Programming for Engineers I
Lab 12
Network Programming

Attique Dawood

July 24, 2014

Last Modified: Thursday 24th July, 2014, 13:34

1 Network Programming Concepts

1.1 IP Address

IP Address is the address of a computer. To send a message to a particular computer we need its IP address.

1.2 Port

Let's say Person A and Person B live in a House. If you want to send a letter to Person A you'll first write the house address and on top of it mention that it is addressed to Person A. Similarly, in a computer, hundreds of processes may be running at a time. To send a message to a particular process you mention **Port** number of process. Two processes cannot have the same port number. So, processes in a computer are distinguished by their port numbers. A port number can be any number in the range 1–65535. Usually port numbers 1–200 are reserved for system processes, e.g., port 80 is for http protocol. Rest of the port numbers can be acquired by any process.

1.3 Socket

Socket is the mail box outside your home. You put letter in it that are taken away by postman and you can take take letters out of it that postman delivers. A process must associate its port number and IP address with a socket in order to communicate with outside world.

1.4 `bind()`

`bind()` is used to bind IP address, port number and socket together. This is necessary before communication can start.

1.5 `send()` and `receive`

`send()` or `sendto()` sends data or message, whereas, `receive()` or `receivefrom()` receives data. `receive`'ing is blocking. A process will wait indefinitely until it receives a message. If two processes are communicating then they must NOT call `receive()` at the same time or they will both block. `send()` and `receive()` calls must alternate between two processes so that if one process is `send`'ing the other is `receive`'ing. It is normally a good practice to ensure that consecutive `send()` or `receive()` calls are never made by a single process. But this isn't necessary if process knows beforehand exactly how many bytes it needs to send or receive.

1.6 TCP and UDP

TCP and UDP are two modes of communication in IP.

- TCP uses `send()` and `receive()` while UDP uses `sendto()` and `receivefrom()`.
- UDP is like posting a letter. Any packet can be sent to or received from any IP address.
- TCP is like a phone call. A connection is established between a Server and Client. Client is the one calling and Server is the one accepting calls.
- Although Client and Server don't exist in UDP technically, the two computers communicating using UDP are commonly distinguished by calling one Server and the other Client. Normally Server is the one that starts in waiting (`receivefrom()`) state.

2 TCP and UDP

TCP and UDP are two modes of communication in IP.

- TCP uses `send()` and `receive()` while UDP uses `sendto()` and `receivefrom()`.
- UDP is like posting a letter. Any packet can be sent to or received from any IP address.
- TCP is like a phone call. A connection is established between a Server and Client. Client is the one calling and Server is the one accepting calls.
- Although Client and Server don't exist in UDP technically, the two computers communicating using UDP are commonly distinguished by calling one Server and the other Client. Normally Server is the one that starts in waiting (`receivefrom()`) state.

3 TCP Model

3.1 Server

- 2 sockets, 1 for accepting connections and another for communication with Client.
- Maintains at least two address structures, one for itself and one to store Client information.
- There can be multiple Clients connected to Server.
- Sequence of Operation:
  1. Create Server `socket()`.
  2. `bind()` Server socket with its address and port.
  3. Set Server socket options, `setsockopt()` (optional).
  4. Configure Server socket to `listen()` to any incoming connection requests.
  5. Set Server socket state to `accept()` any connection request. This is blocking. When a connection is accepted it will return Client socket and Client address is stored.
  6. Use Client socket to `send()` and `recv()` data to that particular Client.
  7. `close()` sockets before exiting.

3.2 Client

- 1 socket to connect and `send()` and `recv()` data.
- Maintains two address structures, for Client and Server.
- Sequence of Operation:
  1. Create Client `socket()`.
  2. `bind()` Client socket with its adress and port.
  3. `connect()` to a Server using Server Address and Server Port.
  4. `send()` and `recv()` using Client socket.
  5. `close()` Client socket before exiting.

4 UDP Model

- Server does not need to `listen()` or `accept()` connections.
- Client does not need to `connect()`.
- Server and Client are mirrors. Only logical difference is sequence of `sendto()` and `recvfrom()` calls.
- Sequence of Operation:
  1. Create `socket()`.
  2. `bind()` socket to own address and port.
  3. `sendto()` 'their' address or `recvfrom()` 'random' address on own socket.
  4. `close()` socket.

5 Using the Provided Code

The code provided with this lab has two examples, one for each TCP and UDP communication model. Extract the zip files and open the solution with the `.sln` file. Each solution contains two projects for Client and Server. Make changes to `ClientMain.cpp` and `ServerMain.cpp` files.

If `winsock2.h` cannot be included from TCP/UDP examples then install the Microsoft SDK. Here's the link to Microsoft SDK for Windows 7: `http://www.microsoft.com/en-us/download/details.aspx?id=8279`

6 Useful Functions

6.1 Send/Receive Variables

Variables can by anything `int`, `float`, `char` or even whole `struct` type variables.

```
// Send or receive a variable. Obj can be server or client object/
   variable.
int x = 3;
char c = '$';
DictinoaryEntry structvar;
// Sending functions. Use SendTo for UDP.
Obj.Send((void*)&x, sizeof(int));
Obj.Send((void*)&c, sizeof(char));
Obj.Send((void*)&structvar, sizeof(DictionaryEntry));

int xtemp;
char ctemp;
DictinoaryEntry structtemp;
// Receiving functions. Use ReceiveFrom for UDP.
Obj.Receive((void*)&xtemp, sizeof(int));
Obj.Receive((void*)&ctemp, sizeof(char));
Obj.Receive((void*)&structtemp, sizeof(DictionaryEntry));
```

6.2 Send/Receive Whole Arrays

```
const int SIZE = 20;
int Data[SIZE];
Obj.Send((void*)Data, sizeof(int)*SIZE);

int RecData[SIZE];
Obj.Receive((void*)RecData, sizeof(int)*SIZE);
```

6.3 Send/Receive Strings Using Built–in Receive() Without Arguments

Strings can be sent in a similar manner. However, note that receiver doesn't receive NULL. A convenient `receive()` function is available that automatically appends NULL to received string and stores the received string in a built–in buffer. Maximum size of internal buffer is 512 bytes so maximum string length can be 511. Larger strings must be split and sent separately.

```
char Message[30] = "Hello world!";
Obj.Send((void*)Message, sizeof(Message)); // Note: sizeof() gives
   string length. strlen(Message) is equally good.
// Can also send const strings.
Obj.Send((void*)"Hello", sizeof("Hello"));

// When receiving a string we can use the built-in Receive()/
   ReceiveFrom() functios without arguments. These automatically
   append a NULL to received string and stores in internal buffer.
Obj.Receive(); // or Obj.ReceiveFrom(); for UDP.
cout << Obj.GetBuffer() << endl; // Display received string from
   internal buffer.
```

7 Keeping Track of Data Sent/Received

The send and receive functions return the number of bytes that were actually sent and received. Receiver may not always know exactly how many bytes it is going to receive beforehand. This can be illustrated using example given below. Sender sends a string to receiver. Both agree on a maximum array size of 512. The length of string can be anything between 1 and 511.

```
// Send whole string including NULL.
char Message[512] = "Hello world!";
int BytesSent; //  BytesSent should be equal to sizeof(Message)+1.
```

```
BytesSent = Obj.Send((void*)Message, sizeof(Message)+1); // sizeof(
    Message) is equivalent to strlen(Message).

char Buffer[512];
int ByteseReceived;
BytesReceived = Obj.Receive((void*)Buffer, 512); // 512 is max
    buffer size. Received bytes will be equal to sent bytes.
```

8 Exercise

Question No. 1: Create a simplex chat application for two people using UDP. In simplex communication only one person can speak at a time. Client and Server will alternate their `send()` and `receive()` calls in a loop.
Question No. 2: Implement file transfer using TCP. The server keeps a pdf file (can be this lab handout. Copy into Server project folder). The sequence of operation is as follows:

1. Client will connect to Server.
2. Server will open file for reading and send file size to Client. Both, Server and Client, file operations should be in binary mode as both needs to read/write byte–by–byte.

3a. Server will loop through, read the file 1 byte at a time (into a temporary `char` variable) and send to client. Loop breaks when Server has sent number of bytes equal to file size.

3b. Client will create an empty file for writing. Client loops through in receiving state storing received data 1 byte at a time into a temporary `char` variable. The received data is written onto opened file. Client loop will break when it receives total number of bytes equal to file size.