Walkthrough 2: TCP CONNECTIONS

Objectives:

The intention of this walkthrough is to introduce students to the WIN SOCKETS API and implement a TCP Connection from one lab machine to another.

As you write the code given to you add comments to each line explaining what is happening. Use the Green explanation boxes to assist you but put the comments in your own words.

You will need two computers so you will need to work in pairs or log into two computers and copy your code between them. (one drive may help make this easy)

Getting Started:

1. Open up the finished Product from the first walkthrough where we outlined a singleton Network Manager inside an SDL Pong Game.
2. Ensure it still runs.

Adding WINSOCK API:

To add the windows Sockets API to our application we will first, add the following defines and headers inside of our NetworkManager.h at the very top:

```
1  #define WIN32_LEAN_AND_MEAN
2  #pragma comment (lib, "ws2_32.lib")
3
4  #include <Windows.h>
5  #include <WinSock2.h>
```

Breakdown of code:

- The first line says to make sure when we include windows.h below it doesn't include everything and bloat our application. Windows.h is huge…
- The second line is a way of defining the linking of the library. Use library ws2_32.lib, alternatively it could be added in the project settings but this makes our application portable to other OS's
- Windows.h is needed for some of the functions we will use with our sockets and Winsock2.h is our main API we will be working with.

Setting up the API:

Before we dive in and start transmitting packets of data between our PC's we will need to do some setup of the API.

1. To begin with, make sure to include our iostream and tell it we are using namespace std at the top of your NetworkManager.cpp.
2. Remove the {} from the header files Init function as we will now give it functionality in the cpp file.
3. Flesh out the Init function with the following code:

```
17  void NetworkManager::Init()
18  {
19      WSADATA wsaInfo;
20
21      // Initialize Winsock using version 2.2
22      int errCode = WSAStartup(MAKEWORD(2, 2), &wsaInfo);
23
24      if (errCode != 0)
25      {
26          cout << "WSAStartup failed with error: " << errCode;
27          return;
28      }
29  }
```

Breakdown of code:

- The first creates a WSADATA object which will be populated when we call the next function.
- The second line of code here calls WSAStartup supplying it two params:
    - First the version (2.2) of WSA we want to use.
    - The second parameter is our WSADATA object we created on the first line.
- Lastly we check to ensure there were no problems initializing the WSA version 2.2 API comparing the number returned from the function call.

Cleaning Up:

When we call WSAStartup() above, it allocated memory for the sockets API and gets everything ready. We need to ensure we let the API know when we are finished with it.

```
31  void NetworkManager::Shutdown()
32  {
33      cout << "about to shutdown" << endl;
34      cout << WSAGetLastError() << endl;
35
36      WSACleanup();
37      system("pause");
38      exit(0);
39  }
```

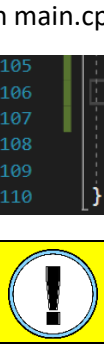
Breakdown of code:

- The first couple of lines are just print outs which can be used for debugging later.
- WSACleanup() cleans up any allocations and setup made by the API to ensure the next time you attempt to run the program it doesn't think everything is already in use.
- Exit(0) is a function used to exit our application entirely without having to return from the main loop.

Connecting in Main.cpp:

Let's put our two new functions to use inside our main.cpp now, our Init() function should already be in place from Walkthrough 1. We need to add our shutdown function call at the end of the main function in main.cpp:

```
105     SDL_Quit();
106
107     NetworkManager::GetInstance()->Shutdown();
108
109     return true;
110 }
```



Before we start on implementing our connection, take a moment to run your code now and ensure there are no errors.

Sockets Setup:

We will now dive into creating and setting up our Sockets which will be our means of communication from one computer to another.

1. The first thing we will need to do is add two new SOCKET variables to our NetworkManager.h under the private area:

```
private:
    NetworkManager();
    ~NetworkManager();

    SOCKET listenSocket;
    SOCKET peerSocket;
```

2. Also inside the NetworkManager.h we can create a new void function called SetupSockets().
3. We now must add the SetupSockets definition to our NetworkManager.cpp file
    a. Ideally this should be located between the init and shutdown functions. Try to keep the shutdown function as the last function defined.
4. Finally we should add a call to SetupSockets as the last function called within our Init() function.

```
void NetworkManager::SetupSockets()
{
    listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenSocket == INVALID_SOCKET)
    {
        cout << "Failed to create tcp socket." << endl;
        Shutdown();
    }

    peerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (peerSocket == INVALID_SOCKET)
    {
        cout << "Failed to create tcp socket." << endl;
        Shutdown();
    }
}
```

Breakdown of code:

- Our SOCKET objects will be used to keep track of all of our connections data. We will need to use this across multiple functions so it's important to keep a local copy of it within our manager.
- Inside SetupSockets our first line creates our socket, the socket() constructor takes in 3 params:
    - The first one indicates that it is a IPV4 connection
    - The second and third are used to indicate we are using a TCP Connection.
- Lastly, we check to ensure our socket was able to be created and error out if it failed.
- We do this twice as we will need one socket if we are the server to listen for players and the other to actually communicate with each other.

Sockets Configuration, Binding & Listening [SERVER]:

We will now configure our sockets with the information needed to communicate and send packets of data. After configuring the sockets we will be binding it. This means that we are reserving the socket for our use and that no other application can use it while we have it bound. This will primarily be used for our Server Player.

1. Begin by adding a void BindSocket() function inside your header for the NetworkManager.h
    - Ensure this is public.
2. We will also need to add the following **private** objects to the header:

```
SOCKADDR_IN listenAddr;
SOCKADDR_IN peerAddr;
```

3. Finally add in the following functionality to our BindSockets() function inside our NetworkManager.cpp file (again, before the Shutdown() function).

```
// SERVER ONLY
void NetworkManager::BindSockets()
{
    listenAddr.sin_family = AF_INET;
    listenAddr.sin_port = htons(8890);
    listenAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(listenSocket, reinterpret_cast<SOCKADDR *>(&listenAddr), sizeof(listenAddr)) == SOCKET_ERROR)
    {
        cout << "[ERROR] TCP bind Error: " << WSAGetLastError() << endl;
        Shutdown();
    }

    listen(listenSocket, SOMAXCONN);
}
```

Breakdown of code:

- The first 3 lines here are used to setup our Address struct to describe the way we will listen for connections
    - AF_INET means IPv4
    - Line 2 sets the port number to 8890 and converts it from host to network order for a short.
    - Line 3 sets the address to accept any address in and again converts it to network byte order for a long.
- The bind function here establishes the socket with our OS so all traffic gets forwarded in to this application.
- The bind takes in 3 params:
    - Our socket to listen on previously created.
    - The address information to listen to we set in the first 3 lines
    - The size of memory to read in for this information.
- We wrap the bind function inside an if check, checking for an error binding the socket.
- Lastly, we tell the game we are open for business and start listening on the socket.
    - The params here are the socket we are listening on, and the max number of connections to queue

Shutdown Modifications:

We need to unbind our sockets inside the shutdown to clean up after ourselves.

Add the following two new if statements to our shutdown function:

```
void NetworkManager::Shutdown()
{
    cout << "about to shutdown" << endl;
    cout << WSAGetLastError() << endl;

    if (peerSocket != INVALID_SOCKET)
    {
        if (closesocket(peerSocket) != 0)
        {
            cout << "[ERROR] error closing TCP socket." << endl;
        }
    }

    if (listenSocket != INVALID_SOCKET)
    {
        if (closesocket(listenSocket) != 0)
        {
            cout << "[ERROR] error closing TCP socket." << endl;
        }
    }

    WSACleanup();
    system("pause");
    exit(0);
}
```

Breakdown of code:

- In each of the two if statements we check to ensure we successfully created the sockets.
    - If so we call closesocket with the socket supplied and check for errors closing the socket.

Connecting Sockets [CLIENT]:

We now have sockets bound and the Server listening for client peers. Here we start to have branching functions. We will add them all to the manager, but only the client will make client calls and the server will make server calls.

For now, we will need to add a function so that clients can connect to the listening server.

You will need to ensure you **#include <WS2tcpip.h>** at the top of the NetworkManager.cpp file.

Add the following ConnectSockets function to your header and the contents as defined here:

```
//CLIENTS ONLY
void NetworkManager::ConnectSockets()
{
    listenAddr.sin_family = AF_INET;
    listenAddr.sin_port = htons(8890);
    inet_pton(AF_INET, "10.105.156.53", &listenAddr.sin_addr);

    if (connect(peerSocket, reinterpret_cast<SOCKADDR *>(&listenAddr), sizeof(listenAddr)) == SOCKET_ERROR)
    {
        cout << "[ERROR] connect Error: " << WSAGetLastError() << endl;

        Shutdown();
    }

    unsigned long b = 1;
    ioctlsocket(peerSocket, FIONBIO, &b);
}
```

Change the IP here to your friends PC's IP.

Breakdown of code:

- The first two lines setup our address information of the computer we want to communicate with.
- htons converts our port to network byte order for a short
- inet_pton is used to convert an IP address from a string to binary data.
    - We then assign it into our address data's address.
    - This is included by the **WS2tcpip.h** inclusion**.**

- The if statement calls the connect() API function which is supplied our peerSocket to establish our connection.
    - We also pass in a reference to the information about the connection supplied earlier and the size.
    - A check is made to ensure it succeeded in connecting to the server

- The last function is used to ensure that we are proceeding in a non-blocking mode so we can send and receive messages from the server Asynchronously.
    - unsigned long b = 1; // 0 = blocking mode, 1 = non blocking mode.
    - input output control socket
    - FIONBIO – set io to non blocking io

Accepting Sockets [SERVER]:

We now have sockets bound and the Server listening for client peers. Clients will attempt to connect to the server and the server will need to accept the incoming requests to connect.

Add the following function to our NetworkManagers .h and cpp file:

```
// SERVER ONLY
void NetworkManager::AcceptConnections()
{
    int clientSize = sizeof(peerAddr);

    peerSocket = accept(listenSocket, reinterpret_cast<SOCKADDR *>(&peerAddr), &clientSize);
    if (peerSocket != INVALID_SOCKET)
    {
        char ipConnected[32];
        inet_ntop(AF_INET, &peerAddr.sin_addr, ipConnected, 32);
        // print out who connected
        cout << ipConnected << " Just connected into the server" << endl;
    }

    unsigned long b = 1;
    ioctlsocket(listenSocket, FIONBIO, &b);
}
```

Breakdown of code:

- Our first line just sets an initial value to an integer which we will use to get the size of the client information connecting.
- Our second line is our main functionality calling the accept socket API call.
    - This function returns the socket connection to the client that connected to the server.
    - **NOTE: this is a different socket than the one listening.**
    - This function will also populate the peerAddr with all the information about the peers connection.
- The next line is a simple if statement that checks the connection was made properly.
- Insnide the if statement we retrieve the ip address and convert it back from binary data to a string using inet_ntop().
    - We pass in AF_INET = IPv4, the address we grabbed from the accept call, a char array to store the string into and the size of the char array = 32.
- The last function like before allows our program to communicate asynchronously moving forward for send and receive calls. See connect call for more details.

Sending Data [BOTH]:

At this point we have the ability for clients to connect to a server listening on a specific port. We can now proceed to send data from the client to the server or the server to the client using the following function. Add a prototype to your header file and the following contents to your cpp for the NetworkManager.

```
// BOTH
void NetworkManager::SendData(const char* data)
{
    int messageLen = MAX_MESSAGE_SIZE;

    if (send(peerSocket, data, messageLen, 0) == SOCKET_ERROR)
    {
        int error = WSAGetLastError();

        if (error != WSAEWOULDBLOCK)
        {
            cout << "[Error] send error: " << WSAGetLastError() << endl;
            Shutdown();
        }
    }
}
```

Breakdown of code:

- This function takes in a char* which means we can supply it something like “Hello World”.
- The if statement calls the send API call passing in our peerSocket we previously established, sending the data passed into the function and telling it how large the data is that we are transmitting.
    - The final 0 parameter is a flag for now we will not use it so it can stay 0. See more details for options here: https://linux.die.net/man/2/sendto
    - We check this send call for a socket error like usual and shutdown / log the error if it occurred.

Receiving Data [BOTH]:

If we are going to send data we also need to be able to receive the sent data. Both the client and server will need to receive data from each other so both will use the following function. Add it to both the header and cpp file of the NetworkManager.

```
// BOTH
char* NetworkManager::ReceiveData()
{
    int ByteReceived = 0;
    char* ReceiveBuf = new char[MAX_MESSAGE_SIZE];
    ReceiveBuf[0] = '\0';

    ByteReceived = recv(peerSocket, ReceiveBuf, MAX_MESSAGE_SIZE, 0);

    if (ByteReceived == SOCKET_ERROR)
    {
        int error = WSAGetLastError();
        if (error != WSAEWOULDBLOCK)
        {
            cout << "[Error] receive error: " << WSAGetLastError() << endl;
            Shutdown();
        }
    }
    else if (ByteReceived > 0)
    {
        ReceiveBuf[ByteReceived] = '\0';
        cout << "TCP received Data: " << ReceiveBuf << endl;
    }

    return ReceiveBuf;
}
```

Breakdown of code:

- The first few lines are setting up temporary variables to store the data we receive.
- The fourth line we call recv (receive) which is passed in the socket to receive from, the buffer to store the string retrieved, the size to receive, and the same 0 flag used for send.
- The recv function will return -1 if there was an error reading.
    - SOCKET_ERROR = -1
- We handle the error if the data value returned was -1.
- If there was no error and no data it would just return 0
- In the final else we check for greater than 0 bytes which means we received a message.
- We print out the contents of the data received and return it back to the main function.

Using our Sockets [SERVER]:

Here's where you and your partner will have to work together and have a slightly different codebase. One of you will need to be the server and one will need to be the client.

1. Immediately after we make the call to Init the NetworkManager from main.cpp you will need to call BindSockets();
2. Right before you start your main game loop add the call to AcceptConnections().
    - This call is presently blocking, meaning the game won't continue until a connection is made.

Using our Sockets [CLIENT]:

On the client side, we will need to make calls to Init() and ConnectSockets()

1. Right before you start your main game loop add the call to ConnectSockets().

Using our Sockets [BOTH]:

1. You can add the following code to your main game loop to communicate after you have connected the client to the server:

```
NetworkManager::GetInstance()->SendData("Hello world");
char* DataFromOpponent = NetworkManager::GetInstance()->ReceiveData();
std::cout << DataFromOpponent;
```

Demo your finished code to the instructor for signoff.