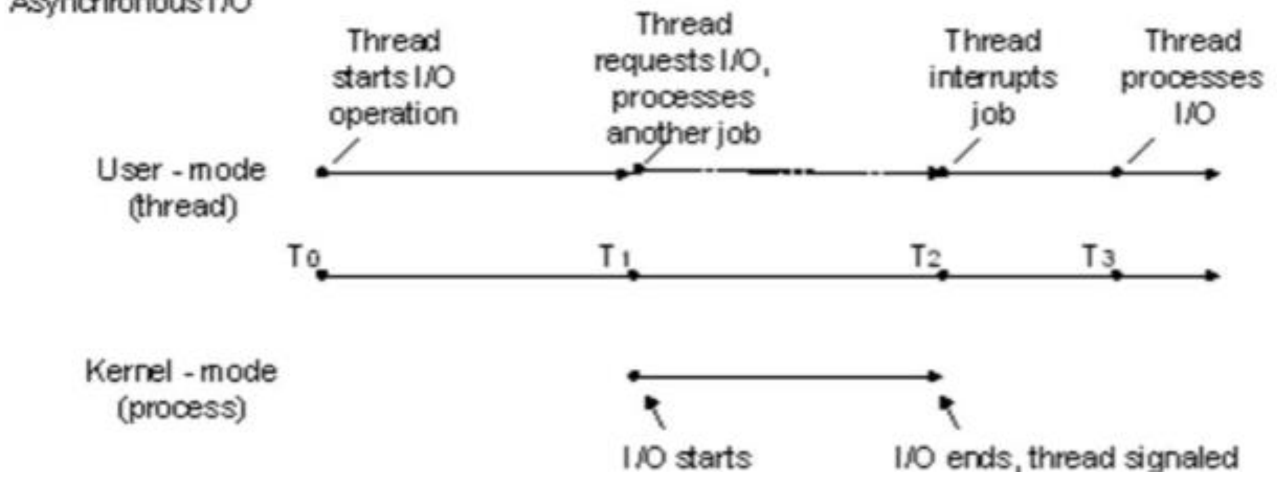
Asynchronously Finding a Service Name When You Know Its Port Number

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);
```

Synchronous and Asynchronous I/O





Asynchronous I/O function

- 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.

Asynchronous I/O function

- 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

Asynchronous I/O function

```
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.

WSASEND()

The **WSASend** function provides functionality over and above the standard send function in two important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped send operations.

2. It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

WSASENDTO()

The **WSASendTo** function sends data to a specific destination, using overlapped I/O where applicable.

```
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).

WSASENDTO()

- **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

The **WSASendTo** function provides enhanced features over the standard sendto function in two important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped send operations.
2. It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

More : https://learn.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsasendto

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 beencompleted (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
);
```

WSARECVFROM()

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() FUNCTION

```
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.

dwloControlCode [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

WSAIOCTL() FUNCTION

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 thecbTransferred 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.

ERROR HANDLING FUNCTIONS;ASYNCHRONOUS OPERATION

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 bylpcbTransfer 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.**

1. 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).
2. 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).
3. 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.

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.