# Network Programming

**BESE-VI – Pokhara University**


Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

# Chapter 4:
# Basics of Winsock Network Programming
# (6 hrs)

# Outline

1. Introduction to Winsock
   - Overview of the Winsock API
   - Differences between Unix and Winsock programming
   - Winosock DLL
     - Setting Up Winsock Environment
     - Initialization: WSAStartup()
     - Clean-up: WSACleanup()
2. Basic Winsock API Functions
   - Socket creation and binding (socket(), bind())
   - Listening and accepting connections (listen(), accept())
   - Sending and receiving data (send(), recv())
   - Closing a socket (closesocket())
   - Functions for handling Blocked IO
   - Asynchronous IO functions
3. Creating Simple TCP/UDP Clients and Servers programs using Winsock

# Introduction to Winsock

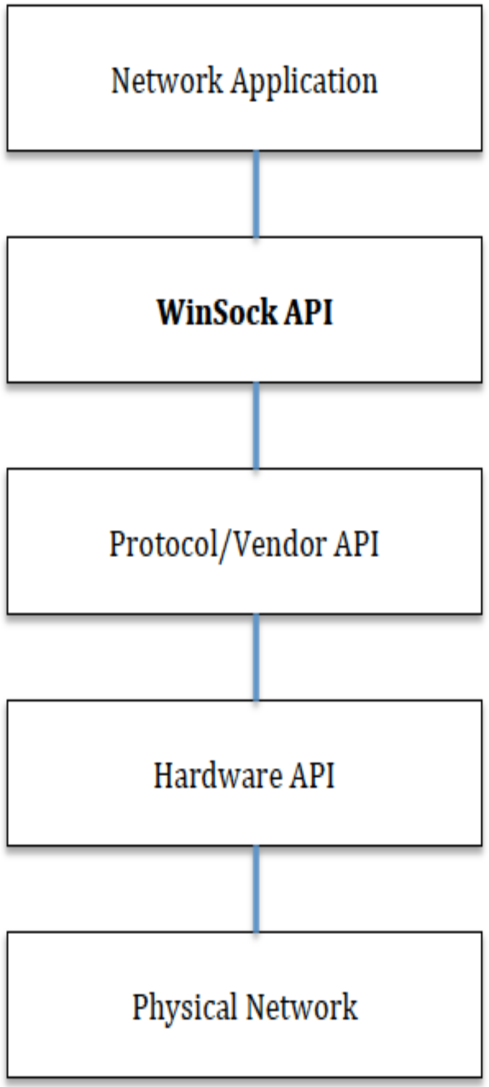
# Introduction to Winsock

- The *Windows Sockets Application Programming Interface* (WinSock API) is a library of functions that implements the socket interface.
- **Winsock (Windows Sockets API)** is a specification for network programming on **Windows**, providing a **standard interface** for TCP/IP.
- Based on the **Berkeley Sockets API** (from UNIX), but includes Windows-specific extensions.

## Key Features

- Support for **TCP/UDP** networking.
- Uses **sockets** as abstractions for communication endpoints.
- Provides **asynchronous (non-blocking)** and **event-driven** models.
- Facilitates **client-server** communication over networks.

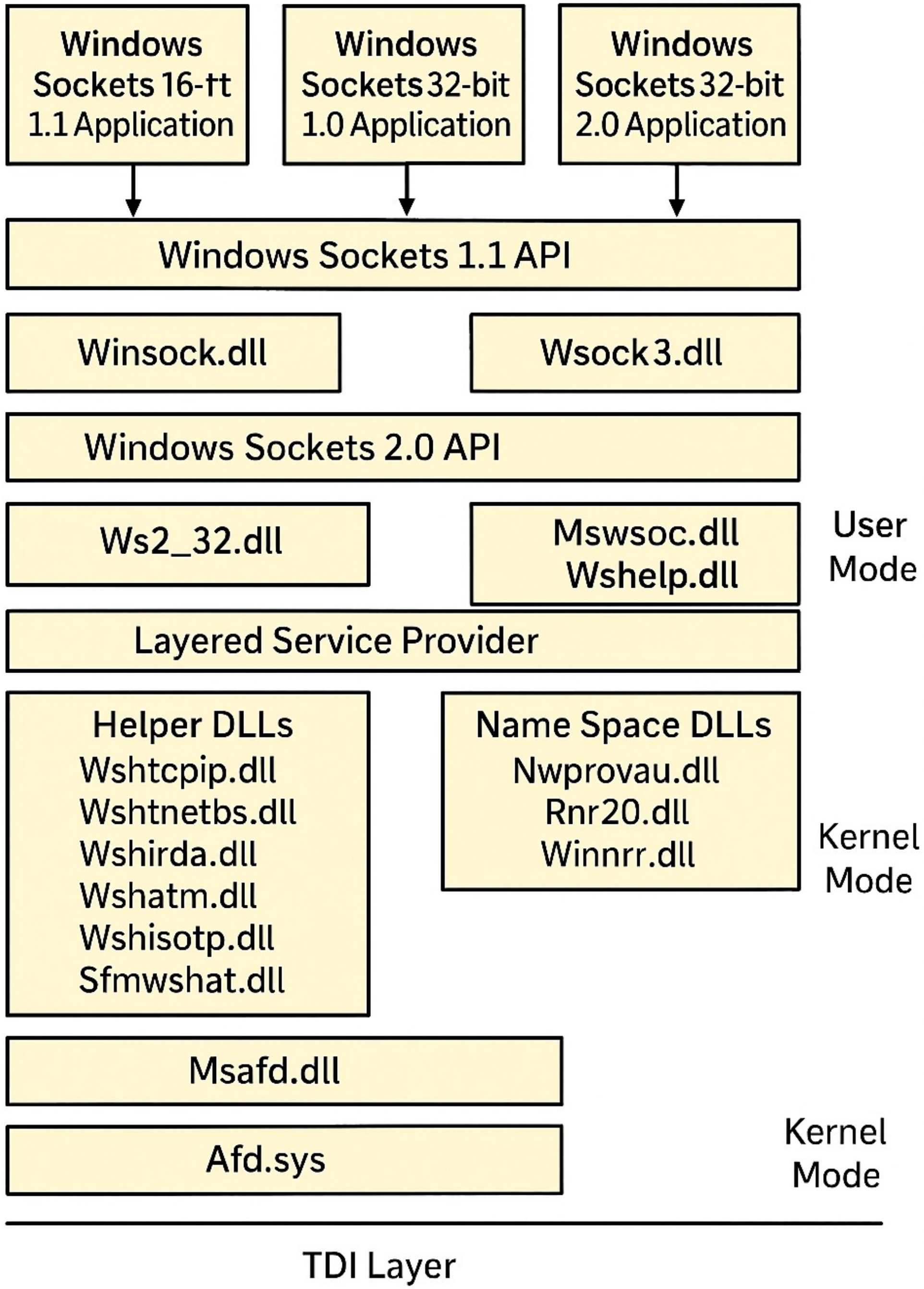
# Introduction to Winsock




- The WinSock specification standardizes the interface for TCP/IP stacks, enabling application developers to write code once and run it on any WinSock-compliant implementation.

- Before WinSock, developers had to link their applications to vendor-specific libraries, making it difficult to support multiple TCP/IP stacks due to differences in Berkeley sockets implementations.

# WinSock Architecture




**1. Application Layer (Top Layer)**

- Windows Sockets 16-bit 1.1 Application
- Windows Sockets 32-bit 1.1 Application
- Windows Sockets 32-bit 2.0 Application
  - These are user programs using Winsock for network communication.
  - Each application interfaces with the corresponding version of the Winsock API.

**2. API Layer**

- **Winsock 1.1 API**:
  - winsock.dll (for 16-bit apps)
  - wsock32.dll (for 32-bit apps)
- **Winsock 2.0 API**:
  - ws2_32.dll: Primary API for modern network apps
  - mswsock.dll, wshelp.dll: Extended and helper API support
  - This layer provides the application programming interface (API) for sockets.

# WinSock Architecture

## 3. Service Provider Interface (SPI)

- **Layered Service Provider**
  - Sits between API and protocol drivers.
  - Can be extended/overridden by developers for filtering or custom network behavior.
  - Communicates with protocol-specific helper DLLs.

## 4. Service Provider Layer

- **Helper DLLs (Protocol-Specific)**
- Examples:
  - wshtcpip.dll – TCP/IP
  - wshnetbs.dll – NetBIOS
  - wshirda.dll – Infrared
  - wshatm.dll, wshisn.dll, wshisotp.dll, sfmwshat.dll
  - Implements protocol-specific functionality.

- **Namespace Provider DLLs (Name Resolution)**
- Examples:
  - nwprovau.dll, rnr20.dll, winrnr.dll
  - Used for resolving names to network addresses (like DNS or directory services).

# WinSock Architecture

## 5. Base Service Provider

- **Msafd.dll**
  - Microsoft Ancillary Function Driver: Bridges user-mode to kernel networking.
- **Afd.sys**
  - Kernel-mode driver for the Windows socket subsystem.

## 6. Kernel Layer (Bottom Layer)

- **TDI Layer (Transport Driver Interface)**
  - Interfaces with actual transport protocols like TCP/IP, NetBIOS, etc.
  - Now deprecated but crucial in legacy systems.

# WinSock Architecture

| Layer | Components | Role |
|---|---|---|
| Application Layer | Winsock Apps (16-bit, 32-bit, 2.0) | Uses Winsock APIs |
| API Layer | winsock.dll, wsock32.dll, ws2_32.dll, etc. | Provides socket functions |
| SPI Layer | Layered Service Provider | Intermediary between API and protocols |
| Service Provider Layer | Helper DLLs, Namespace DLLs | Implements protocols and name resolution |
| Base Provider Layer | msafd.dll, afd.sys | Socket support and kernel bridge |
| Kernel Layer | TDI Layer | Interfaces with transport protocols (legacy) |

# Dynamic Linking

# Dynamic Linking

- Dynamic linking refers to loading and linking libraries at **runtime** rather than **compile time**. It allows executables to use **shared code** from external modules.

**1. Implicit Dynamic Linking (load-time dynamic linking)**

- The application is **linked to DLLs during program startup**.
- DLLs are specified at **compile/link time** and automatically loaded by the OS when the application starts.

```
#include <windows.h>
#pragma comment(lib, "user32.lib") // linked at load-time
```

**Advantages:**

- Simple to use.
- Errors due to missing functions are detected during startup.
- Less manual code to manage linking.

**Disadvantages:**

- If DLL is missing or corrupted, the program fails to start.
- All DLLs are loaded even if not all functions are used.

# Dynamic Linking

## 2. Explicit Dynamic Linking(run-time dynamic linking)

- The application **loads the DLL manually at runtime** using functions like ***LoadLibrary()*** and ***GetProcAddress()***.

```
HMODULE hLib = LoadLibrary("user32.dll");

FARPROC msgBox = GetProcAddress(hLib, "MessageBoxA");
```

## Advantages:

- Load DLLs only when needed → reduces memory usage.
- Better error handling (can continue if DLL is missing).
- Enables plugin architecture and modular programs.

## Disadvantages:

- More complex code.
- Runtime errors if function names or DLL paths are incorrect.
- Performance overhead due to dynamic lookup.

# Dynamic Linking

## General Advantages of Dynamic Linking

- Saves **memory**: Shared libraries used by multiple processes.
- Saves **disk space**: No need to bundle libraries with every program.
- Allows **updates** without recompiling applications.
- Reduces **program size**.

## General Disadvantages of Dynamic Linking

- **Dependency issues**: If a required DLL is missing or incompatible → program fails.
- **Security risk**: Susceptible to DLL hijacking or injection attacks.
- **Performance hit**: Function lookups are slower than static linking.

# Winsock vs Berkeley Socket

# Winsock vs Berkeley Socket

| Feature / Aspect | Unix Sockets (POSIX/BSD) | Winsock (Windows Sockets API) |
|---|---|---|
| Platform | Unix, Linux, macOS | Windows only |
| Initialization | None | WSAStartup() / WSACleanup() required before/after any socket calls |
| Header Files | <sys/socket.h>, <netinet/in.h>, <arpa/inet.h> | <winsock2.h>, <ws2tcpip.h> |
| Library Linking | Standard C library | Ws2_32.lib (and optionally Mswsock.lib) |
| Socket Descriptor Type | int | SOCKET (an opaque unsigned type); invalid value is INVALID_SOCKET |
| Close Function | close(fd) | closesocket(sock) |
| Error Reporting | errno (e.g. EAGAIN, ECONNREFUSED) | WSAGetLastError() returns Winsock-specific codes (e.g. WSAEWOULDBLOCK, WSAECONNREFUSED) |
| Blocking / Nonblocking | fcntl() or ioctl(fd, FIONBIO, &flag) | ioctlsocket() |
| Asynchronous I/O | select(), poll(), epoll, nonblocking + signals (SIGIO), aio_* | Overlapped I/O (WSASend, WSARecv + OVERLAPPED), WSAAsyncSelect(), WSAEventSelect(), I/O Completion Ports |

# Winsock vs Berkeley Socket

| Feature / Aspect | Unix Sockets (POSIX/BSD) | Winsock (Windows Sockets API) |
|---|---|---|
| select()Support | Works on any file descriptor: sockets, pipes, files. | **Only** on sockets. FD_SETSIZE default limit 64 (can be recompiled). No monitoring of stdin or pipes. |
| Socket Options API | setsockopt(), getsockopt() | Same calls, but option names and levels (e.g. SOL_SOCKET, IPPROTO_TCP) can differ slightly. |
| Name Resolution | getaddrinfo(), gethostbyname() | getaddrinfo(), gethostbyname(), plus Win32 DNS APIs |
| IPv6 Support | Native from Linux kernel ≥2.2 / BSD variants | Supported since Winsock 2.0 |
| File I/O Semantics | Sockets are file descriptors → can use read(), write(), dup(), select() | Sockets are not normal file descriptors; must use Winsock calls (send(), recv()) |
| Signals | SIGPIPE on write to closed socket (can be disabled with MSG_NOSIGNAL) | No SIGPIPE; write to closed socket returns SOCKET_ERROR+ WSAGetLastError()==WSAECONNRESET |
| IPC Mechanisms | Sockets plus pipes / FIFOs, System V IPC (message queues, semaphores, shared memory) | Sockets plus Windows IPC: Named Pipes, Mailslots, LPC/ALPC, Shared Memory, RPC, COM |
| DLL Injection / Hijacking | N/A (shared libs loaded at launch) | Risk of DLL pre-loading/hijacking if path not fully qualified |

# Winsock vs Berkeley Socket

## Behavior on Signal / Async Events

- **Unix:** Interrupted by signals (EINTR).
- **Winsock:** Not signal-driven; overlapped or window-message notifications via *WSAAsyncSelect().*

## Summary of Key Conceptual Differences:

- **Initialization:** Winsock needs manual startup/cleanup; Unix doesn't.
- **Error Reporting:** Unix uses global ***errno***; Winsock uses function ***WSAGetLastError().***
- **Close Operation:** Unix uses *close(),* Winsock uses *closesocket()* to avoid interfering with file descriptors.
- **Handles:** Unix sockets are regular file descriptors; Winsock sockets are distinct from file I/O.

# Setting Up Winsock Environment

# Setting Up Winsock Environment

## 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.
- *Header*: `#include<winsock2.h>`
- *Library*: link against *Ws2_32.lib*

```
int WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);
/* Returns 0 on success. On failure,returns a nonzero error
code(e.g. WSASYSNOTREADY, WSAVERNOTSUPPORTED, WSAEINPROGRESS)*/
```

- *wVersionRequired*: The highest version of Winsock your application can use, encoded as *MAKEWORD(major, minor)*
- *lpWSAData*: Pointer to a WSADATA structure that will be filled in with details of the Winsock implementation loaded.

# WSADATA

```
#define WSADESCRIPTION_LEN 256
#define WSASYS_STATUS_LEN 128
typedef struct WSAData {
 WORD wVersion; // Winsock version requested by the application
 WORD wHighVersion; // Highest version supported by the loaded DLL
 char szDescription[WSADESCRIPTION_LEN + 1]; // Description string
 char szSystemStatus[WSASYS_STATUS_LEN + 1]; // Additional status (often unused)
 unsigned short iMaxSockets; // Maximum number of sockets supported
 unsigned short iMaxUdpDg; // Maximum datagram size for UDP
 char *lpVendorInfo; // Pointer to vendor-specific info (can be `NULL`)
} WSADATA, *LPWSADATA;
```

- wVersion: Holds the Winsock version granted to the application(low byte = major, high byte= minor).
- wHighVersion: Highest version of Winsock the underlying DLL can support.
- szDescription: A null-terminated ANSI string describing the Winsock implementation
- szSystemStatus: A null-terminated ANSI string for implementation-specific status
- iMaxSockets: On Windows 9x, indicates the maximum simultaneous sockets supported.
- iMaxUdpDg: On Windows 9x, maximum size in bytes of a UDP datagram.
- lpVendorInfo: If non-NULL, points to vendor-specific data or version strings.Rarely used; can be treated as opaque.

# Usage Example

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

int main(void) {
 WSADATA wsaData;
 int err;
 // Initialize Winsock version 2.2
 err = WSAStartup(MAKEWORD(2,2), &wsaData);
 if (err != 0) {
 printf("WSAStartup failed: %d\n", err);
 return 1;
 }
 printf("Winsock initialized:\n");
 printf(" Version: %d.%d\n",
 LOBYTE(wsaData.wVersion),
 HIBYTE(wsaData.wVersion));
 printf(" Description: %s\n", wsaData.szDescription);
// ... perform socket operations ...
 WSACleanup();
 return 0;
}
```

# 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.
- Purpose is to releases resources allocated by Winsock and decrements the per-process reference count of the Winsock DLL. When the reference count reaches zero, the Winsock implementation is unloaded.
- *Header*: #include<winsock2.h>
- *Library*: link against *Ws2_32.lib*

```
int WSACleanup(void);
/* Returns 0 on success SOCKET_ERROR on failure; call WSAGetLastError() to retrieve the error code (e.g. WSANOTINITIALISED, WSAENETDOWN). */
```

# 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.
- Example:

```
int result = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
if (result == SOCKET_ERROR) {
 int err = WSAGetLastError(); // Handle or report 'err'
}
```

- Always call *WSAGetLastError()* immediately after a Winsock call returns an error.
- Do not call other Winsock functions before retrieving the error, as they may overwrite the error code.

# Few Error Codes

| Error Code | Value | Description |
|---|---|---|
| WSAEINTR | 10004 | Interrupted function call |
| WSAEBADF | 10009 | Bad file/socket descriptor |
| WSAEACCES | 10013 | Permission denied |
| WSAEFAULT | 10014 | Bad address |
| WSAEINVAL | 10022 | Invalid argument |
| WSAEMFILE | 10024 | Too many open sockets |
| WSAEWOULDBLOCK | 10035 | Resource temporarily unavailable (nonblocking) |
| WSAEINPROGRESS | 10036 | Operation now in progress |
| WSAENOTSOCK | 10038 | Descriptor is not a socket |
| WSAEDESTADDRREQ | 10039 | Destination address required |
| WSAEADDRINUSE | 10048 | Address already in use |
| WSAECONNRESET | 10054 | Connection reset by peer |
| WSAETIMEDOUT | 10060 | Connection timed out |
| WSAECONNREFUSED | 10061 | Connection refused |
| WSAHOST_NOT_FOUND | 10001 | Authoritative answer host not found |

# Basic Winsock API Functions

# Basic Winsock API Functions

## Socket Creation and Binding

- **socket()**

```
SOCKET socket(
 int af, // Address family (e.g. AF_INET, AF_INET6)
 int type, // Socket type (SOCK_STREAM, SOCK_DGRAM)
 int protocol // Protocol (IPPROTO_TCP, IPPROTO_UDP, or 0)
);
```

*/* **Returns**: A SOCKET handle on success, or INVALID_SOCKET on error.*
***Errors**: Call WSAGetLastError() for codes like WSAENRTDOWN, WSAEAFNOSUPPORT. */*

- **bind():** Assigns a local address (IP + port) to a socket.

```
int bind(SOCKET s,const struct sockaddr *addr,int addrlen);
```

*/* **Returns**: 0 on success; SOCKET_ERROR on failure*
***Errors**: WSAEADDRINUSE, WSAEINVAL, etc */*

# Basic Winsock API Functions

## Example of socket() and bind()

```
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in local = {0};
local.sin_family = AF_INET;
local.sin_port = htons(12345);
local.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (SOCKADDR*)&local, sizeof(local)) == SOCKET_ERROR) {
 printf("bind failed: %d\n", WSAGetLastError());
 closesocket(s);
 WSACleanup();
 return 1;
}
```

## Establishing a Connection

- **connect():**

```
int connect(SOCKET s,const struct sockaddr *name,int namelen);
/*Returns: 0 on success (connection established).
 SOCKET_ERROR on failure; call WSAGetLastError() to get error code*/
```

# Basic Winsock API Functions

## Listening and Accepting Connections

- **listen():** Marks a bound TCP socket as passive, ready to accept incoming connections.

```
int listen( SOCKET s, int backlog);
/* Returns: 0 on success; SOCKET_ERROR on failure
 Errors: WSAEINVAL, WSAENOTSOCK. */
```

- **accept():** Extracts the first pending connection, returning a new connected socket.

```
SOCKET accept(SOCKET s,struct sockaddr *addr,int *addrlen);
/* Returns: New SOCKET on success; INVALID_SOCKET on failure.
 Errors: WSAEWOULDBLOCK (nonblocking), WSAENOTSOCK, etc*/
```

# Basic Winsock API Functions

## Example of listen() and accept()

```
listen(s, 5);
SOCKET client;
struct sockaddr_in peer;
int peerlen = sizeof(peer);
while ((client = accept(s, (SOCKADDR*)&peer, &peerlen)) != INVALID_SOCKET) {
 // handle client...
}
```

## Sending and Receiving Data

- **send()** and **recv():**

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

- flags is usually 0 or MSG_OOB, MSG_DONTROUTE

/***Returns**: Number of bytes sent/received(0=Connection closed),or SOCKET_ERROR.
**Errors**: WSAEWOULDBLOCK, WSAECONNRESET, etc. */

# Basic Winsock API Functions

## Closing a Socket

- **closesocket():** Gracefully closes a socket handle, releases resources.

```
int closesocket(SOCKET s);
/*Returns: 0 on success; SOCKET_ERROR on failure.
Errors: WSAENOTSOCK, WSAENETDOWN, etc. */
```

**Note:**

- Always call closesocket() before WSACleanup()
- After closing, the SOCKET handle is invalid and must not be reused.

**Example:**

```
char buffer[512];
int bytes = recv(client, buffer, sizeof buffer, 0);
if (bytes > 0) {
 // process data
 send(client, buffer, bytes, 0);
}
closesocket(client);
```

# Flow of TCP Server and Client

## TCP Server

1. WSAStartup()
2. socket()
3. bind()
4. listen()
5. Loop:
    - accept()
    - recv()/send()
    - closesocket(client)
6. closesocket(listening)
7. WSACleanup()

## TCP Client

1. WSAStartup()
2. socket()
3. connect()
4. send()/recv()
5. closesocket()
6. WSACleanup()

# Functions for Handling Blocking I/O

# Functions for Handling Blocking I/O

- Winsock provides several mechanisms to control and react to blocking behavior on sockets.
- These let us multiplex many sockets or integrate with Windows' event and message systems.

- **ioctlsocket()** : Set Blocking/Nonblocking Mode

```
int ioctlsocket(SOCKET s,
 long cmd, // e.g. FIONBIO
 u_long *argp // 0 = blocking, nonzero = nonblocking
);s
```

- **FIONBIO**: turn nonblocking mode on/off.

/* **Returns** 0 on success, SOCKET_ERROR on failure (WSAGetLastError() gives WSAEINVAL, WSAENOTSOCK, etc.). */

- **Example**:

```
u_long mode = 1; // nonblocking
ioctlsocket(sock, FIONBIO, &mode);
```

# Functions for Handling Blocking I/O

- **select()** – I/O Multiplexing

```
int select(
 int nfds, // ignored by Winsock; set FD_SETSIZE before compile
 fd_set *readfds, // sockets to check for readability
 fd_set *writefds,// sockets to check for writability
 fd_set *exceptfds,// sockets to check for errors/out-of-band data
 const struct timeval *timeout // NULL = block indefinitely
);
```

- **Monitors** multiple sockets at once for readiness.

/***Returns** number of sockets ready, 0 on timeout, SOCKET_ERROR on failure.*/

**Notes**:

- Only sockets (not pipes/files).
- Default FD_SETSIZE=64; raise by #define FD_SETSIZE before including headers.

# Functions for Handling Blocking I/O

- **WSAAsyncSelect()** – Message-Driven Notification
- *WSAAsyncSelect() is used* 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.

```
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.

# Functions for Handling Blocking I/O

- *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 we are interested. To cancel all event notifications, call *WSAAsyncSelect()* with wMsg and lEvent set to 0.

# Functions for Handling Blocking I/O

- **WSAEventSelect()** – Event-Handle Notification

```
int WSAEventSelect(
 SOCKET s,
 WSAEVENT hEventObject, // Created by WSACreateEvent()
 long lNetworkEvents // Same bitmask as WSAAsyncSelect
);
```

- Associates a Winsock event object with a socket.
- When any requested network event occurs, the event object is signaled.

/* **Returns** 0 on success, SOCKET_ERROR on failure.*/

**Wait** on the event with:

```
WSAWaitForMultipleEvents(
 DWORD cEvents,
 const WSAEVENT *lphEvents,
 BOOL fWaitAll,
 DWORD dwTimeout,
 BOOL fAlertable
);
```

# Asynchronous Database Functions

# WSAAsyncGetHostByName()

- The function of gethostbyname() is to take a host name and return its IP address.

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

- Its asynchronous counterpart function is *WSAAsyncGetHostByName().*

```
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.
- **wMsg** is the message that will be posted to **hWnd**
- **name** is a pointer to a string that contains the host name
- **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 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.

# WSACancelAsyncRequest()

- *WSACancelAsyncRequest()* can be used to cancel the asynchronous winsock calls.

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

- **hAsyncTaskHandle:** The asynchronous task handle returned by one of the *WSAAsyncGetXByY* functions (such as *WSAAsyncGetHostByName*, *WSAAsyncGetHostByAddr*, *WSAAsyncGetServByName*, etc.).
- On success, this function returns 0 (zero).
- On failure, it returns SOCKET_ERROR, and **WSAGetLastError**() can be called.

# WSAAsyncGetHostByAddr()

- The function of *gethostbyaddr()* is to take the IP address of a host and return its name.
- *struct hostent *PASCAL gethostbyaddr(const char * addr, int len, int type);*
- Its asynchronous counterpart function is *WSAAsyncGetHostByAddr()*

```
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.
- **wMsg** is the user defined message that will be posted to **hWnd**
- **addr** is a pointer to the IP address
- **len** is the length of the address to which **addr** points
- **buf** is a pointer to an area of memory that contains the hostent structure for the desired host.
- **buflen** is the size of the buf buffer.
- On Success returns 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.

# WSAAsyncGetServByName()

- The **getservbyname**() function gets service information corresponding to a pecific service name and protocol.

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

- **WSAAsyncGetServByName**() is its asynchronous counterpart to getservbyname().

```
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.
- **wMsg** is the user defined message that will be posted to hWnd
- **name** is a pointer to a service name about which you want service information.
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory
- **buflen** is the size of the buf buffer.
- It should be MAXGETHOSTSTRUCT for safety's sake.

# WSAAsyncGetServByPort()

- The **getservbyport**() function gets service information corresponding to a specific port and protocol.

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

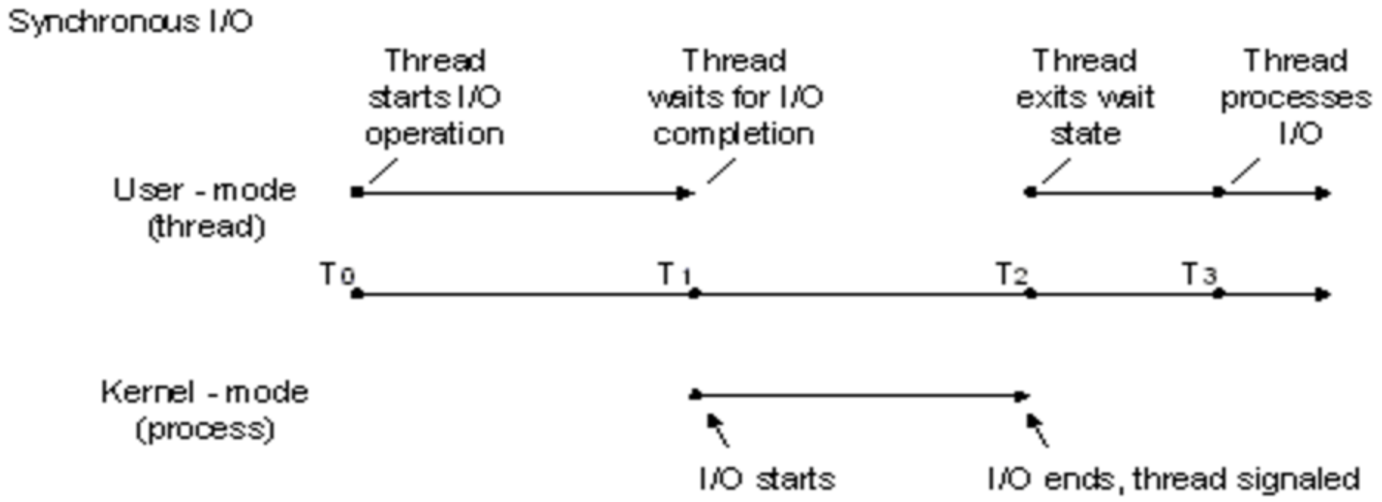
- *WSAAsyncGetServByPort()* is the asynchronous counterpart to getservbyport().

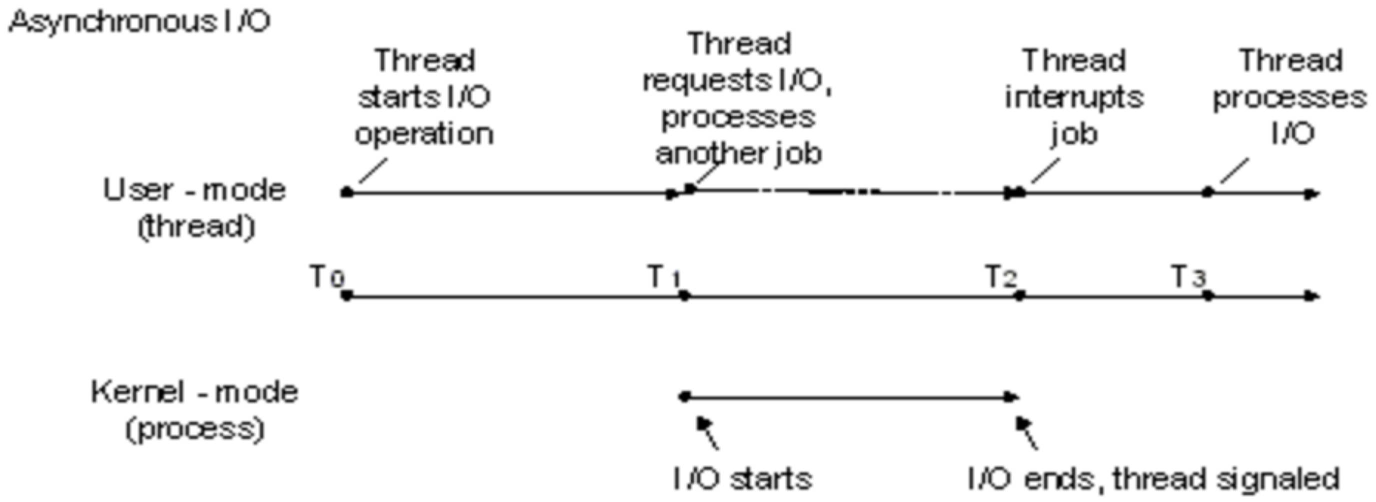
```
HANDLE PASCAL FAR WSAAsyncGetServByPort(HWND hWnd,u_int wMsg,int port,
const char FAR* proto,char FAR *buf,int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent.
- **wMsg** is the user defined message that will be posted to
- **port** is the service port in network byte order
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory
- **buflen** is the size of the buf buffer
- If the asynchronous operation is initiated successfully, the return value of WSAAsyncGetServByName() is a handle to the asynchronous task. On failure, the function returns 0 (zero).

# Asynchronous IO Functions

# Synchronous and asynchronous IO



# Asynchronous IO Functions

- I/O Modes: There are two I/O synchronization types: *synchronous I/O*, where a thread blocks until an operation completes, and *asynchronous (overlapped) I/O*, where a thread issues a request and continues working until notified of completion.
- *Asynchronous Workflow:* In overlapped I/O, a thread calls functions like *WSASend/WSARecv*; if the kernel accepts the request, the thread proceeds with other tasks until the kernel signals that the I/O is done.
- *Use Cases:* Asynchronous I/O boosts efficiency for long-running operations (e.g., large database backups or slow links) by avoiding idle waits.
- *Trade-Offs:* For short, fast I/O operations, the extra kernel signaling overhead may outweigh benefits, making synchronous I/O preferable.
- *Winsock Support*: Winsock APIs (*WSASend*,*WSARecv*,*WSASendTo*, *WSARecvFrom*,*WSAIoctl*,etc.) support both modes and *WSASocket*(..., WSA_FLAG_OVERLAPPED) is used to create sockets for overlapped operations.

# WSASocket()

- Create **overlapped** sockets necessary for high-performance asynchronous I/O
- Select a specific **service provider** via lpProtocolInfo.

```
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]** : IPPROTO_TCP, IPPROTO_UDP, IPROTO_ICMP or 0
- **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.
- **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 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.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpOverlapped[in]**: A pointer to a WSAOVERLAPPED structure
- **lpCompletionRoutine[in]**:A pointer to the completion routine called when the send operation has been completed.
- **lpNumberofBytesSent[out]**:The pointer to the number.
- **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
- **dwBufferCount[in]** :The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** :A pointer to the number data received in bytes.
- **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.
- **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
- **dwBufferCount [in]** :The number of WSABUF structures in the lpBuffers array.
- **lpNumberofBytesRecvd[out]** : A pointer to the number of bytes received by this call
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of this function.
- **lpFrom [out]** : An optional pointer to a buffer that will hold the source address
- **lpFromlen [in, out]** : A pointer to a WSAOVERLAPPED structure
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the WSARecvFrom operation has been completed

# WSAIoctl()

- Performs network I/O control operations on a socket—setting modes, querying status, or retrieving extension function pointers for overlapped I/O helpers.

```
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
- **lpCompletionRoutine [in]** :A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets)

# Getting Started with WinSock

# Setting up Environment

1. **Install a C compiler**
    1. Visual Studio (Community Edition) or
    2. MinGW-w64 with GCC
2. **Ensure Windows SDK is available**
    1. Contains headers (winsock2.h, ws2tcpip.h) and import library (Ws2_32.lib).
3. **Create a project**
    1. In Visual Studio: New → Win32 Console Application → blank project.
    2. In MinGW/GCC: Create a .c file and compile via CLI.

# 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.

# Include Headers & Link Libraries

```
// Before any <windows.h> include:
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Link against Ws2_32.lib
// - Visual Studio: add Ws2_32.lib in Linker → Input → Additional Dependencies
// - MinGW/GCC: gcc myapp.c -lws2_32 -o myapp.exe
```

# Include Headers & Link Libraries

```
// Before any <windows.h> include:
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Link against Ws2_32.lib
// - Visual Studio: add Ws2_32.lib in Linker → Input → Additional Dependencies
// - MinGW/GCC: gcc myapp.c -lws2_32 -o myapp.exe
```

# Initializing 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;
}
```

- **MAKEWORD(2,2)** requests Winsock version 2.2.
- On success, wsaData holds implementation details.

# Creating a Socket for the Client

## 3. Create a socket

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == INVALID_SOCKET) {
    printf("socket() failed: %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
```

- **AF_INET** = IPv4, **SOCK_STREAM** = TCP.
- INVALID_SOCKET indicates failure.

**htons()** converts port to network byte order.
**inet_pton()** converts dotted -decimal string to binary.

## 4. Prepare Server Address

```
struct sockaddr_in serverAddr;
ZeroMemory(&serverAddr, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port   = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &serverAddr.sin_addr);
```

# 5. Connect (Client) / Bind & Listen (Server)

## Client: connect()

```
if (connect(sock, (struct sockaddr*)&serverAddr, sizeof(serverAddr))==SOCKET_ERROR)
{
    printf("connect() failed: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return 1;
}
printf("Connected to server!\n");
```

## Server: bind() + listen() + accept()

```
bind(sock, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
listen(sock, SOMAXCONN);

SOCKET client = accept(sock, NULL, NULL);
if (client != INVALID_SOCKET) {
    printf("Client connected!\n");
    // use 'client' for send/recv
}
```

# 6. Send and Receive Data

```
//send
char sendBuf[] = "Hello, Winsock!";
int sent = send(sock, sendBuf, (int)strlen(sendBuf), 0);
if (sent == SOCKET_ERROR) {
    printf("send() failed: %d\n", WSAGetLastError());
}

// Receive
char recvBuf[512];
int received = recv(sock, recvBuf, sizeof(recvBuf), 0);
if (received > 0) {
    recvBuf[received] = '\0';
    printf("Received: %s\n", recvBuf);
}
```

# 7. Clean Up

```
closesocket(sock);
WSACleanup();
```

- Always close sockets before calling WSACleanup().
- Match each WSAStartup() with one WSACleanup().

# Shutdown the connection

The shutdown function is used to shutdown for sending and receiving .

```
shutdown(socketfd,SD_SEND); //shutdown for sending
shutdown(socketfd,SD_RECV);//shutdown for receiving
shutdown(socketfd,SD_BOTH);//shutdown for both
```

**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.

# shutdown() vs closesocket() (or Unix's close())

| Aspect | shutdown() | closesocket() / close() |
|---|---|---|
| **Purpose** | Gracefully disable sends and/or receives on a socket | Release the socket handle and free all associated resources |
| **Prototype** | int shutdown(SOCKET s, int how); | int closesocket(SOCKET s); (Win)<br>int close(int fd); (Unix) |
| **how / howflags** | SD_SEND (no more sends) | N/A |
| | SD_RECEIVE (no more receives) | |
| | SD_BOTH (both directions) | |
| **Behavior** | Sends a FIN to peer (for SD_SEND/SD_BOTH) but socket remains open for the other direction until closed. | Immediately tears down connection, discards unsent data, and invalidates the socket handle. |
| **Use Cases** | Half-close a connection (e.g., signal "I'm done sending" while still reading).– Control TCP FIN/ACK sequencing. | Final cleanup when you're completely done with the socket.– Release handle back to the OS. |
| **Return Value** | 0 on success; SOCKET_ERROR on failure (check WSAGetLastError()) | Same as above |
| **After Calling** | Socket still valid; you must call closesocket() (or close()) to free it. | Handle is invalid; no further Winsock (or POSIX) calls can use it. |

# Full Minimal Client Program

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
int main() {
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2,2), &wsaData) != 0)
        return 1;
    SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == INVALID_SOCKET) { WSACleanup(); return 1; }
    struct sockaddr_in srv = {0};
    srv.sin_family = AF_INET;
    srv.sin_port   = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &srv.sin_addr);

    if (connect(sock, (struct sockaddr*)&srv, sizeof(srv)) == SOCKET_ERROR) {
        printf("connect failed: %d\n", WSAGetLastError());
    } else {
        printf("Connected!\n");
    }
    closesocket(sock);
    WSACleanup();
    return 0;
}
```

# Full Minimal Echo Server Program

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
// Link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define LISTEN_PORT "8080"
#define BUFFER_SIZE 512

int main(void) {
    WSADATA wsaData;
    SOCKET listenSock = INVALID_SOCKET, clientSock = INVALID_SOCKET;
    struct addrinfo hints = {0}, *res = NULL;
    struct sockaddr_storage clientAddr;
    int addrLen = sizeof(clientAddr);
    char buffer[BUFFER_SIZE];
    int bytes;

    // 1. Initialize Winsock
    if (WSAStartup(MAKEWORD(2,2), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed\n");
        return 1;
    }
```

# Minimal Echo Server Program…

```
// 2. Resolve local address and service (port)
hints.ai_family   = AF_UNSPEC;        // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;      // TCP
hints.ai_flags    = AI_PASSIVE;       // For bind
if (getaddrinfo(NULL, LISTEN_PORT, &hints, &res) != 0) {
    fprintf(stderr, "getaddrinfo failed\n");
    WSACleanup();
    return 1;
}
// 3. Create listening socket
listenSock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (listenSock == INVALID_SOCKET) {
    fprintf(stderr, "socket() failed: %d\n", WSAGetLastError());
    freeaddrinfo(res);
    WSACleanup();
    return 1;
}
// 4. Bind to the port
if (bind(listenSock, res->ai_addr, (int)res->ai_addrlen) == SOCKET_ERROR) {
    fprintf(stderr, "bind() failed: %d\n", WSAGetLastError());
    closesocket(listenSock);
    freeaddrinfo(res);
    WSACleanup();
    return 1;
}
freeaddrinfo(res);
```

# Minimal Echo Server Program…

```
// 5. Listen for incoming connections
if (listen(listenSock, SOMAXCONN) == SOCKET_ERROR) {
    fprintf(stderr, "listen() failed: %d\n", WSAGetLastError());
    closesocket(listenSock);
    WSACleanup();
    return 1;
}
printf("Server listening on port %s...\n", LISTEN_PORT);

// 6. Accept one client
clientSock = accept(listenSock, (struct sockaddr*)&clientAddr, &addrLen);
if (clientSock == INVALID_SOCKET) {
    fprintf(stderr, "accept() failed: %d\n", WSAGetLastError());
    closesocket(listenSock);
    WSACleanup();
    return 1;
}
printf("Client connected.\n");

// 7. Echo loop
while ((bytes = recv(clientSock, buffer, BUFFER_SIZE, 0)) > 0) {
    send(clientSock, buffer, bytes, 0);
}
```

# Minimal Echo Server Program…

```
    // 8. Cleanup
    closesocket(clientSock);
    closesocket(listenSock);
    WSACleanup();
    printf("Server shut down.\n");
    return 0;
}
```

# End of Chapter 4