Lesson 2 - Socket Programming
Building a Simple Server

Summary

We are going to use the functions described in the first lesson to build a very simple server program.

The Server Code

Firstly we need to include the correct header files to use the Sockets library:

```
1  #include <stdio.h>
2  #include <winsock2.h>
3  #include <ws2tcpip.h>
4
5  #define HIGHVERSION 2
6  #define LOWVERSION  2
7  #define HOST        "127.0.0.1"
8  #define PORT        "4376"
9  #define FAMILY      AF_UNSPEC
10 #define TYPE        SOCK_STREAM
11 #define FLAGS       AI_PASSIVE
12 #define PROTOCOL    IPPROTO_TCP
13 #define BACKLOG     10
14 #define BUFFSIZE    100
15
16 WSAData wsaData;
17 struct addrinfo *addr;
```

Header files and definitions

The definitions will make more sense as we continue with the code. The two data structures will store important details for the Sockets library and the server address.

Initialising the Sockets Library

The first we need to do when creating either a server or client program to initialise the Windows Sockets library. The `init()` function below does this and stores the results in our WSAData struct `wsaData`.

```
1  int init()
2  {
3     // call to startup
4     int error = WSAStartup(MAKEWORD(HIGHVERSION,LOWVERSION), &wsaData);
5     if (error != 0)
6     {
7        fprintf(stderr, "WSAStartup failed with error: %d.\n", error);
8        return 1;
9     }
10
11    // check version values
```

```
12    if (LOBYTE(wsaData.wVersion) != LOWVERSION ||
13        HIBYTE(wsaData.wVersion) != HIGHVERSION)
14    {
15        printf("The version requested cannot be supported.\n");
16        printf("Version set is %d.%d\n", LOBYTE(wsaData.wVersion),
17            HIBYTE(wsaData.wVersion));
18        WSACleanup();
19        return 1;
20    }
21    else
22    {
23        printf("The Winsock API has been successfully initialised.\n"
24              "You are using version %d.%d.\n\n",
25              HIBYTE(wsaData.wVersion),
26                 LOBYTE(wsaData.wVersion));
27        return 0;
28    }
29 }
```

Initialising the Sockets library

On line 4, we make our call to `WSAStartup()`. We are using version 2.2 of the Windows Sockets library here. Both `HIGHVERSION` and `LOWVERSION` are used to signify this. It may be that the platform you are developing on does not have the version of the library that you have requested. On line 12 and 13 we test to make sure that the version that has been set in `wsaData` has been set correctly.

Addressing Information

Once the Sockets library has been initialised we need to sort out the addressing structs so a socket can be created.

```
1  int addressing()
2  {
3     int result;
4     struct addrinfo hints;
5     struct addrinfo *temp;
6
7     // initialise the hints struct
8     memset(&hints, 0, sizeof(hints));
9     hints.ai_family = FAMILY;
10    hints.ai_socktype = TYPE;
11    hints.ai_flags = FLAGS;
12    hints.ai_protocol = PROTOCOL;
13
14    result = getaddrinfo(HOST, PORT, &hints, &addr);
15
16    if (result != 0)
17    {
18        printf("getaddrinfo() failed with error: \n%d: %s\n", result,
19            gai_strerror(WSAGetLastError()));
20        WSACleanup();
21        return 1;
22    }
23
24    // need to walk the linked list
25    printf("Addressing information:\n");
26
27    int i = 0;
28    for (temp = addr; temp != NULL; temp = temp->ai_next)
29    {
```

```
30          printf("\nEntry %d:\n", ++i);
31          switch (temp->ai_family)
32          {
33              case AF_INET:
34                  printf("\t Address family: AF_INET\n");
35                  break;
36              case AF_INET6:
37                  printf("\t Address family: AF_INET6\n");
38                  break;
39          }
40          switch (temp->ai_protocol)
41          {
42              case IPPROTO_TCP:
43                  printf("\t Protocol: TCP\n");
44                  break;
45              case IPPROTO_UDP:
46                  printf("\t Protocol: UDP\n");
47                  break;
48          }
49          switch (temp->ai_socktype)
50          {
51              case SOCK_STREAM:
52                  printf("\t Socket type: Stream\n");
53                  break;
54              case SOCK_DGRAM:
55                  printf("\t Socket type: Datagram\n");
56                  break;
57          }
58      }
59
60      return 0;
61  }
```

Creating the addressing information

We first declare the `hints` struct and initialise the fields we need. You must use `memset()` (or `ZeroMemory()`) to clear the struct as it will be pointing to unknown memory locations initially. Our call to `getaddrinfo()` on line 14 uses the structs and definitions we have already created. The host name in this example is localhost as we want to run both client and server locally on the same machine. If successful, the function will return zero. From line 28 onwards we simply walk the linked list and print out the information contained.

Running our server

Using the above two functions we can start to initialise our server.

```
1   int main(void)
2   {
3       SOCKET s = NULL;
4
5       // initialise socket library
6       if (init())
7       {
8           printf("Unable to initialise the Winsock library\n");
9           exit(1);
10      }
11
12      // intiailise addressing information
13      if (addressing() != 0)
14      {
```

```
15          printf("Uanble to initialise addressing information\n");
16          exit(1);
17      }
18
19      // create a socket for the server to listen on
20      if ((s = socket(addr->ai_family, addr->ai_socktype,
21              addr->ai_protocol)) == INVALID_SOCKET)
22      {
23          printf("Unable to create a socket\n");
24          printf("Failed with error: %d\n%s\n", WSAGetLastError(),
25              gai_strerror(WSAGetLastError()));
26          exit(1);
27      }
28      else
29      {
30          printf("\nSocket created successfully.\n");
31      }
32
33      // bind to the socket created above
34      if (bind(s, addr->ai_addr, addr->ai_addrlen) != 0)
35      {
36          printf("Unable to bind to socket\n");
37          printf("Failed with error: %d\n%s\n", WSAGetLastError(),
38              gai_strerror(WSAGetLastError()));
39      }
40      else
41      {
42          printf("Bound to socket.\n");
43      }
44
45      // finished with addrinfo struct now
46      freeaddrinfo(addr);
47
48      // listen on the socket
49      if (listen(s, BACKLOG) != 0)
50      {
51          printf("Unable to listen on socket\n");
52          printf("Failed with error: %d\n%s\n", WSAGetLastError(),
53              ai_strerror(WSAGetLastError()));
54      }
55      else
56      {
57          printf("Listening on the socket.\n");
58      }
59
60      // continually accept new connections
61      while(1)
62      {
63          printf("\nWaiting for connections ...\n");
64
65          SOCKET inc = NULL;
66          struct sockaddr_storage inc_addr;
67          socklen_t inc_size = sizeof (inc_addr);
68
69          // accept new connection from a client
70          if ((inc = accept(s, (struct sockaddr *) &inc_addr,
71                  &inc_size)) == INVALID_SOCKET)
72          {
```

```
73            printf("Unable to accept new connection\n");
74            printf("Failed with error: %d\n%s\n", WSAGetLastError(),
75               gai_strerror(WSAGetLastError()));
76        }
77        else
78        {
79            printf("Accepted a new connection ...\n");
80        }
81
82        // send message to the client
83        char* hw = "Hello Client";
84        send(inc, hw, strlen(hw), 0);
85
86        // receive message from client
87        int bytesreceived;
88        char buff[BUFFSIZE];
89
90        if ((bytesreceived = recv(inc, buff, BUFFSIZE-1, 0)) == -1)
91        {
92            printf("Error receiving\n");
93            printf("Failed with error: %d\n%s\n", WSAGetLastError(),
94               gai_strerror(WSAGetLastError()));
95        }
96        else
97        {
98            buff[bytesreceived] = '\0';
99            printf("Message received. Received %d bytes.
100                \nMessage is: %s\n", bytesreceived,  buff);
101       }
102
103       closesocket(inc);
104   }
105
106   closesocket(s);
107   WSACleanup();
108}
```

Binding

Our listening socket will be `s`, declared on line 3. First we initialise the Sockets library (line 6) and create our addressing information (line 13). Once this is done we then create our listening socket on line 20 and 21. We use the information in the `addrinfo` struct we created by calling `addressing()`. We are using the first element in the linked here as there is only one element. You would typically want to make sure that you are going to be creating the correct socket for the address family you need here.

Once the socket is created, as the server, we need to bind to this. On line 34 we make the call to bind using the socket and additional information in our `addrinfo` struct. We make the call to `freeaddrinfo()` after this as we are finished with the struct and want to reclaim the memory.

Once bound, we want the server to listen for incoming connection requests. On line 49, we make the call to `listen()` using our original socket. We are setting the backlog to ten here but you would want to adjust this depending on the network requirements.

We then enter a loop on line 61 allowing the server to continually listen for new connection requests coming in. Notice that we are making a new Socket on line 65. This is because the call to `accept()` on line 70 returns a new Socket to handle the interaction between client and server. The call to `accept()` takes a `sockaddr_storage` struct which we cast to a `sockaddr` struct. This stores address details about the client that made the connection request.

On line 84 we are sending a message to the client that made the connection request. We use the new socket, `inc` and send the message "Hello Client". Following this we also want to receive the a return message from the client. We declare a buffer on line 88 that will store the incoming message.

On line 90 we make a call to `recv()` to receive the message the client has sent. We need to pass the name of the buffer and the socket in this function call.

We have the size of the buffer to 100 chars and so we can accept any message up to this length. As we are only sending small messages we will never have a problem whereby the message that needs to be received will be larger than the allocated buffer. If the message does happen to be larger than the supplied buffer then an error will be returned and depending on the protocol used the message will be lost (UDP) or will need to be received again with a large enough buffer (TCP). Once done we close the new socket we created for the connection.