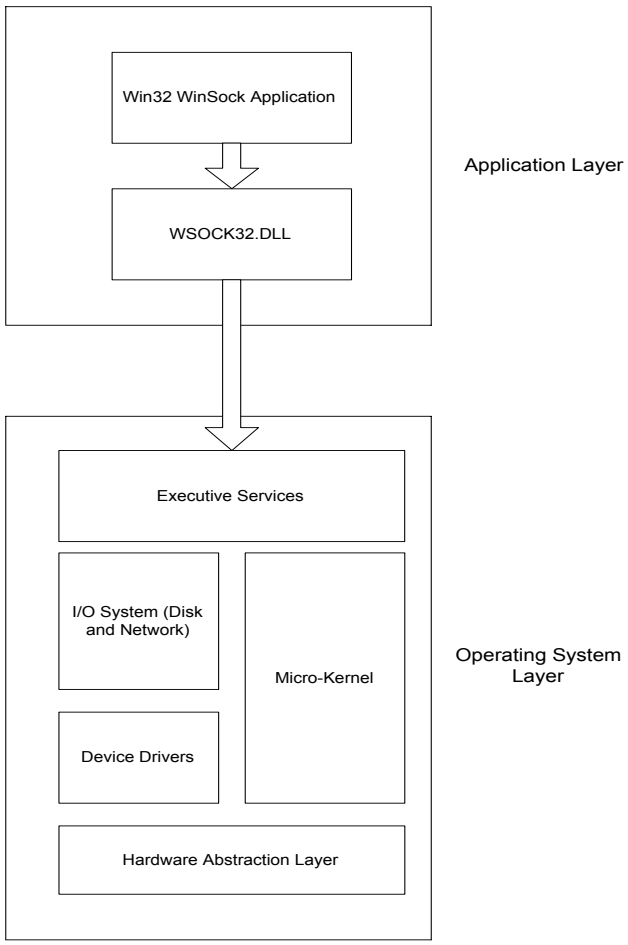
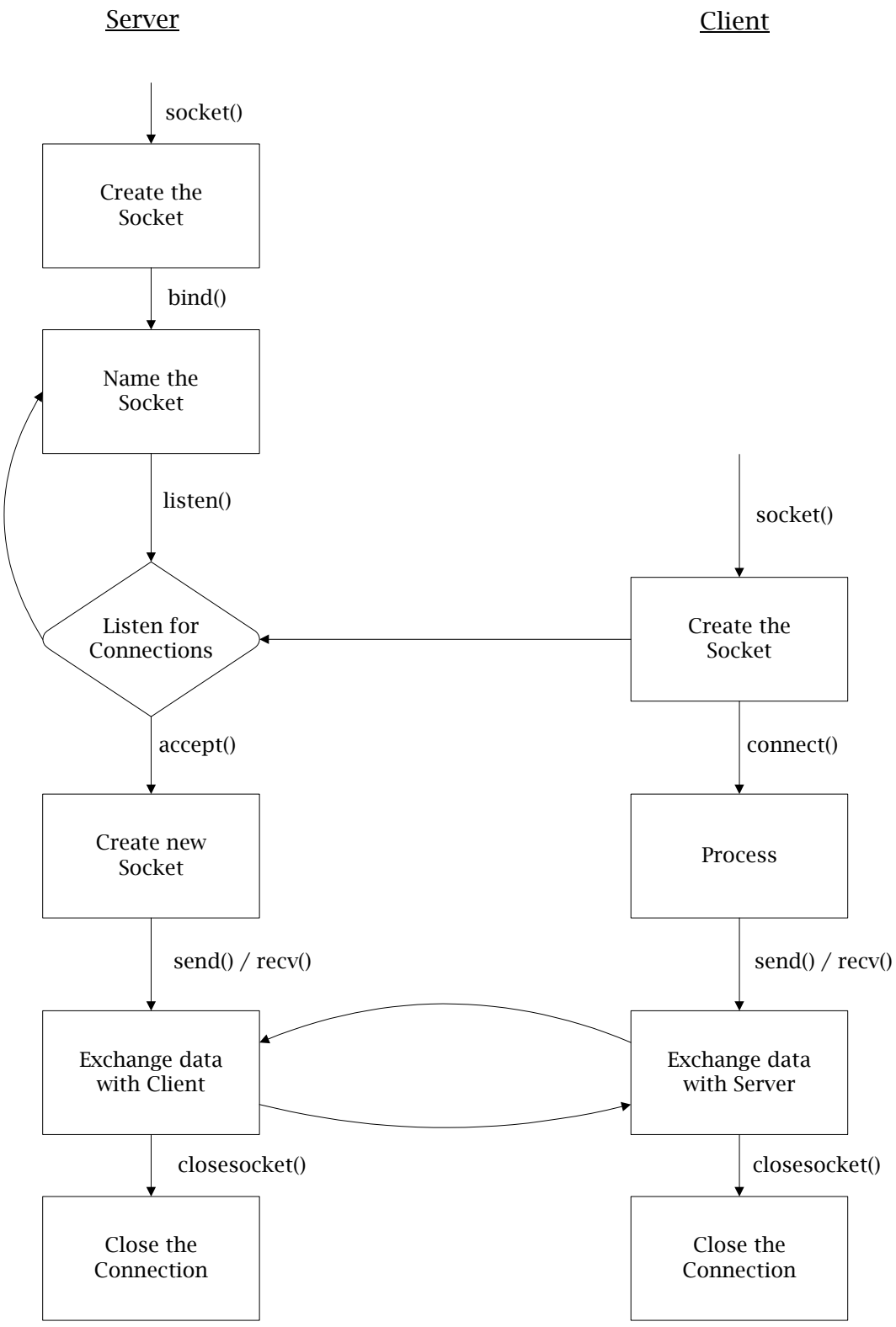
Windows Sockets Applications Programming Interface (API)

- The **Windows Sockets API** (**WinSock API**) is a library of functions that implements the socket interface popularized by the **Berkeley Socket** implementation for **UNIX**.
- **WinSock** is bound to the **TCP/IP** protocol suite and allows **TCP/IP stack** developers to produce applications that will run on any vendor's **WinSock**-compatible **TCP/IP** protocol stack.
- The **WinSock** specification augments the **Berkeley socket** implementation by adding **Windows**-specific extensions to support the **message-driven** nature of the **Windows** operating system.
- Application programmers write to the **WinSock** API and link their applications with the **WSOCK32.LIB** library.
- The **WinSock** API is supported by the WS2_32.DLL Dynamic Link Library.
- The diagram below shows the relative position of **WSOCK32.DLL** to a **32-bit** program on a **Windows** system.



- A **TCP/IP socket** is simply an end-point of communication, and is comprised of an **IP** address and a **port**.
- Some **ports** are reserved for well-known services and others are for use by your applications.
- Sockets can be set up to provide either a **reliable**, **connection-oriented** stream service or an **unreliable**, **connectionless** datagram service.
- The reliable stream socket is based on TCP. It requires that a connection be established before two processes can exchange data.
- The connection-oriented stream service is well suited to the **client-server** architecture.
- The **server** creates a **socket**, gives the socket a **name**, and waits for **clients** to **connect** to the socket.
- The **client** creates a **socket** and connects to the **named socket** on the **server**.
- When the server detects the connection to the named socket, it creates a new socket and uses that new socket for communication with the client.
- The server's named socket continues to wait for more connections from other clients.
- The diagram below illustrates this process.


Server
Client
socket()
Create the Socket
bind()
Name the Socket
listen()
socket()
Listen for Connections
Create the Socket
accept()
connect()
Create new Socket
Process
send() / recv()
send() / recv()
Exchange data with Client
Exchange data with Server
closesocket()
closesocket()
Close the Connection
Close the Connection

WinSock Startup and Closure

- In **UNIX**, the network protocols are incorporated into the kernel, therefore applications are not required to initialize the networking parameters explicitly.

- **Windows** has no such standard implementation of network services.

- Some networking protocols are implemented as **Dynamic-Link Libraries** (**DLL**s) or as **Virtual Devices** (**VxD**s) which need to be initialized before they can be used by an application.

- **WinSock** requires that applications register their intent to use **WinSock** by calling the system function **WSAStartup** (**WSA**: Windows Sockets Asynchronous) before calling other **WinSock API**s.

- **WSAStartup** also allows applications and **WSOCK32.DLL** to negotiate the particular version of **WinSock** that each side supports.

- **WinSock** also requires that applications call the **WSACleanup** function when they finish using the WinSock services.

- The **WSAStartup()** function is defined as:

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

- The **wVersionRequired** parameter indicates the **highest** version of **WinSock** with which the application is known to work.

- The **major revision** number is in the **low-order byte** while the **minor revision** number is in the **high-order byte**.

- The **lpWSAData** is a pointer to a **WSADATA** structure, which returns information about the WinSock implementation to the caller of **WSAStartup**, defined as:

```
typedef struct WSAData {
        WORD            wVersion;
        WORD            wHighVersion;
        char            szDescription[WSADESCRIPTION_LEN+1];
        char            szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short  iMaxSockets;
        unsigned short  iMaxUdpDg;
        char FAR *      lpVendorInfo;
} WSADATA;
```

- The **iMaxSockets** field identifies the maximum number of sockets that an aplication can potentially use.

- The **iMaxUdpDg** field specifies the size, in bytes, the largest UDP datagram that can be sent or received by the application.

- The **WSACleanup** function is a relatively simple function:

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

Synchronous Versus Asynchronous

- **UNIX I/O** is synchronous, meaning that **UNIX I/O** operations occur at the same time as function calls.
- This implies that in UNIX, a read or write call will not return until the operation completes, that is, it will **block**.
- In **UNIX** however, one can specify a system call as **non-blocking**, or, as an **asynchronous** operation.
- In an asynchronous operation (including I/O), a program initiates an operation by calling a function that will return immediately.
- The asynchronous call is simply responsible for initiating the operation and after that it is the operating system's responsibility to monitor the completion of the operation an report success or failure to the calling process.
- Windows is an asynchronous operating system that sends a status message to a Window specified by the application during the initialization of the asynchronous call.
- Windows provides asynchronous functions as part of the **WinSock API** which can be used to develop WinSock (**TCP/IP**) applications.

Blocking and Non-Blocking Functions

- The nature of operating systems has a profound impact on the way networking operations are processed.
- Many networking operations frequently block. An example is when the **recv** function is called to read data from an empty socket.
- Blocking on the **recv** is unacceptable because it might some time for the data to arrive and in the meantime other functions would have to be performed.
- Instead of actually blocking, the **WinSock** call enters a polling loop, waiting for the network event to complete.
- Unfortunately polling does not work when used with database functions such as **gethostbyname**. This is simply due to the fact that the host lookups may take a very long time.
- **WinSock** designers created an alternative way to handle blocking network calls.
- Sockets can be marked non-blocking in which case the network call will return immediately with a status code indicating that it would have blocked.
- Later, **WinSock** sends the program a standard Windows message indicating that the blocking call may be attempted again and that this time it (probably) won't block.

- Using the **WSAAsyncSelect** function, the program can specify the types of socket events for which it wants to be notified.

- The **WSAAsyncSelect** function essentially changes the socket operation from blocking to non-blocking.

- The **WSAAsyncSelect** function is the only **asynchronous**, Windows-specific function in the WinSock specification that uses a **socket handle** as a parameter:

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

- **s** is the socket of interest.

- The **hWnd** parameter is the window handle of the window to which a message should be sent when the socket state changes.
- The **wMsg** parameter is the message to sent to the window as the window message.

- The **lEvent** parameter identifies the state changes that are of interest to the application for the specified socket.

- The **lEvent** parameter is created by a logical **OR** of the constants (defined in **socket.h**) listed below:

Event	Description
FD_ACCEPT	Sends a message when the socket has an incoming connection and accept can be called on the socket without blocking.
FD_CLOSE	Sends a message when the socket has been closed.
FD_CONNECT	Sends a message when the socket has completed its connection.
FD_OOB	Sends a message when Out-of-Band* data is available to be read.
FD_READ	Sends a message when there is data available to be read.
FD_WRITE	Sends a message when the socket is ready for writing.

* Out-of-Band data is usually used to indicate an exceptional error condition that must be processed immediately, before any data pending in the receive queue.

- **WinSock** sends only a **single message** for a given event. No more messages for that event will be sent until the application calls another WinSock function that **re-enables** the particular message.

- For example, if there is data to be read from the socket, and **FD_READ** is enabled, WinSock will send a single **FD_READ** event to the specified window.

- When the application calls a ***recv*** or ***recvfrom*** function, **FD_READ** event generation is re-enabled.

- This will cause another **FD_READ** message to be sent when more data arrives.

- The table below summarizes all the **re-enabling** functions for socket events.

Event	Functions that Re-enable Event Generation
FD_ACCEPT	Accept
FD_CLOSE	None - this can only occur once for a given socket.
FD_CONNECT	None - this can only occur once for a given socket.
FD_OOB	Recv
FD_READ	recv, recvfrom
FD_WRITE	send, sento

- Under Windows when a blocking **WinSock** function is called, the operating system suspends the current thread and task switches to another.
- Although Windows uses pre-emptive multitasking to keep the system running, the current thread will be suspended until the blocking call returns.
- If this is the only thread in the process, all user input is suspended for that process until the thread unblocks and can continue to dispatch window messages to the process.
- The solution to the above is to use **multiple threads** or the **extended**, **even-driven** WinSock API.
- The extension functions use the same names as the standard database functions but with the "**WSAAsync**" prepended and capitalization to reflect Windows' typical style.
- Thus, **gethostbyname** is changed to **WSAAsyncGetHostByName**.
- If the asynchronous database lookup was successfully **initiated**, these **extension functions** return a non-zero **asynchronous task handle**.
- This **task handle** can be used to cancel the operation at a later time, before it completes, or to match up the completion messages with the corresponding asynchronous operation.
- If the lookup could not be successfully **initiated**, the functions return **0**. The specific error code can be retrieved by calling **WSAGetLastError**.
- When the asynchronous lookup is complete, Windows Sockets sends a window message to the application window that was specified in the initiating database lookup function.
- The window message will be the window message that was specified in the initiating database lookup function.
- The **wParam** value contains the asynchronous task handle returned to the application by the database function.

- The high-word of **lParam** indicates any error that may have occurred during the lookup or zero if the lookup completes with no errors.

- **CAUTION**: Do not forget to use the **WSAGETASYNCERROR()** macro to check for an error in your message handlers for the asynchronous calls.

Conversion Routines and Network Byte Ordering

- Different networked hosts will most likely have different hardware architectures based on the CPU used in the computer.

- Each different CPU architecture will store internal numerical data differently, based on its **byte ordering**.

- An network using WinSock must allow these disparate computers to communicate using a **common addressing format**.

- To facilitate the different byte ordering used in various CPUs, WinSock provides a set of conversion functions that convert **host byte-ordered numbers** into numbers using the **network byte-ordering** scheme.

- Network byte ordering is the common standard by which all **TCP/IP** connected computers must transmit certain data.

Unsigned Short Integer Conversion

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

- These are prototyped as:

    u_short htons (u_short hostshort);
    u_short ntohs (u_short netshort);

- On the **Intel 80x86** CPUs, integers are stored with the least significant bit in the lower part of an integer's address space.

- The decimal number **43794** in **hex** is **AB12**. Thus on Intel architectures, the byte value of this number's memory location **n** is **12** and the byte value at **n+1** is **AB.**

- This format is the opposite of network byte ordering. The output of **htons (43794)** has **AB** in the **lower address space** and **12** stored in the **higher address space** of the two-byte quantity.

- On **Motorola 680XX** architectures, the **ntohs()** function does not do any byte manipulation because the native byte ordering is the same as network byte ordering.

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.

- These are prototyped as:

    u_long htonl (u_long hostlong);
    u_long ntohl (u_long netlong);

- On the **Intel 80x86** CPUs, the decimal number **2870136116** is stored in memory, from the lowest address space to the highest, as hexadecimal **34 CD 12 AB**.

- The output of **htonl (2870136116)** has **AB** in the lower address space, **12** stored in the next higher address space, and so on.

Converting IP Addresses

- WinSock provides a set of functions to translate between the ASCII representation of a **dotted-decimal IP address** and the **internal 32-bit, byte-ordered number** required by other WinSock functions.

- **inet_addr()** converts a **dotted-decimal IP address string** into a number suitable for use as an **Internet address**.

- Its function prototype is:

    unsigned long inet_addr (const char FAR * cp);

- **cp** is a pointer to a string representing am IP address in **dotted-decimal** notation.

- The function returns a binary representation of the Internet address in **network byte order**, so there is no need to call **htonl().**

Converting a Binary IP Address to a String

- **inet_ntoa()** performs an opposite to that of **inet_addr()**, that is, it returns a dotted-decimal representation of the IP address, given a host address obtained from the **inet_addr()** function.

- It as prototyped as:

    char FAR * inet_ntoa (struct in_addr in);

- **in** is a structure that contains an Internet host address.

- Note that the string pointer returned by **inet_ntoa()** is only temporary, and will most likely be invalid after the next call to a WinSock function. It is best to copy it into a variable in the application.

What's in a Name?

- Some applications may require the name of the computer on which they are running.
- The **gethostname()** function provides this functionality:

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

- The function returns 0 upon success, and **SOCKET_ERROR** upon failure. Call **WSAGetLastError()** to determine the specifics of the failure.
- **name** is a far pointer to a character array that will accept the null-terminated host name and **namelen** is the size of that character array.
- The **name** argument may be a simple name such as **valhalla**, or a fully qualified domain name such as **valhalla.bcit.ca**.

Host Name Resolution

- Most programs that have a configuration to select the host with which the program communicates enable the user to enter a **host name** or an **IP** address.
- Your program should call **inet_addr()** first with the user's input. If this returns successfully, the conversion is complete.
- Otherwise, the **gethostbyname()** function is called, assuming the user entered a host name.
- The **gethostbyname()** function takes a host name and returns its IP address.
- This function and its asynchronous counterpart **WSAAsyncGetHostByName()**, may perform a simple table lookup on a host file local to the computer, or it may send the request across the network to a **name server**.
- The prototypes for both are as follows:

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

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

- The argument **name** is common to both and is a far pointer to a null-terminated character array that contains the name of the computer about which you require host information.

- The **hostent** structure returned (upon success) has the following format:

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

- On a return value of **NULL** (failure), you can call **WSAGetLastError()** to get the specifics of the failure.

- The **buf** pointer should point to a **hostent** structure that is at least **MAXGETHOSTSTRUCT** bytes in length.

Canceling an Outstanding Asynchronous Request

- The **WSACancelAsyncRequest()** function can be used to terminate a database lookup using the handle returned by asynchronous database functions.

- It is prototyped as:

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

- **hAsyncTaskHandle** is the handle to the asynchronous task you wish to abort.

- This capability is both useful and desirable when a database query is taking an excruciatingly long time to complete, as it might when networked name servers are involved.

IP Address Resolution

- The goal of **IP address resolution** is to get the host name, and other host information when you know the host name when you know the IP address.

- The **gethostbyaddr()** and **WSAAsyncGetHostByAddr()** functions are used to achieve this goal:

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

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

- The **addr** argument points to an Internet address in **network byte order**.

- The **len** parameter specifies the length of the address pointed to by **addr**; its value is always **4** for Internet addresses.

- The **type** parameter indicates the type of address specified by **addr** and must be **PF_INET** for Internet addresses.

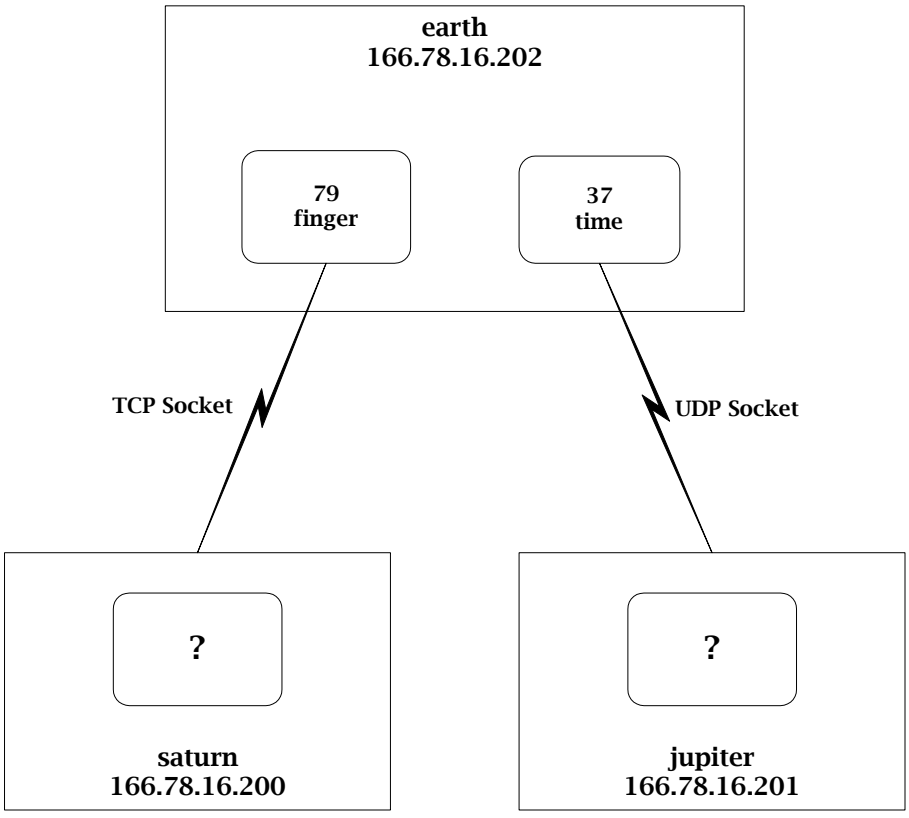
- The **buf** pointer should point to a **hostent** structure that is at least **MAXGETHOSTSTRUCT** bytes in length.

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetHostByAddr()** completes its asynchronous operation.

- **wMsg** is the user defined massage that will be posted to **hWnd** when the asynchronous operation is complete.

Service Name Resolution

- So far, we have seen how to retrieve a binary IP address, whether it be derived from a host name or host's IP address.
- This is only half the task of making a network connection between client and server applications.
- When a **host** is running a **server** application, it is said to be providing a service. Each service is uniquely identified by a well-known **port number**.
- The server program "**listens**" for connections on the well-known port and the client program opens a connection to that port.
- The **port numbers** must be **unique** to distinguish the many server programs that a host may provide.
- The **port numbers** must be **well-known** so that application programmers can request them by **name**.
- The diagram below shows a host that is providing **two services**: **port number 37** is acting as a **time server** over **UDP**, and **port number 79** is acting as a **finger server** over **TCP**.



- Note that the **ports** out of which the **clients** connect are **unknown**. Clients don't need to specify a port when they create their **outbound sockets** because the socket can be assigned a **unique port** at **runtime** by the **TCP/IP stack**.

- The **getservbyname()** and **WSAAsyncGetServByName()** functions are responsible for retrieving the **port number** when you know its **service name**.

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

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

- **name** is a pointer to a string that contains the service for which you are searching. The service name is either the official service name or an alias.

- **proto** is a pointer to a string that contains the **Transport protocol** to use; it's either "**udp**" or "**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 returned **servent** structure is defined as follows:

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

- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetServByName()** completes its asynchronous operation.

- **wMsg** is the user defined massage that will be posted to **hWnd** when the asynchronous operation is complete.

- The **WSAAsyncGetServByName()** function is much like the asynchronous functions discussed earlier.

- The **buf** pointer should point to a **servent** structure that is at least **MAXGETHOSTSTRUCT** bytes in length.

The Services File

- The service name to port translation is commonly supported by flat database called the **services** file.

- A partial listing of a common services file is as follows:

```
# This file contains port numbers for well-known services
# as defined by RFC 1060 (Assigned Numbers).
#
# Format:
# <service name>     <port number>/<protocol> [aliases....]     [#<comment>]

time                 37/tcp         timserver
time                 37/udp         timserver
finger               89/tcp
```

- The **RFC 1060** is a standard for well-known ports that ensures that nobody creates duplicate services that utilize already established port numbers.

- When you create your own custom servers, you need to allocate a port number between **1024** and **5000** exclusive.

- The port numbers from **1024** and below are reserved for universally well-known ports.

- Note that the **time** service responds to **port 37** on both **UDP** and **TCP** connections.

- The port number/protocol pair provides for a unique correlation to a service within each transport protocol.

- That is to say that a service might respond to **UDP port 100**, and a completely different service might respond to **TCP port 100**.

- It is also possible for a specific service to respond to one UDP port and an entirely different TCP port.

- For custom applications that you will produce, it is sufficient to simply refer to a service number by its port number.

- This will free you from having to make an entry in the services table for you custom services.

- Make your server and client applications configurable with respect to the port numbers they use.

- Allow the server to listen for connections on a configurable port and make sure that the port to which the client connects is also configurable.

Port Resolution

- **Port resolution** is the opposite of service name resolution.
- The goal here is to find the corresponding **named service**, given a **port number** and **transport protocol**.
- The **getservbyport()** and **WSAAsyncGetServByPort()** functions are used to achieve this goal:

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

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

- **port** is the **service port**, in network byte order, of the service about which you want information.
- **proto** is a pointer to a string that contains the **Transport protocol** to use; it's either "**udp**" or "**tcp**" or **NULL**.
- A **NULL proto** will match on the first database entry matching **port and** return it.
- **hWnd** is the handle to the window to which a message will be sent when **WSAAsyncGetServByPort()** completes its asynchronous operation.
- **wMsg** is the user defined massage that will be posted to **hWnd** when the asynchronous operation is complete.
- The **buf** pointer should point to a **servent** structure that is at least **MAXGETHOSTSTRUCT** bytes in length.