Chapter 1

Interprocess Communication

1.1 Introduction

The processes running on the same machine uses the below methods to pass messages between each other:

1. Pipe
2. FIFO (Named pipes)
3. Message Queues
4. Shared Memory

Semaphores are used to synchronize shared memory access between multiple processes (related to race conditions and critical sections).

In this chapter, we will see how processes running on two or more networked computers communicate each other using the "Socket" API. Socket is a transport layer concept. Transport Layer is also known as Socket-Layer. As we learn socket programming, we refer to the diagram in Figure 1 to understand computer networking fundamentals.

1.2 Client-Server Communication

Consider that an application running on the host machine "A1" wants to send a message to another application running on the host machine "E2". This message sending activity can be implemented using the transport layer protocols TCP or UDP. The application which initiates

the communication is called the *client*. The application which accepts the message from the client, processes the message and either return a response back to the client or do not return a response is the *Server* application. The server never intiates the communication. If this client-server communication is using TCP protocol, it's called a *connection-oriented* communication and if it's using UDP, it's called a *connectionless* communication. The connection-oriented TCP communication uses a procedure called *Three-Way Handshaking* during the connection initiation stage. The three-way handshaking establishes a dedicated connection between the server and the client before starting the real message exchange.

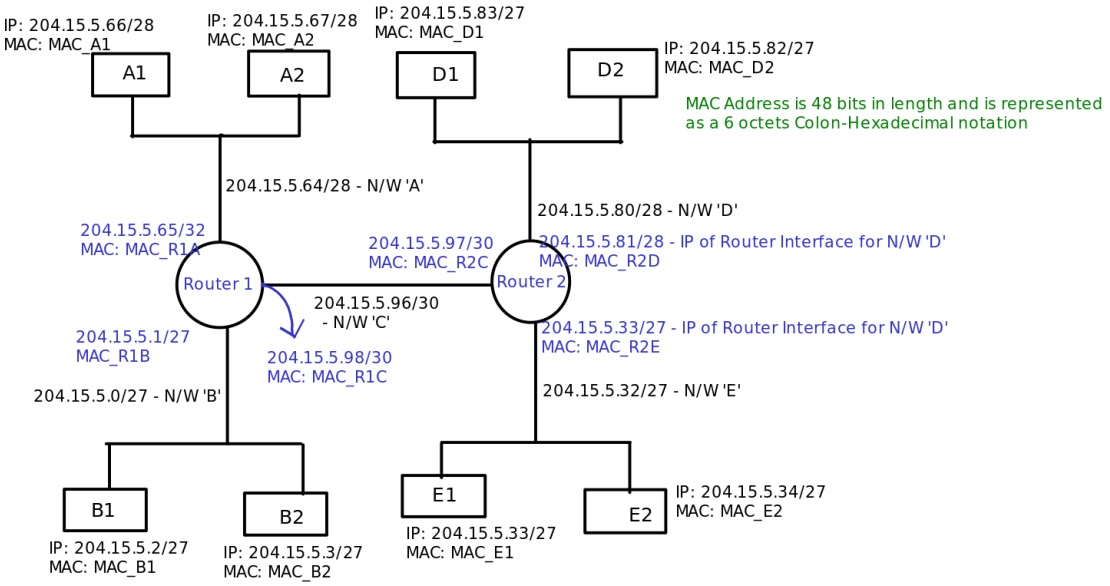

Figure 1: Example n/w architecture diagram

1.3 Socket Programming

Let's now try to understand the architecture of a very basic client-server implementation using TCP. *Sockets APIs* are used to implement network programs. Sockets APIs or simply *sockets* are a set of C library functions and system calls that provides a programming interface to develop user space network applications. In Linux, we use the below system calls to implement a TCP based client-server application. (The descriptions are copied from Linux man page)

1. socket() → Creates a socket
2. connect() → Connects a socket to a remote socket address.
3. bind() → Binds the socket to a local IP Address

4. listen() → Listens for an incoming connection on a port mentioned (tells the socket that new connections shall be accepted)
5. accept() → Accepts an incoming connection request and creates communication file descriptor for the new client.
6. send()/sendto()/sendmsg() → send data over a socket.
7. recv()/recvfrom()/recvmsg() → receive data from a socket.
8. Client uses *connect()* system call to initiate a connection and the Server uses *accept()* system call to accept the connection initiation request from the client.

Example code for the server and client programs can be found at: server client

1.3.1 TCP Server - Multiple Connections

We can implement a TCP Server that accepts connections from multiple clients using *fork()* or *multithreading (pthread)* or *select()*. We will see an example of an implementation based on the *select()* system call.

select() system call

select() system call allows the monitoring of multiple file descriptors in a single threaded program. We a server program accepts a connection request using the accept() system call, it creates a file descriptor called client communications file descriptor.

For example, *client_sock* is the communications file descriptor in the below line of code.

```
client_sock = accept(server_sock, (struct sockaddr *) &client_addr, &len);
```

When we have multiple connections to the server program, we need a way to monitor the server's master socket for new connections and all communications file descriptors for any data sent to the server. The select() system call enables I/O multiplexing hence it helps us to manage this scenario.

select() is a blocking system call. The program execution is unblocked when either a new connection request arrives or some data arrive from a client. All file descriptors (master socket and communications file descriptors) are added to a special data structure called *FD_SET* and select operates on this data structure.

Example code for the server and client programs can be found at: mserver mclient

References

[1] K C Wang. *Systems Programming in Unix/Linux*. Springer. ISBN: 978-3-319-92429-8.

[2] Douglas E. Comer *Internetworking with TCP/IP Volume One*. Pearson.