Object Oriented Methodology

Week – *8*, Lecture – *3*

Introduction to POSIX Sockets

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Sockets

A *Socket* is an end of a logical communication session between two members of a network

Sockets are part of a Protocol that operates on layer 4 – the Transmission Control Protocol or TCP

A Socket consists of four pieces of information

- Local IP Address
- Local Port Number
- Remote IP Address
- Remote Port Number

If two Sockets differ in even one of these, they are treated as different Sockets

- Two different Sockets between the same machines, represent two independent communication sessions
- The data sent over one Socket does not interfere with data sent over another Socket

Understanding the scene

For communication, you need two parties or two machines connected to each other “somehow”

- We don’t actually care about the connection details – it is managed by *lower* layers

Out of the two machines, one plays the role of a *server*, while other becomes the *client*

- This does not mean that the server is some powerful machine, sitting in a data centre !!
- Your laptop, in fact, a VM on your laptop, could be the server in the communication as well
- The only expectation from the server is that it is “online” (i.e., listening for a new connection) all the time
- Whereas, a client can be online when it wants, and then, leave the network if it wishes
- Thus, you will need *one process executing all the time*, called the server …
- … and the other may start and end, i.e., the client

For the communication setup to work, the server *must be* running, before the client starts

Typical Networking Flow

At the *server*

1. A process is started to “listen” for connections on a *fixed* port number, say `P`
2. When a new connection is received, a local socket is created, say S_{server}
   - S_{server} is identified by the four elements we discussed
   - It is logically connected to peer socket on client, S_{client}
3. S_{server} can be used to send messages to, or, receive messages from S_{client}

At the *client*

1. A process is started to “connect” to a server with a known IP, over the *fixed* port, `P`
2. If the server is online and it accepts the connection, a local socket, S_{client} is created
   - S_{client} is identified by the four elements we discussed
   - It is logically connected to peer socket on server, S_{server}
3. S_{client} can be used to send messages to, or, receive messages from S_{server}

Typical Networking Flow

At the *server*

1. A process is started to "listen" for connections on a *fixed* port number, say `P`
2. When a new connection is received, a local socket is created, say S_{server}
   - S_{server} is identified by the four elements we discussed
   - It is logically connected to peer socket on client, S_{client}
3. S_{server} can be used to send messages to, or, receive messages from S_{client}

At the *client*

1. A process is started to "connect" to a server with a known IP, over the *fixed* port, `P`
2. If the server is online and it accepts the connection, a local socket, S_{client} is created
   - S_{client} is identified by the four elements we discussed
   - It is logically connected to peer socket on server, S_{server}
3. S_{client} can be used to send messages to, or, receive messages from S_{server}

Keep in mind that this has to be done before this

Typical Networking Flow

At the *server*

1. A process is started to "listen" for connections on a *fixed* port number, say `P`
2. When a new connection is received, a local socket is created, say S_{server}
   - S_{server} is identified by the four elements we discussed
   - It is logically connected to peer socket on client, S_{client}
3. S_{server} can be used to send messages to, or, receive messages from S_{client}

At the *client*

1. A process is started to "connect" to a server with a known IP, over the *fixed* port, `P`
2. If the server is online and it accepts the connection, a local socket, S_{client} is created
   - S_{client} is identified by the four elements we discussed
   - It is logically connected to peer socket on server, S_{server}
3. S_{client} can be used to send messages to, or, receive messages from S_{server}

When we are at this stage, there is no difference between the server and the client – they can both send and receive messages

Creating Sockets with POSIX library

Similar to the multithreading library, there is another POSIX library for networking

POSIX sockets allow you to use a Socket Descriptor for performing Socket I/O

- A Socket Descriptor is similar to a File Descriptor in C – it is just an integer
- You can even use the `read()` and `write()` methods directly over a Socket Descriptor as well !

However, unlike the threading library, you'll need to know about too many details

- This involves including multiple header files and knowing about different types of structures

Since it would be challenging for you to do so, I have created two classes for you

- If you wish to communicate over sockets in your project, feel free to use these classes

We will have a look at the code only at a higher level in this lecture

The idea is to learn how typical network programming flow is achieved …

- … and we will do so by just looking at some library functions and class methods

Typical POSIX Workflow - (1/2)

The library functions invoked at the server are

1. `socket()` – to create a new Socket Descriptor
2. `bind()` – to attach the socket to a specific Port Number (the port to which the client connects)
3. `listen()` – to mark the socket as an entity that remains in a listening mode
4. `accept()` – a "blocking" function call, which ideally returns when a connection has been established
   - It returns another socket descriptor – this one is used for further communication with the client
5. `send()` and `recv()` – to send or receive messages over the network
6. `close()` – to close the created sockets after we are done

Typical POSIX Workflow – (2/2)

The library functions invoked at the client are

1. `socket()` – to create a new Socket Descriptor
2. `connect()` – to initiate a connection to the server over the fixed port
3. `send()` and `recv()` – to send or receive messages over the network
4. `close()` – to close the created socket after we are done

Utility Classes for your use

The `Socket` class

- An instance of the `Socket` class is the communicating entity at the client end
- To create an instance of `Socket`, you need to provide the server's IP and the pre-decided Port number
- An attempt to for a connection to the server can be made by calling `connect_to_server()` method

The `ServerSocket` class

- An instance of `ServerSocket` class is the communicating entity at the server end
- To create an instance of `ServerSocket`, you need to provide the pre-decided Port Number for listening
- A blocking call to listen for an incoming connection can be made by calling `listen_for_connection()`

Post connection establishment

- Instances of `Socket` as well as `ServerSocket` can send and receive messages
- Both classes have a `send_message()` and a `receive_message()` method for the same

Let us now see some code…

Homework !!

The server we created is pretty simple

- It can only handle one client, and then, it must be restarted
- See if you can make it handle multiple simultaneous connections
- Check out this link: https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-clients-on-server-without-multi-threading/

Create a class called `AbstractSocket` …

- … and make `Socket` and `ServerSocket` derive from it
- What fields/methods from these classes could be part of `AbstractSocket`?

Additional Reading

Read more about User Datagram Protocol or UDP

- Figure out how it is different from TCP
- This link may be helpful: https://www.geeksforgeeks.org/differences-between-tcp-and-udp/

If you could not ping your VM from your host machine, you may be tempted to read this article

- https://www.virtualbox.org/manual/ch06.html#networkingmodes