Ch3 winsock programming

Winsock (windows sockets) is a network API provided by microsoft that enable developers to create network-based applications. It serves as an abstraction layer over various network protocol, allowing applications to communicate over the internet or local networks without needing to manage low level networking details.

some of the key features of winsock:

① socket-based communication
Enables applications to establish connections using TCP/IP or UDP

② client/server architecture: supports the development of distributed applications like: chat app, web browser etc using client/server architecture.

③ Protocol Independence: winsock 2.0 extends beyond TCP/IP to support other protocols like:- AppleTalk, & NetWare

④ Name resolution service: converts domain names into IP addresses.

The winsock programming basic steps are:

① Initialize winsock: use WSAStartup() to initialize winsock

② Create a socket: use socket() to create a socket object.

③ Bind the socket (server-side): use bind() to associate the socket with a local address & port.

④ Listen for connection (server-side)

↳ use listen() and accept() method to listen (wait for incoming connection request) & accept the client's connection request.

⑤ Connect to server (client-side)

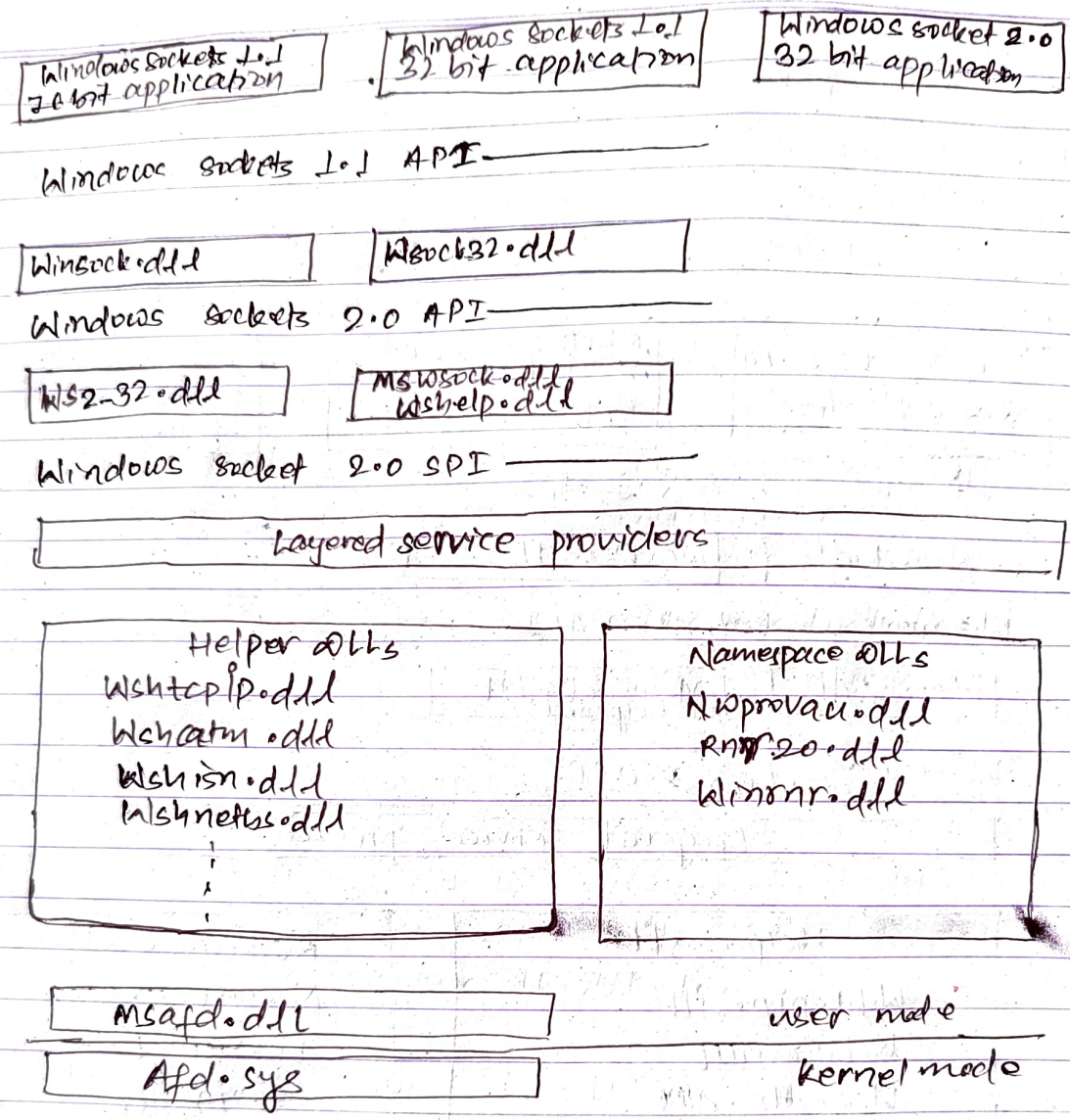
↳ Client will use connect() method to establish a connection to the server.

⑥ Send & Receive data : use send() & recv() for sending & receiving data.

⑦ Close the socket : use closesocket() function to close the socket & terminate connection.

Intro to WinSock Architecture

↳ Winsock 2.0 consists of multiple layers that manage network communication between applications and underlying n/w stack. Below diagram represents the Winsock 2.0 architecture & its components:



① Application layer

It is the top layer of Winsock Architecture 2.0 where it contains Windows sockets applications (16 bit & 32 bit) using Winsock 1.1 API & Winsock 2.0 API.

② Winsock APIs

ⓐ Winsock 1.1 API : uses Winsock.dll (16 bit) and Wsock32.dll (32-bit) for legacy support.

ⓑ Winsock 2.0 API : It uses WS2_32.dll, MSWsock.dll and Wshelp.dll for enhanced functionality.

नेपाल सरकार
शिक्षा मन्त्रालय
परीक्षा नियन्त्रण कार्यालय, सानोठिमी
एसएलसी थप उत्तरपुस्तिका

विषय..... मिति .....
मूल उत्तरपुस्तिका नं .....
निरीक्षकको सही: .....

(3) Winsock 2.0 SPI (service provider Interface)
↳ Allows 3rd party extensions to add extra networking capabilities

(4) Layered service providers
→ Extends winsock functionality eg:- encryption, logging, compression.

(5) Helper & Namespace DLLs (Dynamic link library)
↳ Helper DLLs provide support for different protocols eg:- TCP/IP, atm, NetBios etc.

Namespace DLLs handles name resolution (converting domain names to IP addresses).

(6) User and kernel mode
User mode contains component like: Msafd.dll which acts as an interface between user-mode application and transport layer.

Kernel mode contains component like: Afd.sys which acts as TDI component that interact with windows n/w stack.

(3.2) Winsock DLL (Dynamic Link Library)

Winsock DLLs or DLLs are the files containing reusable functions, procedures and resources that multiple applications can use at runtime. Winsock ~~DLL~~ DLLs enable network communication by acting as an interface between user applications and windows networking stack.

Why winsock DLLs are used?

(a) Code reusability : Multiple applications can use same DLL function, procedures or classes without embedding them into their executables.

(b) Memory efficiency : Since DLLs are loaded only when needed, they help in efficient memory usage and conserve system resources.

(c) Modular Architecture : It uses modular architecture where huge task is divided sub-task. foreg handling network connection, we can use net.dll and for data base connections, we can use db.dll etc.

(d) Dynamic linking : Winsock DLLs are linked to applications at runtime, allowing ~~application~~ flexible & efficient n/w operations.

Cons of DLL

(a) version conflict
(b) Missing DLLs

Types of DLL in windows

@ Static DLLs

→ It is the process of including all of the code from a library directly into the application executable file at compile time and known to be ~~dynamic~~ static library.

→ Extension of static DLL is .lib. Some of the characteristics of static DLL is:

ⓐ Compile time binding directly.
All the code from library is included into application's executable file at compile time

ⓑ No runtime dependencies: There is no runtime dependencies in the static DLL since the code from library is directed added to apps executible file at compile time.

ⓒ Application bundled with code.

Drawbacks

@ Large size of Executable
Since all of the code is directly included in application's executable file, so the size of executable will be large.

(b) Not synchronized with updates
If library is updated, the application must be recompiled.

(c) No sharing : Here sharing of library is not possible as each program will get copy of library separately.

(5) Dynamic DLLs
Another type of DLLs where code is loaded and linked from the library at runtime rather than compile time.
It is loaded into the memory when application needs it.
The extension of dynamic dlls is .dll

Some of the characteristics
(1) Runtime binding : Application loads DLL at runtime (explicitly or implicitly) and bind to function it needs.

(2) Space efficient : only required DLL being loaded at runtime, it ensure efficient use of system memory.

(3) Depend on runtime dependencies
↳ Application dependent on DLL at runtime. If the DLL is not provided, application crashes or fails to load.

(4) Shared libraries : only instance of libraries are loaded into memory and can be shared across multiple apps thereby saving memory space.

(5) Run time loading

Application can decide when to load the DLLs and where they want using functions like LoadLibrary() and GetProcAddress()

pros & cons of dynamic DLLs

pros	cons
(a) Smaller executable	(a) Dependent on runtime dependency
(b) memory efficient	(b) Slight overhead
(c) code reusability	(c) DLL hell problem: If different apps uses different version of same DLL, it leads to conflict called as DLL hell.
(d) Easier update: If we update any dll, only dll need to be recompiled.	
(e) Runtime loading	

Winsock ಅಂದರೇನು?

(3.3) Windows sockets and Blocking I/O

Blocking I/O in winsock means that a function doesn't return control to the program until it has completed its operation. It means:

For reading (recv()) : The function waits until data arrives before returning. If no data is available, the function blocks execution infinitely.

For writing (send()) : The function waits until all of the data has been written successfully to kernel's network buffer before returning.

A simple example to demonstrate Blocking I/O in winsock (not a full fledged code but contains core logic) is:

```
#include<winsock2.h>

WSADATA wsadata;
SOCKET confd, fd;
char recvmsg[100];

//Initialize winsock
 WSAStartup(MAKEWORD(2,2), &wsadata);
 //create socket
  fd= socket( AF_INET, SOCK_STREAM, 0);
 // bind socket

 struct sockaddr_in server; // severstructure
  server.sin_family= AF_INET;
 server.sin_port= htons(8080);
 server.in_addr.sin_addr= htonl(INADDR_ANY);
```

```
// Bind socket:
bind(fd, (struct sockaddr*)&server, sizeof(server));

// listen for incoming connection
listen(fd, 5);
// Accept connection (Blocking)
confd = accept(fd, NULL, NULL);

while(TRUE)
{  // Blocking read
   int r = recv(confd, recvmsg, sizeof(recvmsg), 0);
   // Blocking write
   int s = send(confd, "OK", 2, 0);

}

// cleanup
closesocket(confd);
closesocket(fd);
WSACleanup();
```

Here is how above code demonstrates blocking I/O.

(a) accept() is blocking : It waits until a client connects and when the client connects, then only it returns.

(b) recv() is also blocking : It waits until data arrives before proceeding.

(c) send() is also blocking : It waits until all of the data is written successfully to kernel's n/w buffers and then only returns.

(3.4) Winsock extension : setup & cleanup function

→ Window socket extension refers to the enhancement made to the winsock API to provide additional features and enhancement to introduce advanced networking features, improved performance and support for modern n/w protocols.

Why windows extended winsock

(a) To introduce advance networking features
(b) To support modern network protocols like IPV6
(c) To improve performance of n/w based apps.

Key features introduced by winsock API extension

(a) Async I/O

↳ Introduced in winsock 2 to allow non blocking socket communication. functions used to enable asynchronous mode:

(i) WSAAsyncSelect()
(ii) WSAEventSelect()
(iii) WSAOverlapped()

(b) multicast n/w support

↳ Allow sending data to multiple ~~receipents~~ recipients in a network. uses multicast groups and Internet group mgmt protocol (Igmp)

(c) IPV6 support

↳ provides compatibility with modern IPV6 n/w. introduces new address structures and API functions for IPV6.

(d) socket filtering : Allows filtering n/w traffic based on protocol, source or destination. Can intercept and process packets before sending them to apps.

(e) DLL support for winsock : Allows to load the library dynamically at runtime; leads to ~~efficient~~ minimal resource consumption

(f) Socket security (SSL/TLS)

↳ supports secure socket layer and transport layer security (TLS) (SSL)

↳ Provides encryption & authentication for secure communication.

setup and clean up function {3.4 ko part ho}

In Winsock programming, setup and cleanup function are the most important function. In winsock programming before using any networking function, we need to initialize winsock and for that we need to use setup function.

Similarly, once we are done with networking, we need to cleanup the winsock resources. Hence to cleanup winsock resources, we use cleanup function.

(a) setup function (WSAStartup())

↳ Initialize the Winsock API before using any socket functions. The setup method needs to be called before any socket function like ~~create~~, socket() bind(), listen(), recv(), send(), accept().

The syntax is:

WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);

Here parameters:

wVersionRequested: specifies the Winsock version required like MakeWord(2,2) is for version 2.2

lpWSAData: indicates a pointer WSADATA structure which contains information about winsock implementation on the system.

This function returns 0 on success and SOCKET_ERROR on failure.

⑤ Cleanup function

↳ Cleanup function i.e. WSACleanup() is invoked after the application has finished using Winsock functions.

Once we are done with networking operation, we invoke WSACleanup() to cleanup winsock resources. It doesn't require any parameters and returns 0 on success and SOCKET_ERROR on failure.

(3.5) Functions for handling blocked I/O

Blocked I/O is a operation where function doesn't return control to program unless its ongoing operation is completed. This can cause a program to freeze while waiting for a data or connection.

Winsock provides several functions for handling blocked I/O :

ⓐ WSAIsBlocking()
ⓑ WSACancelBlockingCall()
ⓒ WSASetBlockingHook()
ⓓ WSAUnhookBlockingHook()

WSAIsBlocking() checks if the current winsock call is blocking or not. Based on the return value, we can know whether it is blocking or non blocking:

ⓐ If return value is non zero, blocking call is in progress
ⓑ If return value is zero, no blocking call in progress.

It is depricated in winsock 2.

C prototype:

```
// Initialize winsock
  WSAStartup(.....);

  if(WSAIsBlocking)
  {
    printf("Blocking call is in progress);
  }
  else
  {
    printf("No blocking call in progress");
  }
```

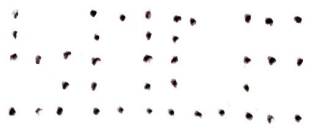
WSACancelBlockingCall() cancels blocking call if it is taking too long. Yes we can use WSACancelBlockingCall() whenever we want but there's an important catch:

→ Normally, a method like recv(), or send() is blocking, so it won't return until the operation is completed. But if the operation is taking too long to complete (eg:- n/w is slow, server is unresponsive), program gets stuck. So WSACancelBlockingCall() allows you to forcefully interrupt this stuck operation and forcefully return control to your program.

Return value will be 0 on successful canceling the blocking call else it will return SOCKET_ERROR on failure. It is also deprecated on winsock2.

C prototype

```
WSAStartup(- - - );
sock=socket(- - - - - );

// filling socket address struct
struct sockaddr_in server;
server.sin_family= AF_INET
server.sin_port= htons(8080);
        :
        :
result= connect(- - -);
result = recv(- - - );

// cancelling blocking call
flag= WSACancelBlockingCall();
if (flag == SOCKET_ERROR)
{ printf("Blocking call is not cancelled");
```



```
}
else
{
  printf("Blocking call cancelled successfully");
}

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

WSASetBlockingHook() is used to set a custom function that executes when a socket is blocked.
Return value will be:
- (a) pointer to previous blocking function: in success (case)
- (b) Null: in case of failure.

It is also depricated in winsock2

C prototype:

```
BOOL FAR PASCAL myBlockingHookFunction()
{
   printf("custom Blocking Hook is ready\n");
   sleep(1500); // simulating some work
   return FALSE // returning false to allow blocking call to
                   continue
}

int main()
{
  WSAData wsadata;
  FARPROC prevHook;
  WSAStartup(....);
```

```
//setting a custom Blocking Hook

prevHook = WSASetBlockingHook((LPARPROC)myBlockingHookFunction);

if(prevHook==NULL)
{
   printf("Error setting custom Hook\n");
}

sock= socket(....);
 //setup address
struct sockaddr_in server;
server.sin_family = AF_INET;
        .
        .
        .
connect(---);

// Restore original blocking hook
WSASetBlockingHook(prevHook);

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

save

WSAUnhookBlockingHook() is used to remove the custom blocking hook function. It returns 0 in case of success and SOCKET_ERROR in case of failure.

```
if(WSAUnhookBlockingHook()==0)
   printf("custom hook removed successfully");
else
   printf("Error removing custom hook");
```

(B.6) Asynchronous database function

Asynchronous database function are the non blocking network queries, whose results are sent to windows message handler instead of blocking execution.

ⓐ WSAAsyncGetServByName().
ⓑ WSAAsyncGetServByPort().
ⓒ WSAAsyncGetP ByName();
ⓓ WSAAsyncGetP ByNumber();
ⓔ WSAAsyncGetHostByName();
ⓕ WSAAsyncGetHostByAddr();

ⓐ WSAAsyncGetServByName()

↳ It retrieves service details (port, protocol) by service name (eg: "http").

Syntax

```
UINT WSAAsyncGetServByName(

HWND hWnd; // handle to window receiving message
u_int wMsg; // message to be posted, when operation completes
const char *lpServiceName; // service name eg: "http"
const char * lpProtocolName, // protocol name eg: "tcp"
struct servent *lpServEnt,
int nBufLen
);
```

eg:

```
struct servent serv;
UINT res = WSAAsyncGetServByName(hWnd, uMsg, "http",
    "tcp", serv, sizeof(serv));
```

```
#Include<iostream.h>

WSADATA wsadata;
char recvmsg[10];
SOCKET confd, fd;

// Initializing winsock
WSAStartup(MakeWord(2,2), &wsadata);

// creating socket
fd=socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in server;
server.sin_family= AF_INET;
server.sin_port = htons(8080);
server.in_addr.sin_addr= htonl(INADDR_ANY);

// Binding to a socket
bind(fd, (struct sockaddr*)&server, sizeof(server));

// listening to @ incoming connecting
confd listen(fd, 5);

// accepting incomming conn req
confd= accept(fd, NULL, NULL);

While (TRUE)
{
    int r = recv(confd, recvmsg, sizeof(recvmsg), 0
    int s= send(confd, "OK", 2, 0);
}
```

नेपाल सरकार
शिक्षा मन्त्रालय
परीक्षा नियन्त्रण कार्यालय, सानोठिमी
एसएलसी थप उत्तरपुस्तिका

विषय..... मिति .....
मूल उत्तरपुस्तिका नं .....
निरीक्षकको सही: .....

थप उत्तरपुस्तिकाको दुवैतर्फ यसै पेजदेखि लेख्नुहोला । यसमा सि.नं. तथा अन्य कुनै सङ्केत लेखेको पाइएमा परीक्षा रद्द हुनेछ ।

(b) Async
WSAGetServByPort()

i) It receives service information by port number & protocol.

Eg:

```
UINT WSAGetServByName(
    HWND hWnd,
    u_int uMsg,
    int port,
    const char *proto
    struct servent *lpServEnt;
    int nBuffLen
);
```

Eg:-

```
struct servent serv;
.
.
UINT res = WSAGetServByName(
        hWnd, uMsg, htons(8080), &serv,
        sizeof(serv));
```

(c) Async
WSAGetProtoByName()

This method is used to retrieve protocol details by name

Syntax:

```
UINT WSAAsyncGetProtoByName(
    HWND hWnd,
    u_int wMsg,
    const *char *name,
    struct servent ser
    struct protoent *lpProtoEnt, int nBuflen);
```

eg:-

```
struct protoent proto;
UINT res= WSAAsyncGetProtoByName (
            hWnd,
            wMsg,
            "tcp",
            &proto,
            sizeof (proto));
```

④ WSAAsyncGetProtoByNumber()

↳ It gives protocol details on the basis of protocol numbers (eg:- 6 → "tcp")

Syntax

```
UINT WSAAsyncGetProtoByNumber
(
  HWND hWnd,
  u_int wMsg,
  int protocolNumber,
  struct protoent *lpProtoEnt,
  int nBuffLen );
```

eg:-

```
struct protoent proto;

UINT asyncResult= WSAAsyncGetProtoByNumber(
                    hWnd,
                    wMsg
                    IPPROTO_TCP,
                    &proto,
                    sizeof(proto));
```

नेपाल सरकार
शिक्षा मन्त्रालय
परीक्षा नियन्त्रण कार्यालय, सानोठिमी
एसएलसी थप उत्तरपुस्तिका

विषय..... मिति .....
मूल उत्तरपुस्तिका नं .....
निरीक्षकको सही: .....

थप उत्तरपुस्तिकाको दुवैतर्फ यसै पेजदेखि लेख्नुहोला । यसमा सि.नं. तथा अन्य कुनै सङ्केत लेखेको पाइएमा परीक्षा रद्द हुनेछ ।

⑤ WSAAsyncGetHostByName()

↳ ~~It provides the host information~~

↳ It resolves the hostname to an IP address asynchronously.

- Syntax

```
UINT WSAAsyncGetHostByName(
    HWND hWnd,
    u_int wMsg,
    const char *name,
    struct hostent *lpHostEnt,
    int nBuflen
);
```

Eg:-

```
struct hostent host;
UINT asyncRes= WSAAsyncGetHostByName(
        hWnd,
        wMsg,
        "www.google.com",
        &host,
        sizeof(host));
```

(3.8) Error handling functions; Async ops

In windows socket programming, error handling is crucial for diagnosing and managing issues related to socket operation. The Winsock API provides functions like:

WSASetLastError()
WSAGetLastError()

to handle error efficiently.

@ WSAGetLastError()

↳ This function retrieves the last error code that was set by a Winsock function. It is used when you need to check the error condition after winsock function call has failed.

Syntax: int WSAGetLastError();

It doesn't need any input params and returns the error code of (type int) last winsock function whose call has failed. The common error codes are:

WSAEINPROGRESS : A blocking op is currently in progress
WSAEINVAL : An invalid args was passed
WSAENOTSOCK : The descriptor is not a socket
WSAENOTCONN : The socket is not connected
WSAETIMEOUT : A connection attempt timeout.

Eg:-
```
if((WSAStartup(MAKEWORD(2,2), &wsadata) != 0){
    printf("Error in initialising winsock API
        :%d\n", WSAGetLastError());
}
```

(b) WSASetLastError()

b) This method allows you to explicitly set the last error code for a winsock function. It is used when we need to simulate or manually set an error condition in winsock API.

Syntax: void WSASetLastError (int iError);

Here params: iError is the error codes that you wanna set as the lastError. It must be one of the error code that we mentioned above (eg: WSAEINVAL, or WSAENOTCONN, etc).

Also, it doesn't return anything

eg: WSASetLastError(WSAEINVAL); //sets and error

int error = WSAGetLastError();
printf("Last error code: [%d]\n", error);

नेपाल सरकार
शिक्षा मन्त्रालय
परीक्षा नियन्त्रण कार्यालय, सानोठिमी
एसएलसी थप उत्तरपुस्तिका

विषय..... मिति .....
मूल उत्तरपुस्तिका नं .....
निरीक्षकको सही: .....

थप उत्तरपुस्तिकाको दुवैतर्फ यसै पेजदेखि लेख्नुहोला । यसमा सि.नं. तथा अन्य कुनै सङ्केत लेखेको पाइएमा परीक्षा रद्द हुनेछ ।

(6) WSASetLastError()

b) This method allows you to explicitly set the last error code for a winsock function. It is used when we need to simulate or manually set an error condition in winsock API.

Syntax: void WSASetLastError(int iError);

Here params: iError is the error codes that you wanna set as the lastError. It must be one of the error code that we mentioned above (eg: WSAEINVAL, or WSAENOTCONN, etc).

Also, it doesn't return anything

eg: WSASetLastError(WSAEINVAL); //sets an invalid args error

int error = WSAGetLastError();
printf("Last error code: [%d]\n", error);

Ch3 @Bank

24 & 25 Explain WSASend(), WSASendto(), WSARecv(), WSARecvfrom() in detail. over socket efficiency

Sol^n In winsock programming sending and receiving data is crucial. The method WSASend(), WSASendto(), WSARecv(), & WSARecvfrom() are advanced socket function for windows OS, which allows asynchronous and overlapped I/O operation. These functions are commonly used in winsock programming to handle transmission and receiving of data.

WSASend()

↳ This function is used to send data on a connected socket. This function supports both synchronous and overlapped (async) I/O operation.

The return value of this function is 0 if the sending operation is successful but if the operation fails, the return value will be SOCKET_ERROR i.e. -1 and we can use WSAGetLastError() to get the error code.

```
eg: WSABUF wsaBuf;
    char data[] = "Hello world!";

    wsaBuf.buf = data;
    wsaBuf.len = strlen(data);

    DWORD bytesSent = 0;
    if(WSASend(sock, &wsaBuf, 1, &bytesSent, 0, NULL,
                NULL) == SOCKET_ERROR)
    {
        printf("WSASend failed with error: %d\n",
                WSAGetLastError());
    }
```

WSASendto()

1) This function is used to send data to a specific address when using connectionless (UDP) sockets. Return value of the function will be 0 if successful and SOCKET_ERROR in case of failure and we can use WSAGetLastError() to get the error.

eg:
```
WSABUF wsaBuf;
char data[] = "Hello world!";

wsaBuf.buf = data;
wsaBuf.len = strlen(data);

DWORD bytesSent = 0;

struct sockaddr_in destAddr;
destAddr.sin_family = AF_INET;
destAddr.sin_port = htons(8080);
destAddr.sin_addr.s_addr = inet_addr("192.168.1.10");

if (WSASendto(sock, &wsaBuf, 1, &bytesSent, 0, (sockaddr*)&destAddr,
    sizeof(destAddr), NULL, NULL) == SOCKET_ERROR)
{
    printf("WSASendto failed with error: %d\n",
           WSAGetLastError());
}
```

WSARecv()

It is used to receive data from a connected socket. The return value of this function will be 0 in success case and SOCKET_ERROR in case of failure. Also, we can use WSAGetLastError() to get the error code & error details.

Eg:

```
// other codes
.
.
{
WSABUF wsaBuf;
Char recvMsg[512];
wsaBuf.buf = recvMsg;
wsaBuf.len = sizeof(recvMsg);
DWORD bytesReceived = 0;
DWORD flags = 0;

if (WSARecv(sock, &wsaBuf, 1, &bytesReceived, &flags,
    NULL, NULL) == SOCKET_ERROR)
  {
    printf(" WSARecv failed due to error:
          %d \n", WSAGetLastError());
  }
else
  {
    recvBuffer[bytesReceived] = "\0";
    printf(" WSARecv succeeded \n
        Received bytes: %d \n
          bytes: %s \n ", bytesReceived,
                    recvBuffer);
  }
```

WSARecvfrom()

1) It is used to receive data from an specific address when using connectionless (UDP) sockets. It returns 0 in success and SOCKET_ERROR on failure.

Eg:-

```
WSABuf wsaBuf;
char recvMsg[512];

wsaBuf.len = sizeof(recvMsg);
wsaBuf.buf = recvMsg;

struct sockaddr_in senderAddr;
senderAddr.sin_family = AF_INET;
int senderAddrSize = sizeof(senderAddr);
DWORD bytesReceived = 0;
DWORD flags = 0;

if( WSARecvfrom(sock, &wsaBuf, 1, &bytesReceived, &flags,
    (SOCKADDR *)&senderAddr, &senderAddrSize, NULL, NULL)
    == SOCKET_ERROR)
{
    printf("WSARecvFrom failed with error: %d\n",
        WSAGetLastError());

}
```

required in winsock programming?
nking in winsock programming with 8

POKHARA UNIVERSITY
Level: Bachelor Semester:Spring
Programme: BE Year : 2024
Course: Network Programming Full Marks:100
Pass Marks:

24sp6a What is overlapped I/O? Which functions is used to set windows socket in non blocking mode? Explain non blocking socket with connect() in case of winsock architecture.

Solv

Functions used to set windows socket in non blocking mode:

@ ioctlsocket() {Recommended}

↳ ioctlsocket() function allows setting various socket options, including non blocking mode.

syntax is:

int ioctlsocket(SOCKET s, long cmd, u_long *argp);

where s: is the socket descriptor,

cmd: is the command to set socket in non blocking mode i.e. FIONBIO

*argp: pointer to a u_long type variable. 1 for non blocking & 0 for blocking.

Eg:

```
// other code
:
u_long mode=1;
if((ioctlsocket(sock, FIONBIO, &mode))!=0)
{
    printf("failed to set socket to non blocking mode");
}
else
{
    printf("socket is set to non blocking mode");
}
```

(b) WSAAsyncSelect()

↳ This function allows the window socket to operate asynchronously and notifies window message when an event occurs.

Syntax:

```
WSAAsyncSelect(sock, hWnd, WM_USER+1,
    FD_READ | FD_WRITE | FD_CONNECT);
```

(c) WSAEVENTSELECT

↳ Allows event based notification using WSAWaitForMultipleEvents();

Non Blocking socket with connect()

There is problem with blocking connect(). The function doesn't return control to program until the connection is established or fails, which can cause delays in program.

Here's how connect() works with non blocking socket.

Step1: set the blocking socket to non blocking using ioctlsocket() method.

Step2: use the connect(). Since the operation is non blocking so it returns immediately.

Step3: use select() or WSAPoll() to check whether the connection is established or not.

Eg:- #include<winsock2.h>
#include<stdio.h>

```
int main()
{
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2,2), &wsaData);

    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock==INVALID_SOCKET)
    {
        printf("socket creation failed\n");
        return 1;
    }

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

    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(8080);
    server.in_addr.sin_addr = inet_addr("192.168.
                                1.100");

if(connect(sock,(struct sockaddr*)&server, sizeof(server))
    == SOCKET_ERROR && WSAGetLastError() ==
        WSAEWOULDBLOCK)
{
    printf("connecting ....");

}
```

```
closesocket(sock);
WSACleanup();
return 0;
}
```

24s|06 Explain different async I/O function in ~~etc~~ console programming

Soln @ WSAAsyncSelect()

↳ It is an asynchronous I/O function used in console programming which will allow window socket to operate asynchronously and notifies ~~the~~ window message when an event occurs. or WM_USER+1,

Eg: WSAAsyncSelect(~~socket~~ sock, hWnd, WM_SOCKET, FD_READ | FD_WRITE | FD_CONNECT);

~~Here~~,

This registers the socket with hWnd (window handle) and notifies via WM_SOCKET message when data is available or connection closes etc.

(b) WSACancelAsyncRequest()

↳ It cancels the async request initiated by ~~WSAAsyncSelect~~ WSAAsyncSelect(). It prevents further notification from being sent.

Syntax: int WSACancelAsyncRequest(HANDLE dwContext);

Programme: BE Semester:Spring
Course: Network Programming Year : 2024
Full Marks:100

③ WSARecvEx() { Depricated }

↳ It is extended version of WSARecv() used for efficient data reception. It is used to handle large data efficiently but not recommended to use at present.

~~Syntax: int WSARecvEx(~~

Eg:

```
#define BUFFER_SIZE 1024
DWORD bytesReceived = 0;
DWORD flags = 0;
char buffer[BUFFER_SIZE];

int result = WSARecvEx(
        sock,
        buffer,
        BUFFER_SIZE,
        &bytesReceived,
        &flags);

if (result != SOCKET_ERROR)
{
    pf(" Received data : %s\n", buffer);
}
else
{ pf(" WSARecvEx failed\n Error: %d\n",
        WSAGetLastError());
}
```

23fall66

Explain the types of dynamic linking in winsock programming with pros and cons of each.

Sol4

(a) Load-time dynamic linking (implicit linking)
(b) Run-time dynamic linking (explicit linking).

Load time dynamic linking is the types where the DLL is linked during the compilation of the application. The application uses the import library (.lib) and header file (.h) to reference the DLL functions.

The DLL is loaded automatically when the application starts.
Example: using ws2_32.lib for winsock functions:

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

#pragma comment(lib, "ws2_32.lib") //link with winsock library

int main()
{
    WSADATA wsadata;
    WSAStartup(MAKEWORD(2,2), &wsadata);
    printf("Winsock initialized");
    WSACleanup();
    return 0;
}
```

POKHARA UNIVERSITY
Semester:Spring
Year : 2024
Full Marks:100
Pass Marks: 45
Time :3hrs.
Bachelor

Runtime dynamic linking (explicit linking)

- Here the application loads the DLL at the runtime depending on its needs using LoadLibrary() and GetProcAddress() function. It is more flexible and allows loading

Runtime dynamic linking (explicit linking)

- Here the application loads the DLL at the runtime depending on its needs using LoadLibrary() and GetProcAddress() functions. It is more flexible and allows loading different versions of the DLL dynamically.

ex: dynamically loading ws2_32.dll

```
#include<windows.h>
#include<stdio.h>

typedef int (*WSAStartup_t)(WORD, LPWSADATA);

int main()
{
    HMODULE hDLL = LoadLibrary("ws2_32.dll");

    if(!hDLL) return printf("failed to load DLL\n");
                    1;
                    → myWSAStartup variable etc

    if(
        (WSAStartup_t)GetProcAddress(hDLL, "WSAStartup")
    ){
        WSADATA wsaData;
        MyWSAStartup(MAKEWORD(2,2), &wsaData);
        printf("Winsock initialization");
    }

    FreeLibrary(hDLL);
    return 0;
}
```

23Sp65 Is it possible to write a common n/w apps that runs in both LINUX and windows? How would you do it. Show simple example code as well.

Soln Yes its possible to write a common n/w app that runs on both LINUX & win. The key is to use cross platform socket progremming, which include using an API which is used by both O.S.

The Berkeley Sockets API, which is used by LINUX and windows (via winsock) provides a common interface for n/w communication. With minimal changes, we can develop n/w based apps which works on both platform.

Steps to make it cross platform:

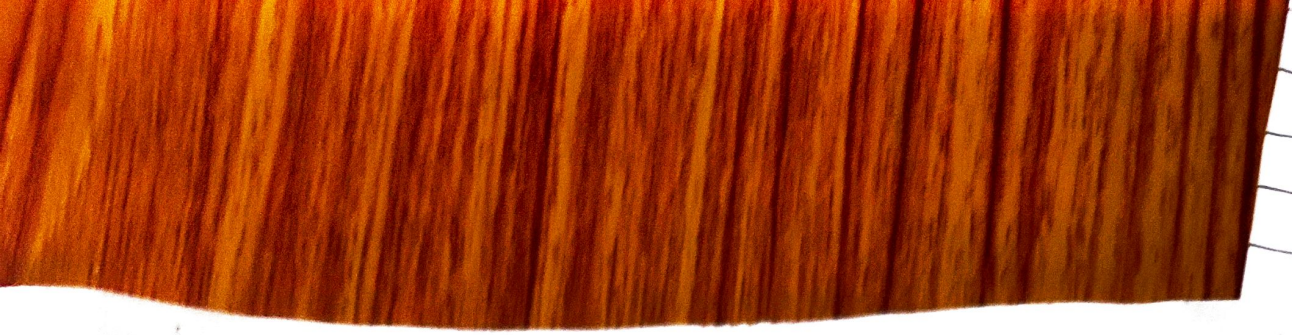
(a) Include the necessary header files, i.e. Winsock.h for windows and sys/socket.h for linux.

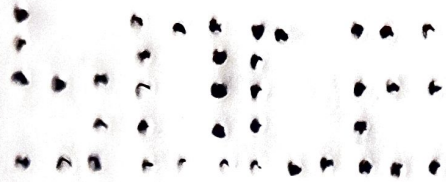
(b) Initialize winsock on window using WSAStartup() before using sockets.

(c) close sockets properly; using closesocket() for windows & close() for linux.

The simple example (in java)

```
import java.io.*;
import java.net.*;

public class Server
{
    public static void main (String[] args)
    {
        try{
            ServerSocket serverSocket = new
                                ServerSocket(8080);
            System.out.println("Server listening on port
                                8080");

            //waiting for a client connection
            Socket socket = serverSocket.accept();
            System.out.println("Client connected");

            //create input & o/p streams
            in = new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()
                    ));
            PrintWriter out = new PrintWriter(socket.getOutputStre
                                am(), true);

            //Reading data from the client
            String clientMsg = in.readLine();
            System.out.println("client message: " + clientMsg);
```





```
//sending response to the client
out.println("Hello from server");

//close conn
    socket.close();
    serversocket.close();

} catch (IOException e)
    {
        e.printStackTrace();

    }

} }

}
```

client

```java
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try {
            // Connect to the server
            Socket socket = new Socket("127.0.0.1", 8080);
            System.out.println("Connected to server");

            // Create input and output streams
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            // Send message to the server
            out.println("Hello from client");

            // Read the response from the server
            String serverResponse = in.readLine();
            System.out.println("Server reply: " + serverResponse);

            // Close the connection
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

नेपाल सरकार
शिक्षा मन्त्रालय
परीक्षा नियन्त्रण कार्यालय, सानोठिमी
एसएलसी थप उत्तरपुस्तिका

विषय..... मिति .....
मूल उत्तरपुस्तिका नं .....
निरीक्षकको सही: .....

थप उत्तरपुस्तिकाको दुवैतर्फ यसै पेजदेखि लेख्नुहोला । यसमा सि.नं. तथा अन्य कुनै सङ्केत लेखेको पाइएमा परीक्षा रद्द हुनेछ ।

Q1 fall 5b Differentiate winsock and unix (linux) socket implementation with the help of respective functions.

feature	Winsock (windows)	Unix socket (Linux)
Initialization	WSAStartup()	Not required
cleanup	WSACleanup()	Not required.
Error handling	WSASetLastError(), WSAGetLastError()	perror(), errorno
create socket	socket()	socket()
bind socket	bind()	bind()
listen for conn	listen()	listen()
accept connection	accept()	accept()
send data	send()	send()
recv data	recv()	recv()
close socket	closesocket()	close()
Header file	<winsock2.h>	<sys/socket.h> <net/inet.h> <arpa/inet.h>

Comparison of Key Functions

Below are some key functions in both implementations.

1. Socket Initialization

Winsock:

```cpp
WSADATA wsa;
WSAStartup(MAKEWORD(2, 2), &wsa);
```

Unix/Linux:

```cpp
// No initialization needed in Linux
```

2. Creating a Socket

Winsock:

```cpp
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
```

Unix/Linux:

```cpp
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

3. Binding a Socket

Winsock:

```cpp
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(8080);

bind(sock, (struct sockaddr*)&server, sizeof(server));
```

Unix/Linux:

(Same as Winsock, except socket descriptor type)

```cpp
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(8080);

bind(sock, (struct sockaddr*)&server, sizeof(server));
```

4. Listening for Connections

Winsock:

```cpp
listen(sock, 5);
```

Unix/Linux:

```cpp
listen(sock, 5);
```

5. Accepting a Connection

Winsock:

```cpp
SOCKET client = accept(sock, NULL, NULL);
```

Unix/Linux:

```cpp
int client = accept(sock, NULL, NULL);
```

6. Sending Data

Winsock:

```cpp
send(sock, "Hello", strlen("Hello"), 0);
```

Unix/Linux:

```cpp
send(sock, "Hello", strlen("Hello"), 0);
```

7. Receiving Data

Winsock:

```cpp
recv(sock, buffer, sizeof(buffer), 0);
```

Unix/Linux:

```cpp
recv(sock, buffer, sizeof(buffer), 0);
```

8. Closing a Socket

Winsock:

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

Unix/Linux:

```cpp
close(sock);
```

```c
/*=========================Close function demo code====================*/
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main()
{
    // Create a socket
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("Socket creation failed");
        return -1;
    }

    // Define server address
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080);
    server_address.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connect to the server
    if (connect(sockfd, (struct sockaddr *)&server_address, sizeof(server_address)) < 0)
    {
        perror("Connection failed");
        close(sockfd); // Close socket if connection fails
        return -1;
    }

    // Send data (for example, sending a simple HTTP request)
    char message[] = "GET / HTTP/1.1\r\nHost: localhost\r\n\r\n";
    send(sockfd, message, sizeof(message), 0);

    // Close the socket after use
    close(sockfd); // Releases the socket resource
    return 0;
}

```

```c
/*=========================Shutdown function demo code==================*/
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main()
{
    // Create a socket
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("Socket creation failed");
        return -1;
    }

    // Define server address
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080);
    server_address.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connect to the server
    if (connect(sockfd, (struct sockaddr *)&server_address, sizeof(server_address)) < 0)
    {
        perror("Connection failed");
        close(sockfd); // Close socket if connection fails
        return -1;
    }

    // Send data (e.g., sending an HTTP request)
    char message[] = "GET / HTTP/1.1\r\nHost: localhost\r\n\r\n";
    send(sockfd, message, sizeof(message), 0);

    // Shutdown the socket for sending data
    shutdown(sockfd, SHUT_WR); // No more data will be sent (write is disabled)

    // Optionally, you could read response from server here
    // char response[1024];
    // recv(sockfd, response, sizeof(response), 0);

    // Close the socket after shutdown
    close(sockfd); // Releases the socket resource
    return 0;
}

```

What is overlapped I/O?

Overlapped I/O (asynchronous I/O) is a mechanism that allows Windows applications to perform non-blocking operations efficiently. In Winsock, **overlapped I/O** means that an operation (such as sending or receiving data) is initiated and then performed in the background, allowing the application to continue executing other tasks without waiting for the operation to complete.

Key Features of Overlapped I/O:

- Allows multiple socket operations to execute concurrently.
- Uses the `OVERLAPPED` structure to manage asynchronous operations.
- Requires event objects or completion routines for notification when an operation is completed.
- Ideal for high-performance applications, such as server applications handling multiple clients.

Functions Supporting Overlapped I/O in Winsock:

- `WSASend()`
- `WSASendTo()`
- `WSARecv()`
- `WSARecvFrom()`
- `AcceptEx()`
- `ConnectEx()`
- `TransmitFile()`

These functions support an **optional** `OVERLAPPED` parameter, allowing them to execute asynchronously.

In what situation asynchronous I/O is preferred over synchronous IO. Explain different Async IO functions in winsock programming.

Asynchronous I/O is preferred over synchronous I/O in situations where:

1. **Non-blocking Execution is Needed**:
    - The application should continue executing other tasks instead of waiting for I/O operations to complete.
    - Example: A GUI application should remain responsive while handling network communication.
2. **High Performance and Scalability**:
    - When dealing with multiple concurrent connections, asynchronous I/O avoids the overhead of creating multiple threads.
    - Example: A web server handling thousands of client requests.
3. **I/O-bound Applications**:
    - Applications where the CPU spends more time waiting for data (like fetching data from a database or network) benefit from asynchronous I/O.
    - Example: Online gaming, stock market applications, and video streaming.
4. **Event-driven Programming**:
    - Asynchronous I/O allows event-based notification instead of constantly polling for data.
    - Example: Applications that rely on Windows message handling (GUI apps).

How do you implement window's socket descriptor ??

In Windows, a **socket descriptor** is implemented using the `SOCKET` type provided by the Winsock API. Unlike UNIX, where sockets are file descriptors (integers), Windows defines `SOCKET` as an **unsigned integer (HANDLE-like type)**.

Steps to Implement a Windows Socket Descriptor

1. **Initialize Winsock** using `WSAStartup()`.
2. **Create a socket** using `socket()`, which returns a `SOCKET` descriptor.
3. **Use the socket descriptor** for operations like `bind()`, `connect()`, `send()`, `recv()`, etc.
4. **Close the socket** using `closesocket()`.
5. **Cleanup Winsock** using `WSACleanup()`

==========

What is Windows Sockets (Winsock)? What are API, SPI and ABI in windows kernel architecture? Explain

Windows Sockets (**Winsock**) is an API that provides a **standardized way for applications to communicate over network protocols (such as TCP/IP)** in **Windows OS**. It allows programs to send and receive data using **sockets**, which are endpoints for communication between machines.

Winsock is based on the **Berkeley Sockets API** (used in UNIX/Linux) but includes Windows-specific enhancements. It supports both **blocking and non-blocking (asynchronous) communication**.

API, SPI, and ABI in Windows Kernel Architecture

In Windows networking and kernel architecture, **API, SPI, and ABI** refer to different layers of interaction between software and hardware.

1. API (Application Programming Interface)

- The **API** is a set of **functions and libraries** that applications use to interact with the system.
- In **Winsock**, the API includes:
    - `socket()`, `bind()`, `connect()`, `send()`, `recv()`, `select()`, etc.
- Located in **User Mode**.
- Example: **Winsock 2.0 API (`Ws2_32.dll`, `Mswsock.dll`)**.

2. SPI (Service Provider Interface)

- The **SPI** allows third-party developers to extend the networking capabilities of Winsock.
- It acts as an **intermediate layer** between the API and the lower networking stack.
- Examples:
    - Layered Service Providers (**LSP**) like **`Msafd.dll`**, which helps in handling network traffic.
    - Helper DLLs (**`Wship6.dll`**, **`Wshtcpip.dll`**) that provide protocol-specific implementations.

3. ABI (Application Binary Interface)

- The **ABI** defines the **low-level binary interface** between the application and the system.
- It ensures that compiled programs work across different Windows versions without modification.
- It includes system calls, memory layout, and calling conventions.
- Example:
    - The **TDI (Transport Driver Interface)** layer (`Afd.sys`), which connects Winsock with the Windows Kernel.

==========

How windows sockets are different from berkeley sockets. Explain in short

Windows Sockets (Winsock) and Berkeley Sockets (BSD Sockets) are both API standards for network communication, but they differ mainly in their origin, platform, and some details in their implementation.

1. **Origin**:

- **Berkeley Sockets**: Developed as part of the BSD Unix operating system, BSD Sockets is the standard socket interface for Unix-like systems.

- **Windows Sockets (Winsock)**: Created by Microsoft to provide a network API for Windows operating systems.

2. **Platform**:

   - **Berkeley Sockets**: Primarily used on Unix-like systems (Linux, macOS, BSD).

   - **Windows Sockets**: Used on Windows-based systems.

3. **API Differences**:

   - **BSD Sockets**: The API is relatively simple and integrates well with Unix-like system calls. It directly interacts with the kernel, making it lightweight and efficient.

   - **Winsock**: While based on BSD Sockets, Winsock has some additional features specific to Windows, such as handling Windows-specific networking concepts, asynchronous I/O, and integration with Windows services.

4. **Initialization**:

   - In **Winsock**, you must call `WSAStartup()` to initialize the library before using any socket functions, and `WSACleanup()` to clean up when done.

   - **BSD Sockets** don't require explicit initialization or cleanup calls.

==========

In what approach of communication, does Windows socket provide better functionalities than unix socket? What will happen if you call shutdown() and close() functions? These methods are different from the WSACleanup() function. Explain how/how not

Windows Sockets (Winsock) offer more functionality than Unix Sockets in terms of being feature-rich and integrated with the broader Windows network stack. Winsock provides many advanced network features and services out of the box, which are tightly integrated with the Windows operating system. As a result, users don't need to rely on third-party libraries or services to access these features. This makes

Winsock more convenient and feature-complete for developers working within the Windows ecosystem.

On the other hand, Unix Sockets are generally more lightweight due to Unix's design philosophy. The focus is on simplicity, efficiency, and minimalism, which results in a lower resource consumption compared to Winsock. While Unix Sockets might require third-party libraries for certain advanced features, their design allows for better performance in resource-constrained environments.

Thus, Winsock is better suited for applications that require rich networking features and tight integration with the Windows OS, whereas Unix Sockets are preferred for lightweight, performance-critical applications on Unix-based systems.

`shutdown()`:

- **Purpose**: The `shutdown()` function is used to disable certain operations on a socket. It can be used to stop further communication in a specific direction (reading, writing, or both).
    - `shutdown(socket, SHUT_RD)` disables reading on the socket.
    - `shutdown(socket, SHUT_WR)` disables writing on the socket.
    - `shutdown(socket, SHUT_RDWR)` disables both reading and writing.
- **Difference**: Unlike `close()`, which completely closes the socket, `shutdown()` allows you to stop only the data flow in a particular direction. It is useful when you want to gracefully shut down a socket (i.e., notify the remote end that no more data will be sent or received).

`close()`:

- **Purpose**: The `close()` function is used to release the socket descriptor and terminate the connection. It should be called after communication is finished. Once `close()` is invoked, the socket is effectively unusable.
- **Difference**: This function is used to cleanly release the socket, closing it fully. After `close()`, no further operations can be performed on the socket.

`WSACleanup()` (Windows only):

- **Purpose**: `WSACleanup()` is used in Windows Sockets programming to clean up the Winsock library once all socket operations are completed. It should be called

after all Winsock operations are finished, typically at the end of the program.

- **Difference**: Unlike shutdown() and close(), WSACleanup() is specific to Windows and is used for cleaning up the Winsock subsystem. It is necessary to call it when done using Winsock, as it ensures proper cleanup of resources, particularly on Windows systems that require explicit initialization and cleanup.

==========

How do you implement Stream Communication in winsock? Describe each step with the help of relevant APIs

Code Examples

1. Initialize Winsock

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

2. Create a Socket

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

3. Bind the Socket (Server Only)

```
sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_addr.s_addr = INADDR_ANY;
serverAddr.sin_port = htons(8080);

if (bind(serverSocket, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)
{
   printf("Bind failed: %d\n", WSAGetLastError());
   closesocket(serverSocket);
   WSACleanup();
   return 1;
}
```

4. Listen for Incoming Connections (Server Only)

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

5. Accept a Connection (Server Only)

```
SOCKADDR_IN clientAddr;
int clientSize = sizeof(clientAddr);
SOCKET clientSocket = accept(serverSocket, (SOCKADDR*)&clientAddr, &clientSize);
if (clientSocket == INVALID_SOCKET) {
   printf("Accept failed: %d\n", WSAGetLastError());
   closesocket(serverSocket);
   WSACleanup();
   return 1;
}
printf("Client connected!\n");
```

6. Connect to Server (Client Only)

```
SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

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

7. Send and Receive Data

```
const char* message = "Hello from client!";
send(clientSocket, message, strlen(message), 0);

char buffer[1024];
```

```
int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);
if (bytesReceived > 0) {
   buffer[bytesReceived] = '\0';
   printf("Received: %s\n", buffer);
}
```

8. Close the Connection

```
shutdown(clientSocket, SD_BOTH);
closesocket(clientSocket);
```

9. Cleanup Winsock

```
WSACleanup();
```

==========

Explain select() function in conjunction with accept() call in winsock

The select() function in Winsock allows a program to monitor multiple sockets for readiness to perform I/O operations, such as reading, writing, or handling exceptions. When used with accept(), it helps manage multiple incoming client connections efficiently without blocking execution.

Why Use select() with accept()?

1. Non-blocking I/O Management: accept() is normally a blocking call. Using select() first ensures accept() is only called when a new connection request is available.

2. Handling Multiple Clients: Instead of using multiple threads or a loop with blocking accept(), select() monitors multiple sockets and handles new connections efficiently.

3. Efficient Resource Usage: It avoids busy-waiting by allowing the program to wait for new connections and data on existing sockets simultaneously.

How select() Works

1. select() checks a set of sockets and determines if they are ready for:
    a. Reading (readfds) → Useful for accept() or recv().
    b. Writing (writefds) → Useful for send().
    c. Exception Handling (exceptfds) → Detects out-of-band data or socket errors.
2. If select() detects an event, the program can process the respective socket.

==========

Differentiate between Load time dynamic linking and runtime dynamic linking

1. **Time of Linking**
    - **Load-Time Dynamic Linking**: The DLL (Dynamic Link Library) is linked when the program is loaded into memory before execution starts.
    - **Run-Time Dynamic Linking**: The DLL is linked explicitly during execution using functions like LoadLibrary (Windows) or dlopen (Linux).
2. **Function Access**
    - **Load-Time Dynamic Linking**: Functions from the DLL can be called directly as if they were part of the executable.
    - **Run-Time Dynamic Linking**: Functions must be accessed via function pointers obtained using GetProcAddress (Windows) or dlsym (Linux).
3. **Performance Impact**
    - **Load-Time Dynamic Linking**: Slightly faster as all dependencies are resolved before execution begins.
    - **Run-Time Dynamic Linking**: May introduce a slight performance overhead since functions are loaded and resolved dynamically at runtime.

4. **Flexibility**

    - **Load-Time Dynamic Linking**: Less flexible because all required libraries must be available when the program starts.

    - **Run-Time Dynamic Linking**: More flexible since libraries can be loaded and unloaded as needed during execution.

5. **Error Handling**

    - **Load-Time Dynamic Linking**: If a required DLL is missing or corrupted, the program fails to start.

    - **Run-Time Dynamic Linking**: The program can handle missing DLLs gracefully by attempting alternative solutions.

6. **Memory Usage**

    - **Load-Time Dynamic Linking**: The library remains loaded for the entire duration of the program.

    - **Run-Time Dynamic Linking**: The library can be loaded and unloaded as needed, potentially reducing memory usage.

7. **Use Cases**

    - **Load-Time Dynamic Linking**: Used when all dependencies are known beforehand and need to be available at startup.

    - **Run-Time Dynamic Linking**: Useful when working with optional plugins, modules, or dynamically determined dependencies.

==========