CH3-Windows Socekt Proramming Slides

network programming (Pokhara University)

since 2001

COSMOS COLLEGE
OF MANAGEMENT AND TECHNOLOGY

Affiliated to Pokhara University

Windows Socket Programming

Instructor

Dr. Babu R Dawadi

Windos Socket basics

- WinSock provides an interface to network protocols
  - Windows Sockets, derived from the BSD Sockets API
- A *socket* is an *endpoint* of a bi-directional communication flow over an IP-based network (like Internet).
  - Characterized by a protocol type, remote socket address and local socket address.
  - Socket addresses consist of an IP-address and a Port Number.
    - IP-address maps a number to a specific computer
    - Port number maps a number to a specific process/application running on a computer

Client/Server communication

- So...how can two computers communicate over a network?
  - **Client/Server** model
- Server: program/computer providing a service
  - Creates a local socket
  - Binds local socket to a specific port
  - Listens for incoming connections
  - Accepts a connection, assigning a new socket for the connection
  - Sends/receives data

Client: Requesting a service,
- Determines remote socket address,
- Creates a local socket,
- Connects to remote socket,
- Sends/receives data

Setting up the WinSock API

- Include the winsock2.h header file:

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

- Link with WinSock library:
  type `ws2_32.lib` in project->properties->linker->commandline->additional options

- Initizalize Winsocket API:
  - Declare a 'WSADATA' variable to store info on WinSock (you don't need to set it, only declaration is enough) - `WSADATA info;`

- Call to WSAStartup function:

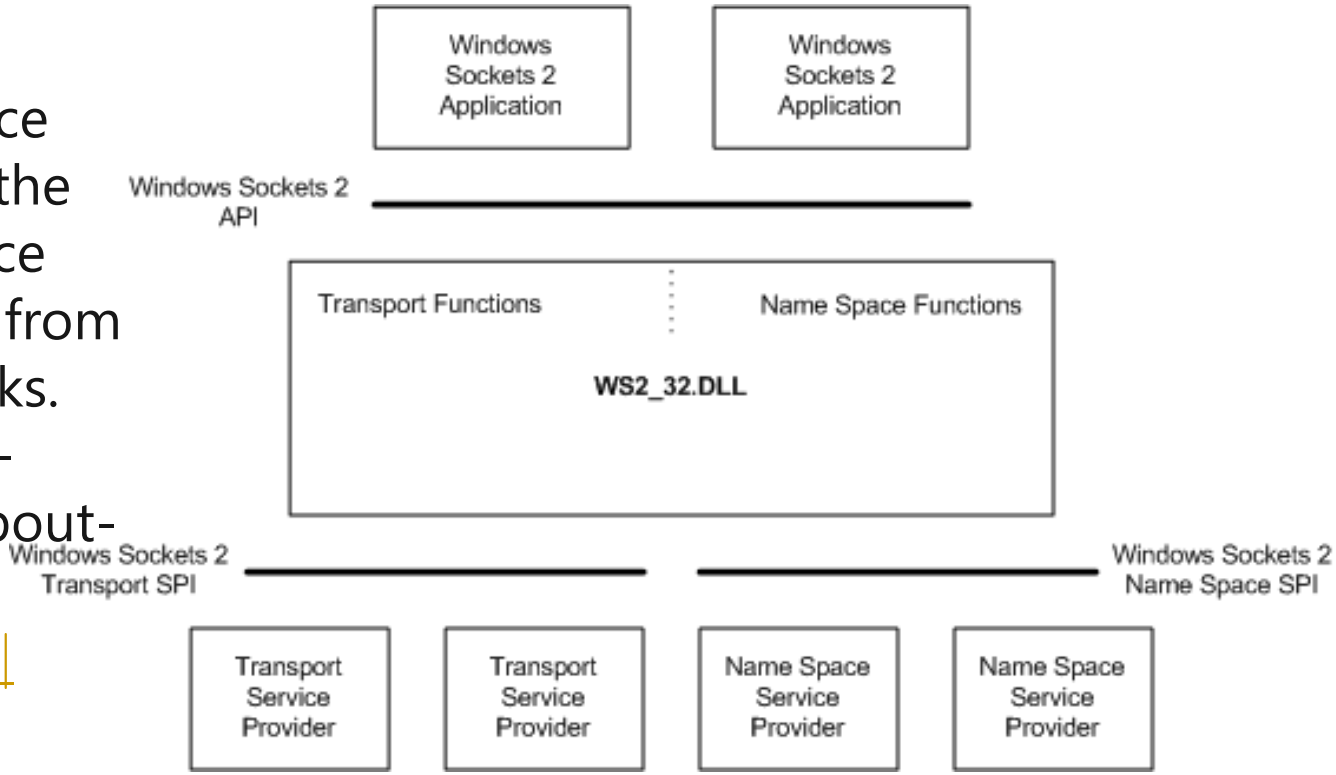
```
if (WSAStartup(MAKEWORD(1,1), &info) !=0){

    printf("Winsock startup failed\n");

    exit(1);

}
```

Winsock Architecture

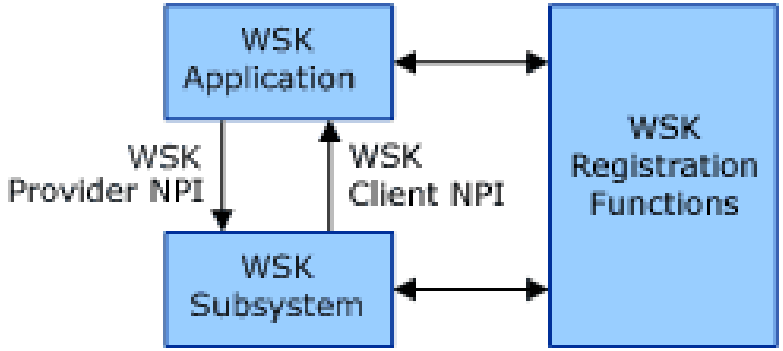
- The Windows Sockets 2 architecture is compliant with the Windows Open System Architecture (WOSA), as illustrated in Figure

Winsock defines a standard service provider interface (SPI) between the application programming interface (API), with its functions exported from WS2_32.dll and the protocol stacks.

- https://docs.microsoft.com/en-us/windows/win32/winsock/about-winsock
- About Winsock - Win32 apps | Microsoft Docs



Winsock Kernel Architecture



WSK applications discover and attach to the WSK subsystem by using a set of WSK registration functions. Applications can use these functions to dynamically detect when the WSK subsystem is available and to exchange dispatch tables that constitute the provider and client side implementations of the WSK NPI.

- At the core of the WSK architecture is the WSK subsystem.
- The WSK subsystem is a network module that implements the provider side of the WSK Network Programming Interface (NPI).
- The WSK subsystem interfaces with transport providers on its lower edge that provide support for various transport protocols.
- WSK applications are kernel-mode software modules that implement the client side of the WSK NPI in order to perform network I/O operations.

API, SPI and ABI

- Application Provider Interface (API) for application developer
- Service Provider Interface (SPI) for Network software vendors
- Application Binary Interface (ABI-binary interface) to ensure consistent interoperability between applications and various implementation of Winsock. It is an interface between two program modules, often between operating systems and user programs.

WSK communication-setting up a server (1)

- Create local socket:
  - Declare a variables of type sockaddr and type SOCKET:

```
struct sockaddr_in server_address;

struct sockaddr_in client_address;
 SOCKET s, new_s;
```

  - Set up the local socket:

```
#define SERVER_PORT 5432 //port number of
server > 1024
 server_address.sin_family = AF_INET;

server_address.sin_addr.s_addr = INADDR_ANY;

server_address.sin_port = htons(SERVER_PORT);

memset(&(server_address.sin_zero),'\0', 8);
```

Create a new socket

```
s = socket(PF_INET, SOCK_STREAM, 0);
  if (s == INVALID_SOCKET) {
   printf("Socket creation failed\n");
        exit(1);
    }
```

Setting up a server (2)

- Bind the socket (associate with a specific port number):

```
if (bind(s, (struct sockaddr
 *)&server_address,
 sizeof(server_address)) ==
 SOCKET_ERROR) {

   printf("Bind failed\n");

   exit(1);

 }
```

- Listen for incoming connections on the new socket:

```
#define MAX_PENDING 2 //max #
 simultaneous connections

listen(s, MAX_PENDING);
```

Accept incoming connection; connects to a new socket:

```
len = sizeof(client_address);
new_s = accept(s,(struct sockaddr
*)&client_address,&len);
if (new_s == INVALID_SOCKET)
{printf("Accepting connection failed \n");
        exit(1);
    }
```

Setting up a server (3)

- Now we can send and receive:
  - `send (SOCKET s, const char FAR * buf, int len, int flags);`
  - `recv (SOCKET s, char FAR * buf, int len, int flags);`
  - Example: receive a string, and then send back an echo:

```
char buf[50];
n=recv(new_s, buf,sizeof(buf),0);
buf[n]='\0';
send(new_s, buf, n, 0);
```

At the end of the program, free socket resources:

```
closesocket(s);
WSACleanup();
```

Setting up a client (1)

- Initialize WinSock API, like we did with the server:
  - Include header, link with library
  - Start up WinSock:

```
 WSADATA info;

  if (WSAStartup(MAKEWORD(1,1),
&info) !=0){

    printf("Winsock startup
failed\n");

    exit(1);

 }
```

Lookup remote server address:

Declare variable of type 'struct hostent':

```
struct hostent *hp;
```

Get server address and store it in variable:

```
    char *servername;
   hp =
gethostbyname(servername);
    if (!hp) {
      printf("Unknown server:
%s\n",servername);
      exit(1);
    }
```

Setting up a client (2)

- Set up socket:

```
SOCKET s;

struct sockaddr_in server_address;

server_address.sin_family = AF_INET;

server_address.sin_addr = *((struct in_addr *)hp->h_addr);

server_address.sin_port = htons(SERVER_PORT);

memset(&(server_address.sin_zero),'\0',8);
```

Create socket:

```
s = socket(PF_INET, SOCK_STREAM,0);
if (s == INVALID_SOCKET){
   printf("Socket creation
failed");
   exit(1);
}
```

Setting up a client (3)

- Connect to remote socket of server:

```
if (connect(s,(struct
 sockaddr
 *)&server_address,

 sizeof(server_address))
 == SOCKET_ERROR) {

   printf("Connecting
 to server failed");

   closesocket(s);

   exit(1);

 }
```

Now we can send and receive:

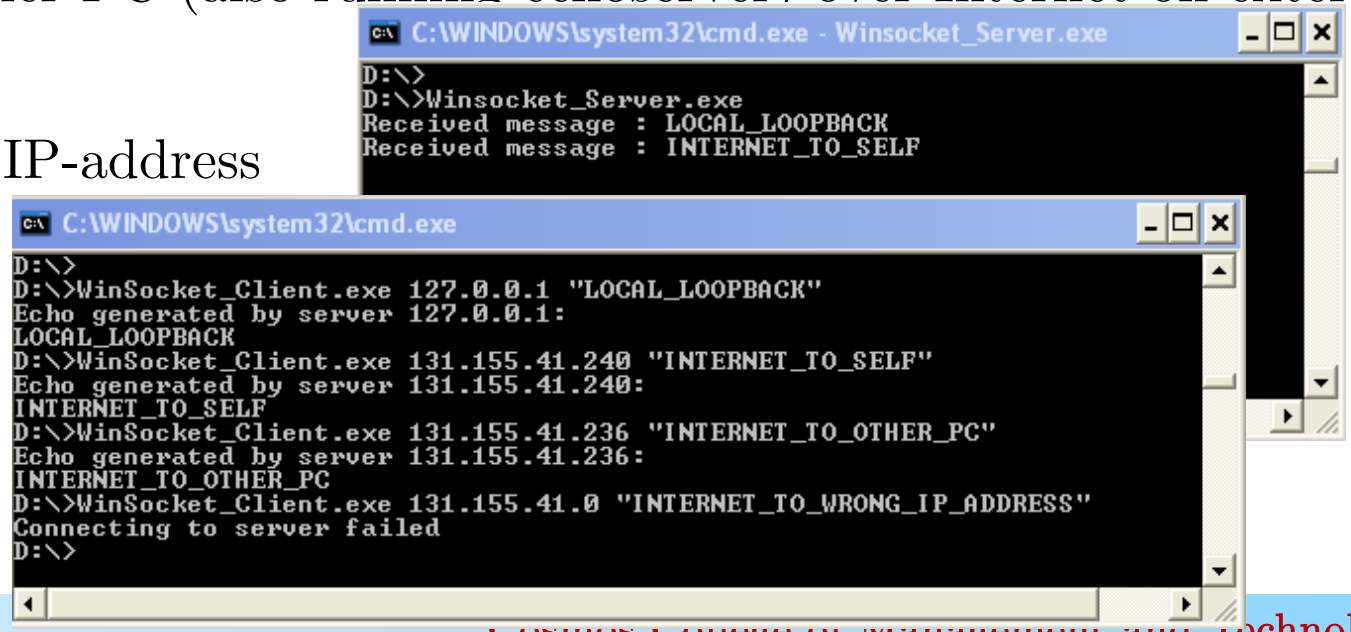
```
send (SOCKET s, const char FAR * buf, int
len, int flags);
recv (SOCKET s, char FAR * buf, int len,
int flags);
```

Example: sending a string and receiving an echo from server:

```
 char buf[256], buf_in[256];
 n = strlen(buf);
   send(s,buf,n,0);
   n=recv(s, buf_in, sizeof(buf_in),
0);
   buf_in[n] = '\0';
   printf("Echo generated by server
%s:\n",

inet_ntoa(server_address.sin_addr));
   printf(buf_in);
```

The Echo server/client in action…

- Client sends strings to
  - Server on same pc, on local loopback address
  - Server on same pc over Internet on external IP-address
  - Server on other PC (also running echoserver) over Internet on external IP-address
  - Some wrong IP-address



Winsock Lab Turotiral (Lab 7, 8, 9)

- Refer the examples from The Windows sockets and Win32 network programming with practical program examples using socket() and bind() functions (tenouk.com)
- https://www.tenouk.com/Winsock/Winsock2example1.html

- Create basic winsock application
- https://docs.microsoft.com/en-us/windows/win32/winsock/creating-a-basic-winsock-application

Dynamic Link Libraries (DLL)

- Dynamic Link Library (DLL) is Microsoft's implementation of the shared library concept.
- A DLL is a library that contains code and data that can be used by more than one program at the same time.
- By using a DLL, a program can be modularized into separate components.
- For the Windows operating systems, much of the functionality of the operating system is provided by DLL.
- The use of DLLs helps promote modularization of code, code reuse, efficient memory usage, and reduced disk space.

DLL Advantages

- Uses fewer resources - When multiple programs use the same library of functions, a DLL can reduce the duplication of code that is loaded on the disk and in physical memory.
- Promotes modular architecture - A DLL helps promote developing modular programs. It helps you develop large programs that require multiple language versions or a program that requires modular architecture.
- Eases deployment and installation - When a function within a DLL needs an update or a fix, the deployment and installation of the DLL does not require the program to be relinked with the DLL. Additionally, if multiple programs use the same DLL, the multiple programs will all benefit from the update or the fix.

Some DLL Files

- **COMDLG32.DLL** – Controls the dialog boxes.
- **GDI32.DLL** – Contains numerous functions for drawing graphics, displaying text, and managing fonts.
- **KERNEL32.DLL** – Contains hundreds of functions for the management of memory and various processes.
- **USER32.DLL** – Contains numerous user interface functions. Involved in the creation of program windows and their interactions with each other.

DLL Types

- When you load a DLL in an application, two methods of linking let you call the exported DLL functions.
- The two methods of linking are load-time dynamic linking and run-time dynamic linking.
- In load-time dynamic linking, an application makes explicit calls to exported DLL functions like local functions. To use load-time dynamic linking, provide a header (.h) file and an import library (.lib) file when you compile and link the application.
- In run-time dynamic linking, an application calls either the LoadLibrary function or the LoadLibraryEx function to load the DLL at run time. After the DLL is successfully loaded, you use the GetProcAddress function to obtain the address of the exported DLL function that you want to call.

DLL entry points

- When you create a DLL, you can optionally specify an entry point function. The entry point function is called when processes or threads attach themselves to the DLL or detached themselves from the DLL.
- You can use the entry point function to initialize data structures or to destroy data structures as required by the DLL.
- In Visual C++ 6.0, you can create a DLL by selecting either the Win32 Dynamic-Link Library project type or the MFC AppWizard (dll) project type.

Sample DLL/VisualC++/win32.dll

Creating DLL

```
// SampleDLL.cpp

#include "stdafx.h"
#define EXPORTING_DLL
#include "sampleDLL.h"

BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call,
LPVOID lpReserved )
{return TRUE;}
void HelloWorld()
{
   MessageBox( NULL, TEXT("Hello World"),
   TEXT("In a DLL"), MB OK);
}
// File: SampleDLL.h
//
#ifndef INDLL_H
#define INDLL_H

   #ifdef EXPORTING_DLL
      extern __declspec(dllexport) void HelloWorld() ;
   #else
      extern __declspec(dllimport) void HelloWorld() ;
   #endif

#endif
```

Calling DLL

```
// SampleApp.cpp

#include "stdafx.h"
#include "sampleDLL.h"

int APIENTRY WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
   HelloWorld();
   return 0;
}
```

DLL Registration

- In order to use a DLL, it has to be registered by having appropriate references entered in the Registry.
- It sometimes happens that a Registry reference gets corrupted and the functions of the DLL cannot be used anymore. The DLL can be re-registered by opening Start-Run and entering the following command:
  - regsvr32 somefile.dll
- To unregister
  - regsvr32 /u somefile.dll

DLL UPS

- UPS – universal problem solver
- The DLL UPS tool is used to audit, compare, document, and display DLL information. The following list describes the utilities that make up the DUPS tool:
  - **Dlister.exe** – This utility enumerates all the DLLs on the computer and logs the information to a text file or to a database file.
  - **Dcomp.exe** – This utility compares the DLLs that are listed in two text files and produces a third text file that contains the differences.
  - **Dtxt2DB.exe** – This utility loads the text files that are created by using the Dlister.exe utility and the Dcomp.exe utility into the dllHell database.
  - **DlgDtxt2DB.exe** – This utility provides a graphical user interface (GUI) version of the Dtxt2DB.exe utility.

DLL Vs EXE

- An exe always runs in its own address space i.e., It is a separate process.
- The purpose of an EXE is to launch a separate application of its own.
- An EXE file contains the entry point or the part in the code where the operating system is supposed to begin the execution of the application.
- The file format of DLL and exe is essentially the same. Its reusability is the most major advantage of DLL files. A DLL file can be used in other applications as long as the coder knows the names and parameters of the functions and procedures in the DLL file. Because of this capability, DLL files are ideal for distributing device drivers.

DLL Vs EXE

- The DLL would facilitate the communication between the hardware and the application that wishes to use it. The application would not need to know the intricacies of accessing the hardware just as long as it is capable of calling the functions on the DLL.
- DLL files cannot be executed on their own. In other words, a dll always needs a host exe to run. i.e., it can never run in its own address space. The purpose of a DLL is to have a collection of methods/classes which can be re-used from some other application.
- How to create and use DLL : see example from https://www.bogotobogo.com/Win32API/Win32API_DLL.php

Windos soecket descriptor

- Each socket is identified by an integer called socket descriptor, which is an unsigned integer.
- A process may open multiple sockets for multiple concurrent communication sessions.
- Windows keeps a table of socket descriptors for each process.
- Each socket descriptor is associated with a pointer, which points to a data structure that holds the information about the communication session of that socket
- The data structure contains many fields, and they will be filled as the application calls additional WinSock functions.

Windos soecket descriptor

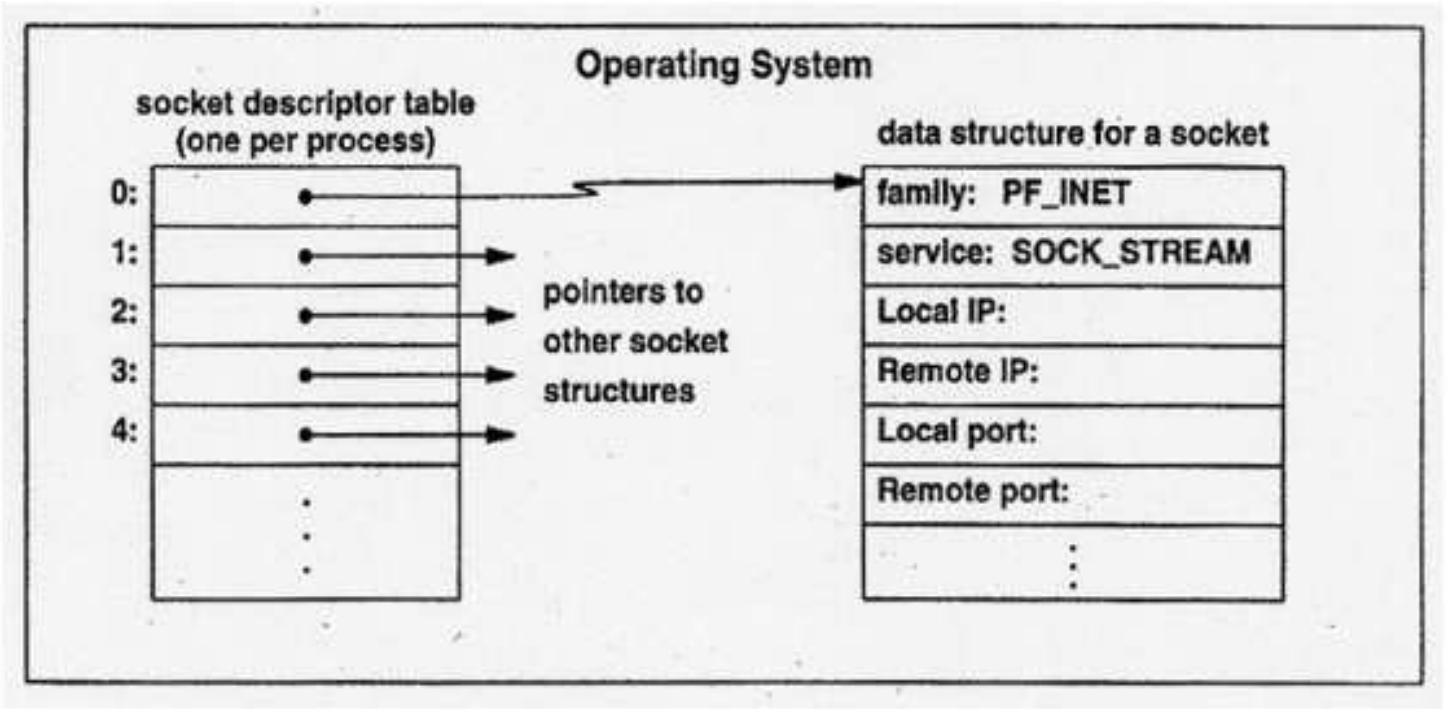


Fig 5.2 from Comer.

Conceptual operating system (Windows) data structures after **five calls** to *socket()* by a process. The system keeps a separate socket descriptor table for each process; threads in the process share the table.

Winsock IO

- Two calls for sending: WSPSend and WSPSendTo and the two calls for receiving: WSPRecv and WSPRecvFrom.
- Service providers determine how to perform the I/O operation based on socket modes, attributes, and the input parameter values.
- Any I/O operation with a blocking socket will not return until the operation has been fully completed.
- Any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer.

Winsock IO

- There are three primary ways of doing I/O in Windows Sockets 2:
  - Blocking I/O.
  - Nonblocking I/O along with asynchronous notification of network events.
  - Overlapped I/O with completion indication.
- Blocking I/O is the default behavior, nonblocking mode can be used on any socket that is placed into nonblocking mode, and overlapped I/O can only occur on sockets that are created with the overlapped attribute.
- Reference: https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5.html

Socekt I/O Models

- Essentially six types of socket I/O models are available that allow Winsock applications to manage I/O:
  - *Blocking*,
  - *Select*,
  - *SAAsyncSelect*,
  - *SAEventSelect*,
  - *Overlapped*, and
  - *Completion port*

Blocking IO

- Blocking sockets cause concern because any Winsock API call on a blocking socket can do just that-block for some period of time.

Blocking: Invoking a function which does not return until the associated operation is completed.

- The problem with this code is that the *recv* function might never return if no data is pending because the statement says to return only after reading some bytes from the system's input buffer.

```
SOCKET  sock;
char    buff[256];
int     done = 0;
    ...


while(!done)
{
      nBytes = recv(sock, buff, 65);
      if (nBytes == SOCKET_ERROR)
      {
          printf("recv failed with error %d\n",
              WSAGetLastError());
          Return;
      }
      DoComputationOnData(buff);
}
```

- One drawback of blocking sockets is that communicating via more than one connected socket at a time becomes difficult for the application.

Blocking IO

- An obvious example is a recv() which may block until data has been received from the peer system.
- The completion of functional task or the call of WSACancelBlockingCall() resulted to complete the blocking function with appropriate results.
- If a Windows message is received for a process for which a blocking operation is in progress, there is a risk that the application will attempt to issue another Windows Sockets call.
- WSAIsBlocking() may be called to determine whether or not a blocking Windows Sockets call is in progress.

Non-Blocking IO

- Nonblocking sockets are a bit more challenging to use, but every bit are as powerful as blocking sockets.
- Once a socket is placed in nonblocking mode, Winsock API calls return immediately.

- In most cases, these calls fail with the error WSAEWOULDBLOCK, which means that the requested operation did not have time to complete during the call
- For example, a call to *recv* returns *WSAEWOULDBLOCK* if no data is pending in the system's input buffer.

```
SOCKET          s;
unsigned long ul = 1;
int             nRet;


s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIOBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
      // Failed to put the socket into nonblocking mode
}
```

Winsock Non-Blocking IO

- Service providers determine how to perform the I/O operation based on socket modes, attributes, and the input parameter values.
- Any I/O operation with a blocking socket will not return until the operation has been fully completed.
- Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer.
- These events can be polled or waited on by using WSPSelect, or they can be registered for asynchronous delivery by calling WSPAsyncSelect or WSPEventSelect.

The *select* Model

- The *select* model is the most widely available I/O model in Winsock. *Select()* function is used to manage I/O
- The *select* model was incorporated into Winsock 1.1 to allow applications that want to avoid blocking on socket calls the ability to manage multiple sockets in an organized manner.
- The *select* function blocks for I/O operations until the conditions specified as parameters are met.

For example, if you want to find out when it is safe to read data from a socket without blocking, simply assign your socket to the fd_read set using the FD_SET macro and then call select()

```
int select(
      int nfds,
      fd_set FAR * readfds,
      fd_set FAR * writefds,
      fd_set FAR * exceptfds,
      const struct timeval FAR * timeout
);
```

The *select* Model

- It is required to set up either one or all of the read, write, and exception *fd_set* structures by assigning socket handles to a set before setting select() for socket monitor.
- Winsock provides the following set of macros to manipulate and check the *fd_set* sets for I/O activity.
  - *FD_CLR(s, *set)* Removes socket *s* from *set*
  - *FD_ISSET(s, *set)* Checks to see whether *s* is a member of *set* and returns *TRUE* if so
  - *FD_SET(s, *set)* Adds socket *s* to *set*
  - *FD_ZERO(*set)* Initializes *set* to the empty set

Select function example:
https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5a.html

The *WSAAsyncSelect* Model

- Winsock provides a useful asynchronous I/O model that allows an application to receive Windows message-based notification of network events by using *WSAAsyncSelect* function after creating a socket.
- This model is also used by the Microsoft Foundation Class (MFC) *CSocket* object.

```
WSAAsyncSelect(s, hwnd, WM_SOCKET,
      FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

- To use the *WSAAsyncSelect* model, your application must first create a window using the *CreateWindow* function and supply a window procedure (*winproc*) support function for this window
- Once window infrastructure is setup, we can begin creating sockets and turning on window message notification by calling the W*SAAsyncSelect* function

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

hWnd: window handle identifying the window,
wMsg: identifies the message to be received when a network event occurs
lEvent: bitmask that specifies a combination of network events

The *WSAAsyncSelect* Model

- WSAAsyncSelect() allows interest to be declared in the following conditions for a particular socket:
  - Socket readiness for reading
  - Socket readiness for writing
  - Out-of-band data ready for reading
  - Socket readiness for accepting incoming connection
  - Completion of non-blocking connect()
  - Connection closure
  - https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5b.html

The *WSAAsyncSelect* Model

- WSAAsyncSelect(s, hwnd, WM_SOCKET, FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
- This allows our application to get connect, send, receive, and socket-closure network event notifications on socket *s*.
- When your application calls *WSAAsyncSelect* on a socket, the socket mode is automatically changed from blocking to the nonblocking mode
- Program example: https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5b.html

The *WSAEventSelect* Model

- Winsock provides another useful asynchronous event notification I/O model that is similar to the WSAAsyncSelect model that allows an application to receive event-based notification of network events on one or more sockets.
- The major difference with this model compared to WSAAsyncSelect is that network events are notified via an event object handle instead of a window procedure.
- The event notification model requires your application to create an event object for each socket used by calling the WSACreateEvent() function, which is defined as: *WSAEVENT WSACreateEvent(void);*
- The WSACreateEvent() function simply returns a manual reset event object handle. Once you have an event object handle, you have to associate it with a socket and register the network event types of interest

The WSAEventSelect Model

int WSAEventSelect(SOCKET s, WSAEVENT hEventObject,
long lNetworkEvents);

- *hEventObject* parameter represents the event object
- *lNetworkEvents*, represents a bitmask that specifies a combination of network event types that the application is interested in.

- WSAEventSelect() has two operating states and two operating modes. States are known as signaled and non-signaled. The operating modes are known as manual reset and auto reset.
- WSACreateEvent() initially creates event handles in a non-signaled operating state with a manual reset operating mode as network events trigger an event object associated with a socket, the operating state changes from non-signaled to signaled.

Winsock Overlapped IO

- Windows Sockets 2 introduces overlapped I/O and requires that all transport providers support this capability.
- Overlapped I/O can be performed only on sockets that were created through the WSPSocket function with the WSA_FLAG_OVERLAPPED flag set, and follow the model established in Windows.
- For receiving, a client uses WSPRecv or WSPRecvFrom to supply buffers into which data is to be received.
- If one or more buffers are posted prior to the time when data has been received by the network, it is possible that data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur.

Winsock Overlapped IO

- Overlapped send and receive calls return immediately. A return value of zero indicates that the I/O operation completed immediately and that the corresponding completion indication has already occurred.
- A return value of SOCKET_ERROR coupled with an error code of WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that a subsequent indication will be provided when send buffers have been consumed or when receive buffers are filled.
- Both send and receive operations can be overlapped. The receive functions may be invoked multiple times to post receive buffers in preparation for incoming data, and the send functions may be invoked multiple times to queue up multiple buffers to be sent.

The Overlapped Model

- The overlapped I/O model in Winsock offers applications better system performance than any of the I/O models explained so far.
- The overlapped model's basic design allows your application to post one or more asynchronous I/O requests at a time using an overlapped data structure.
- The model's overall design is based on the Windows overlapped I/O mechanisms available for performing I/O operations on devices using the ReadFile() and WriteFile() functions.
- Since the release of Winsock 2, overlapped I/O has been incorporated into new Winsock functions, such as WSASend() and WSARecv().

The Overlapped Model

- To use the overlapped I/O model on a socket, you must first create a socket that has the overlapped flag set.
- After you successfully create a socket and bind it to a local interface, overlapped I/O operations can commence by calling the Winsock functions listed below and specifying an optional WSAOVERLAPPED structure.
  - WSASend(), WSASendTo(), WSARecv(), WSARecvFrom(), WSAIoctl(), WSARecvMsg(), AcceptEx(), ConnectEx(), TransmitFile(), TransmitPackets(), DisconnectEx(), WSANSPIoctl()
- The event notification method of overlapped I/O requires associating windows event objects with WSAOVERLAPPED structures.
- When I/O calls such as WSASend() and WSARecv() are made using a WSAOVERLAPPED structure, they return immediately.

The Overlapped Model

- The WSAOVERLAPPED structure provides the communication medium between the initiation of an overlapped I/O request and its subsequent completion, and is defined as:

- typedef struct WSAOVERLAPPED
  {DWORD Internal; DWORD InternalHigh; DWORD Offset;
  DWORD OffsetHigh; WSAEVENT hEvent;
  } WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;

- The Internal, InternalHigh, Offset, and OffsetHigh fields are all used internally by the system and an application should not manipulate or directly use them.

Prgoram example:
https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5e.html

The Completion Port Model

- I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system.
- When a process creates an I/O completion port, the system creates an associated queue object for requests whose sole purpose is to service these requests.
- This model is well suited to handling hundreds or thousands of sockets. a completion port is actually a Windows I/O construct that is capable of accepting more than just socket handles.
- The completion port model requires you to create a Windows completion port object that will manage overlapped I/O requests using a specified number of threads to service the completed overlapped I/O requests.

The Completion Port Model

- The CreateIoCompletionPort function creates an I/O completion port and associates one or more file handles with that port.
- HANDLE CreateIoCompletionPort(
  HANDLE FileHandle, HANDLE ExistingCompletionPort, DWORD CompletionKey, DWORD NumberOfConcurrentThreads);
- The following code creates an I/O completion port.

CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

- Example: https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports
- https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancediomethod5i.html

More on winsock

- Echo Server/Client code available at the Computation website:
  - http://www.es.ele.tue.nl/~heco/courses/Computation/index.html
  - download **`echoc.cpp`** and **`echod.cpp`**

- Some WinSock tutorials on the Internet:
  - http://tangentsoft.net/wskfaq/
  - http://www.linuxhowtos.org/C_C++/socket.htm
  - http://www.madwizard.org/programming/tutorials/netcpp/3
  - http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#theory
  - http://www.win32developer.com/tutorial/winsock/winsock_tutorial_1.shtm

Socekt extension: setup and cleanup functions

- The WinSock specification provides a number of extensions to the standard set of Berkeley Sockets routines.
- Extended functions allow message or function-based, asynchronous access to network events, as well as enable overlapped I/O

Socekt extensions

accept() *	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind()	Assign a local name to an unnamed socket.
closesocket() *	Remove a socket from the per-process object reference table. Only blocks if SO_LINGER is set with a non-zero timeout on a blocking socket.
connect() *	Initiate a connection on the specified socket.
getpeername()	Retrieve the name of the peer connected to the specified socket.
getsockname()	Retrieve the local address to which the specified socket is bound.
getsockopt()	Retrieve options associated with the specified socket.
htonl() ∞	Convert a 32-bit quantity from host byte order to network byte order.
htons() ∞	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr() ∞	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa() ∞	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket()	Provide control for sockets.
listen()	Listen for incoming connections on a specified socket.
ntohl() ∞	Convert a 32-bit quantity from network byte order to host byte order.
ntohs() ∞	Convert a 16-bit quantity from network byte order to host byte order.
recv() *	Receive data from a connected or unconnected socket.
recvfrom() *	Receive data from either a connected or unconnected socket.
select() *	Perform synchronous I/O multiplexing.
send() *	Send data to a connected socket.
sendto() *	Send data to either a connected or unconnected socket.
setsockopt()	Store options associated with the specified socket.
shutdown()	Shut down part of a full-duplex connection.
socket()	Create an endpoint for communication and return a socket descriptor.

Socekt extension: setup and cleanup functions

WSAAccept()*	An extended version of **accept()** which allows for conditional acceptance and socket grouping.
WSAAsyncGetHostByAddr()**	A set of functions which provide asynchronous versions of the standard Berkeley **getXbyY()** functions. For example, the **WSAAsyncGetHostByName()** function provides an asynchronous message based implementation of the standard Berkeley **gethostbyname()** function.
WSAAsyncGetHostByName()**	
WSAAsyncGetProtoByName()**	
WSAAsyncGetProtoByNumber()***	
WSAAsyncGetServByName()**	
WSAAsyncGetServByPort()**	
WSAAsyncSelect()**	Perform asynchronous version of **select()**
WSACancelAsyncRequest()**	Cancel an outstanding instance of a **WSAAsyncGetXByY()** function.
WSACleanup()	Sign off from the underlying WinSock DLL.
WSACloseEvent()	Destroys an event object.
WSAConnect()*	An extended version of **connect()** which allows for exchange of connect data and QOS specification.
WSACreateEvent()	Creates an event object.
WSADuplicateSocket()	Allow an underlying socket to be shared by creating a virtual socket.
WSAEnumNetworkEvents()	Discover occurrences of network events.
WSAEnumProtocols()	Retrieve information about each available protocol.

Assignment:
Explain the functionality of those functions with basic syntax.

Socekt extension: setup and cleanup functions

WSAEventSelect()	Associate network events with an event object.
WSAGetLastError()**	Obtain details of last WinSock error
WSAGetOverlappedResult()	Get completion status of overlapped operation.
WSAGetQOSByName()	Supply QOS parameters based on a well-known service name.
WSAHtonl()	Extended version of **htonl()**
WSAHtons()	Extended version of **htons()**
WSAIoctl()*	Overlapped-capable version of **ioctl()**
WSAJoinLeaf()*	Add a multipoint leaf to a multipoint session
WSANtohl()	Extended version of **ntohl()**
WSANtohs()	Extended version of **ntohs()**
WSAProviderConfigChange()	Receive notifications of service providers being installed/removed.
WSARecv()*	An extended version of **recv()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvFrom()*	An extended version of **recvfrom()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent()	Resets an event object.
WSASend()*	An extended version of **send()** which accommodates scatter/gather I/O and overlapped sockets
WSASendTo()*	An extended version of **sendto()** which accommodates scatter/gather I/O and overlapped sockets
WSASetEvent()	Sets an event object.
WSASetLastError()**	Set the error to be returned by a subsequent **WSAGetLastError()**

Homework:
Explain the functionality of those functions with basic syntax.

Socekt extension: setup and cleanup functions

WSASocket()	An extended version of **socket()** which takes a WSAPROTOCOL_INFO struct as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
WSAStartup()**	Initialize the underlying WinSock DLL.
WSAWaitForMultipleEvents()*	Blocks on multiple event objects.

* = The routine can block if acting on a blocking socket.

$^{\infty}$ = The routine is realized by queries to name space providers that supports AF_INET, if any

** = The routine was originally a WinSock 1.1 function.

Socket close

- int closesocket([in] SOCKET s);
- [in] s >> a descriptor identifying the socket to close.
- If no error occurs, **closesocket** returns zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Winsock Database Functions

- **gethostname()**
- **gethostbyaddr()**
- **gethostbyname()**
- **getprotobyname()**
- **getprotobynumber()**
- **getservbyname()**
- **getservbyport()**

Asynchronous versions of these functions were also defined:

- **WSAAsyncGetHostByAddr()**
- **WSAAsyncGetHostByName()**
- **WSAAsyncGetProtoByName()**
- **WSAAsyncGetProtoByNumber()**
- **WSAAsyncGetServByName()**
- **WSAAsyncGetSetvByPort()**

There are also two functions (now implemented in the WinSock 2 DLL) used to convert dotted IPv4 internet address notation to and from string and binary representations, respectively:

- **inet_addr()**
- **inet_ntoa()**

Berkeley Style Functions

SPI function name	Description
GETXBYYSP_gethostbyaddr	Supplies a hostent structure for the specified host address.
GETXBYYSP_gethostbyname	Supplies a hostent structure for the specified host name.
GETXBYYSP_getprotobyname	Supplies a protoent structure for the specified protocol name.
GETXBYYSP_getprotobynumber	Supplies a protoent structure for the specified protocol number.
GETXBYYSP_getservbyname	Supplies a servent structure for the specified service name
GETXBYYSP_getservbyport	Supplies a servent structure for the service at the specified port.
GETXBYYSP_gethostname	Returns the standard host name for the local machine.

Table 1

Async Style Functions

SPI function name	Description
GETXBYYSP_WSAAsyncGetHostByAddr	Supplies a hostent structure for the specified host address.
GETXBYYSP_WSAAsyncGetHostByName	Supplies a hostent structure for the specified host name.
GETXBYYSP_WSAAsyncGetProtoByName	Supplies a protoent structure for the specified protocol name.
GETXBYYSP_WSAAsyncGetProtoByNumber	Supplies a protoent structure for the specified protocol number.
GETXBYYSP_WSAAsyncGetServByName	Supplies a servent structure for the specified service name.
GETXBYYSP_WSAAsyncGetServByPort	Supplies a servent structure for the service at the specified port.
GETXBYYSP_WSACancelAsyncRequest	Cancels an asynchronous GetXbyY() operation.

Error handling functions

- WSAGetLastError() is a wrapper for GetLastError(), which is a standard function for reporting errors in Windows
- WSASetLastError() to set a virtual error that your application can retrieve with a call to WSAGetLastError().
- WSAGetAsyncError() and WSAGetSelectrror() are functions that extract additional error information whenever an error occurs.
- Use WSAGetAsyncError() and WSAGetSelectError() rather than WSAGetLastError() when you use Microsoft's family of asynchronous functions

Send/receive on server/client over connection

Server

```
#define DEFAULT_BUFLEN 512

char recvbuf[DEFAULT_BUFLEN];
int iResult, iSendResult;
int recvbuflen = DEFAULT_BUFLEN;

// Receive until the peer shuts down the connection
do {

    iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        printf("Bytes received: %d\n", iResult);

        // Echo the buffer back to the sender
        iSendResult = send(ClientSocket, recvbuf, iResult, 0);
        if (iSendResult == SOCKET_ERROR) {
            printf("send failed: %d\n", WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
        printf("Bytes sent: %d\n", iSendResult);
    } else if (iResult == 0)
        printf("Connection closing...\n");
    else {
        printf("recv failed: %d\n", WSAGetLastError());
        closesocket(ClientSocket);
        WSACleanup();
        return 1;
    }

} while (iResult > 0);
```

Client

```
#define DEFAULT_BUFLEN 512

int recvbuflen = DEFAULT_BUFLEN;

const char *sendbuf = "this is a test";
char recvbuf[DEFAULT_BUFLEN];

int iResult;

// Send an initial buffer
iResult = send(ConnectSocket, sendbuf, (int) strlen(sendbuf), 0);
if (iResult == SOCKET_ERROR) {
    printf("send failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

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

// shutdown the connection for sending since no more data will be sent
// the client can still use the ConnectSocket for receiving data
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

// Receive data until the server closes the connection
do {
    iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0)
        printf("Bytes received: %d\n", iResult);
    else if (iResult == 0)
        printf("Connection closed\n");
    else
        printf("recv failed: %d\n", WSAGetLastError());
} while (iResult > 0);
```

IOCTL Socekt function

- The **ioctlsocket** function controls the I/O mode of a socket.
- int ioctlsocket(

  [in] SOCKET s,

  [in] long cmd,

  [in, out] u_long *argp

  );
- [in] s : A descriptor identifying a socket.
- [in] cmd: A command to perform on the socket *s*.
- [in, out] argp: A pointer to a parameter for *cmd*.

IOCTL Socekt function

- Upon successful completion, the **ioctlsocket** returns zero, Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Error code	Meaning
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEFAULT	The *argp* parameter is not a valid part of the user address space.

IOCTL Socekt Example

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

#pragma comment(lib, "Ws2_32.lib")

void main()
{
//-----------------------
// Initialize Winsock
WSADATA wsaData;
int iResult;
u_long iMode = 0;

iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != NO_ERROR)
  printf("Error at WSAStartup()\n");
//----------------------
// Create a SOCKET object.
SOCKET m_socket;
m_socket = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
if (m_socket == INVALID_SOCKET) {
  printf("Error at socket(): %ld\n", WSAGetLastError());
  WSACleanup();
  return;
}
//-----------------------
// Set the socket I/O mode: In this case FIONBIO
// enables or disables the blocking mode for the
// socket based on the numerical value of iMode.
// If iMode = 0, blocking is enabled;
// If iMode != 0, non-blocking mode is enabled.

iResult = ioctlsocket(m_socket, FIONBIO, &iMode);
if (iResult != NO_ERROR)
  printf("ioctlsocket failed with error: %ld\n", iResult);
  }
```

Notes useful

- Bees Guide
  - http://beej.us/guide/bgnet/
  - http://beej.us/guide/bgnet/pdf/bgnet_usl_c_1.pdf
  - For example code: http://beej.us/guide/bgnet/examples/

Queries?

Thank YOU!!