Network Lab Report

Shuvayan Ghosh Dastidar
001810501044
JU-BCSE - III

Assignment 7:
Implement Application Layer Protocols using TCP/UDP socket as applicable.

Protocols implemented:

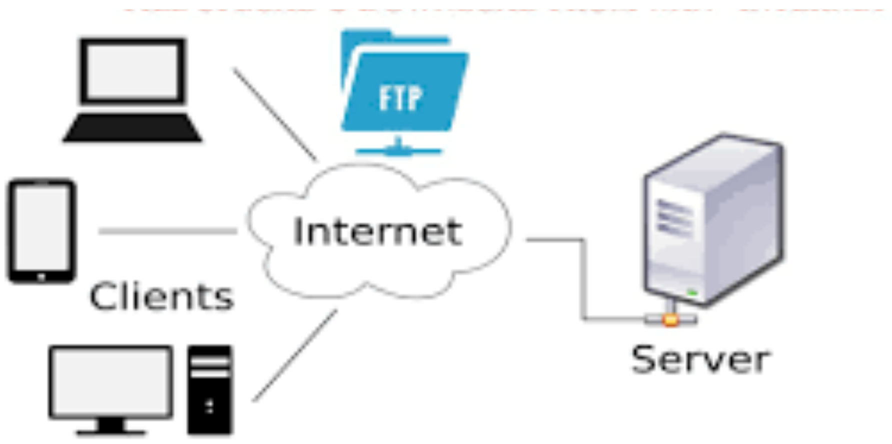
- FTP(File Transfer Protocol based on TCP sockets)
- DHCP (Dynamic Host Configuration Protocol based on UDP sockets)

FTP (File Transfer Protocol)

The **File Transfer Protocol** (**FTP**) is a standard network protocol used for the transfer of computer files from a server to a client on a computer network. FTP is built on a client-server model architecture using separate control and data connections between the client and the server. FTP users may authenticate themselves with a clear-text sign-in protocol, normally in the form of a username and password, but can connect anonymously if the server is configured to allow it. For secure transmission that protects the username and password, and encrypts the content, FTP is often secured with SSL/TLS (FTPS) or replaced with SSH File Transfer Protocol (SFTP).

The first FTP client applications were command-line programs developed before operating systems had graphical user interfaces, and are still shipped with most Windows, Unix, and Linux operating systems. Many FTP clients and automation utilities have since been developed for desktops, servers, mobile devices, and hardware, and FTP has been incorporated into productivity applications, such as HTML editors.

Schematic Diagram



Communication between client and server

The server is initiated and listens for commands from the clients on a thread.

```
 private final int port;

    private ServerSocketChannel serverSocket;

    private Selector selector;

    private ByteBuffer buffer = ByteBuffer.allocate(1024);

    private ExecutorService executor;



    public Server(int port) throws IOException {

        this.port = port;

        this.serverSocket = ServerSocketChannel.open();

        this.serverSocket.socket().bind(new InetSocketAddress("localhost", port));

        this.serverSocket.configureBlocking(false);

        this.selector = Selector.open();



        this.serverSocket.register(selector, SelectionKey.OP_ACCEPT);

        executor = Executors.newSingleThreadExecutor();

        executor.execute(() -> listen());

    }



    private void listen() {

        try {

            System.out.println("[INFO] Server starting on port " + this.port +
" ....");

            Iterator<SelectionKey> iterator;
            SelectionKey key;

            while (this.serverSocket.isOpen()) {

                selector.select();

                iterator = this.selector.selectedKeys().iterator();

                while (iterator.hasNext()) {

                    key = iterator.next();

                    iterator.remove();

                    if (key.isAcceptable()) {

                        handleAccept(key);

                    } else if (key.isReadable()) {

                        handleRead(key);

                    }


                }

            }

        } catch (IOException e) {

            System.out.println("IOException, server of port " + this.port + "
terminating. Stack trace:");

            e.printStackTrace();

        }

    }
```

The handle accept accepts a new connection and handleRead reads the command from the client parses it and returns the appropriate response with data, if applicable.

```
 private void handleRead(SelectionKey key) throws IOException {

        SocketChannel socketChannel = ((SocketChannel) key.channel());

        StringBuilder stringBuilder = new StringBuilder();


        buffer.clear();

        int read = 0;

        try {
            socketChannel.read(buffer);

            // for reading

            buffer.flip();

            byte[] bytes = new byte[buffer.limit()];

            buffer.get(bytes);

            stringBuilder.append(new String(bytes));

            buffer.clear();


        } catch (Exception e) {

            key.cancel();

            read = -1;

        }

        String client_message = stringBuilder.toString()

      byte[] bytes = analyze(stringBuilder.toString());


        if (bytes == null) {

            socketChannel.write(ByteBuffer.wrap("EXITING SOCKET BYE".getBytes()));

            socketChannel.close();

        }

}
```

Commands Implemented

- ### *FTP List*

This command is implemented in the server side. The client enters the command in the shell and is passed on to the FTP server and on receiving the command, the server checks all the files present in the server and returns all the files present.

Code in the server end

```
File[] files = new File("src/Server/Data").listFiles();
                String res = "\tLISTING --\n\n";
                for (File file : files) {
                    res += file.getName() + "\n";
                }
                res += "\n200 OK .\n";
                bytes = res.getBytes();
                return bytes;
```

- *FTP DSIZ*

This command is implemented in the server side. The client enters the command in the shell and is passed on to the FTP server and on receiving the command, the server calculates the size of all the files in the storage space of the FTP server and returns to the FTP client.

Code in the server end

```
File[] files = new File("src/Server/Data").listFiles();
                double sum = 0;
                for (File file : files) {
                    sum += (double) file.length() / (1024 * 1024);
                }
                String res = "\nDirectory size : " + String.valueOf(sum) + "MB .\n";
                res += "\n200 OK .\n";
                bytes = res.getBytes();
                return bytes;
```

- *FTP DELE <file>*

This command is implemented in the server side. The client enters the command in the shell and is passed on to the FTP server and on receiving the command, the server checks if file is present in the server and on success, deletes the file in the FTP server and returns 200 OK.

Code in the server end

```
String filename = tokens[2];

                ArrayList<String> files = new ArrayList<>(Arrays.asList(new File("src/
Server/Data").list()));

                if (!files.contains(filename)) {

                    bytes = "ERROR 400 - FILE NOT FOUND".getBytes();

                    return bytes;

                }

                for (File file : new File("src/Server/Data").listFiles()) {

                    if (file.getName().equals(filename)) {

                        file.delete();

                        bytes = "200 OK - FILE DELETED".getBytes();

                        return bytes;

                    }

                }
```

• *FTP RECV <file>*

This command is implemented in the server side. The client enters the command in the shell and is passed on to the FTP server and on receiving the command, the server checks if file is present in the server and on success, converts the binary data of the file into a base64 encoded string and passes on to the FTP client.

Code in the server end

```
String filename = tokens[2];

                bytes = "ERROR".getBytes();


                ArrayList<String> files = new ArrayList<>(Arrays.asList(new File("src/
Server/Data").list()));

                if (!files.contains(filename)) {

                    bytes = "ERROR 400 - FILE NOT FOUND".getBytes();

                    byte[] base64 = Base64.getEncoder().encode(bytes);

                    return base64;

                }

                for (File file : new File("src/Server/Data").listFiles()) {

                    if (file.getName().equals(filename)) {

                        try {

                            bytes = Files.readAllBytes(Paths.get(new
File(file.getPath()).toURI()));

                            break;


                        } catch (IOException e) {

                            bytes = "ERROR 500 - IO Exception".getBytes();

                            System.out.println("ERROR IO Exception in reading file : "
+ file.getName());

                            e.printStackTrace();

                        }
                    }

                }


                byte[] base64 = Base64.getEncoder().encode(bytes);

                return base64;


            }
```

- *FTP SEND<file>*

This command is implemented in the client side. The client enters the command in the shell and the base64 encoded binary data of the file and the server parses it and stores the file after decoding it in the server storage.

Code in the server end

```
 String filename = tokens[2];

            int index = client_message.indexOf(filename);


            index += filename.length();

            index++;

            String content = client_message.substring(index);

            byte[] res = Base64.getDecoder().decode(content.getBytes());

            byte[] status = makeFileandWrite(filename, res);

            System.out.println("[INFO] SERVER GOT MESSAGE " + key.attachment());

            socketChannel.write(ByteBuffer.wrap(status));
```

Observations and Analysis

- For transferring image files from the client to the server, the image has to be encoded in a base64 string and then sent, otherwise it can’t go to the server.
- Image files take much time to transfer to the server as compared to the text files.

DHCP (Dynamic Host Configuration Protocol)

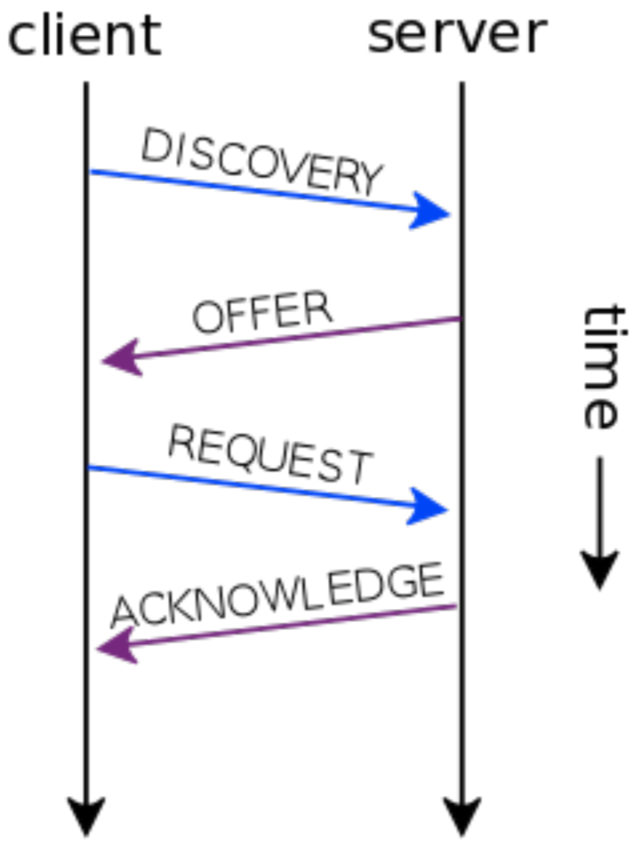
The **Dynamic Host Configuration Protocol** (**DHCP**) is a network management protocol used on Internet Protocol (IP) networks, whereby a DHCP server dynamically assigns an IP address and other network configuration parameters to each device on the network, so they can communicate with other IP networks. A DHCP server enables computers to request IP addresses and networking parameters automatically from the Internet service provider (ISP), reducing the need for a network administrator or a user to manually assign IP addresses to all network devices. In the absence of a DHCP server, a computer or other device on the network needs to be manually assigned an IP address, or to assign itself an APIPA address, the latter of which will not enable it to communicate outside its local subnet.

DHCP can be implemented on networks ranging in size from home networks to large campus networks and regional ISP networks. A router or a residential gateway can be enabled to act as a DHCP server. Most residential network routers receive a globally unique IP address within the ISP network. Within a local network, a DHCP server assigns a local IP address to each device connected to the network.

Depending on implementation, the DHCP server may have three methods of allocating IP addresses:

Dynamic allocation
A network administrator reserves a range of IP addresses for DHCP, and each DHCP client on the LAN is configured to request an IP address from the DHCP server during network initialization. The request-and-grant process uses a lease concept with a controllable time period, allowing the DHCP server to reclaim and then reallocate IP addresses that are not renewed.
Automatic allocation
The DHCP server permanently assigns an IP address to a requesting client from the range defined by the administrator. This is like dynamic allocation, but the DHCP server keeps a table of past IP address assignments, so that it can preferentially assign to a client the same IP address that the client previously had.

Manual allocation
Also commonly called *static allocation* and *reservations*. The DHCP server issues a private IP address dependent upon each client's *client id* (or, traditionally, the client MAC address), based on a predefined mapping by the administrator. This feature is variously called *static DHCP assignment* by DD-WRT, *fixed-address* by the dhcpd documentation, *address reservation* by Netgear, *DHCP reservation* or *static DHCP* by Cisco and Linksys, and *IP address reservation* or *MAC/IP address binding* by various other router manufacturers. If no match for the client's *client ID* (if provided) or MAC address (if no client id is provided) is found, the server may or may not fall back to either Dynamic or Automatic allocation.
DHCP is used for Internet Protocol version 4 (IPv4) and IPv6. While both versions serve the same purpose, the details of the protocol for IPv4 and IPv6 differ sufficiently that they may be considered separate protocols.[7] For the IPv6 operation, devices may alternatively use stateless address autoconfiguration. IPv6 hosts may also use link-local addressing to achieve operations restricted to the local network link.



Working of a DHCP server and client

Implementation

There is a base Packet class which is used as a base class for implementing the packet structure of the all the DHCP requests - DHCP discover, DHCP offer, DHCP Request and DHCP acknowledge.

Code for the Base Packet Class

```
package Packet;

public class BasePacket {

    String ciaddr; // IP Address of this machine ( if already there )
    String yiaddr; // IP Address of this machine ( provided by the DHCP server )
    String siaddr; // IP Address of the DHCP server
    String giaddr; // IP Address of the DHCP relay

    String macaddr; // The mac address of this machine
    String option;

}
```

The DHCP Discover Packet Class has all the fields in the IP address fields as 00 meaning not yet configured and sets the Mac address field and sets the option field to distinguish as a DHCP Discover Packet.

Code for the DHCP Discover Packet Class

```
package Packet;

public class DHCPDiscoverPacket extends BasePacket {


    public static DHCPDiscoverPacket decode(String s) {
        String[] tokens = s.split("[;]+");
        DHCPDiscoverPacket packet = new DHCPDiscoverPacket();
        packet.setCiaddr(tokens[0]);
        packet.setYiaddr(tokens[1]);
        packet.setSiaddr(tokens[2]);
        packet.setGiaddr(tokens[3]);
        packet.setMacaddr(tokens[4]);
        packet.setOption(tokens[5]);
        return packet;
    }

    public static DHCPDiscoverPacket getDHCPDiscoverPacket(String macAddress) {
        DHCPDiscoverPacket packet = new DHCPDiscoverPacket();
        packet.setCiaddr("00");
        packet.setYiaddr("00");
        packet.setGiaddr("00");
        packet.setSiaddr("00");
        packet.setMacaddr(macAddress);
        packet.setOption(Integer.toHexString(1));
        return packet;

    }


    @Override
    public String toString() {
        String s = ciaddr + ";" + yiaddr + ";" + siaddr + ";" + giaddr + ";" + macaddr
+ ";" + option;
        return s;
    }
}
```

The DHCP Offer Packet Classsets the option field to distinguish as a DHCP Offer Packet.

Code for the DHCP Offer Packet Class

```
public class DHCPOfferPacket extends BasePacket {


    public static DHCPOfferPacket decode(String s) {
        String[] tokens = s.split("[;]+");
        DHCPOfferPacket packet = new DHCPOfferPacket();
        packet.setCiaddr(tokens[0]);
        packet.setYiaddr(tokens[1]);
        packet.setSiaddr(tokens[2]);
        packet.setGiaddr(tokens[3]);
        packet.setMacaddr(tokens[4]);
        packet.setOption(tokens[5]);
        return packet;
    }

    public static DHCPOfferPacket getDHCPOfferPacket(String macAddress) {
        DHCPOfferPacket packet = new DHCPOfferPacket();
        packet.setCiaddr("00");
        packet.setYiaddr("00");
        packet.setGiaddr("00");
        packet.setSiaddr("00");
        packet.setMacaddr(macAddress);
        packet.setOption(Integer.toHexString(2));
        return packet;

    }


    @Override
    public String toString() {
        String s = ciaddr + ";" + yiaddr + ";" + siaddr + ";" + giaddr + ";" + macaddr
+ ";" + option;
        return s;
    }
}
```

The server checks if the mac address in the DHCP discover packet is in the static table stored in the server and if present, it returns the mapped IP address to the client. If not present it generates a new IP and stores it in the static table creating a mapping and then returns the IP address to the client by setting the server IP address and **yiaddr (offered IP address).**

Code for configuring DHCP offer packet

```
DHCPOfferPacket packet2 = DHCPOfferPacket.getDHCPOfferPacket(packet1.getMacaddr());
                    if (macToIps.containsKey(packet1.getMacaddr())) {
                        packet2.setSiaddr(serverAddress);
                        packet2.setYiaddr(macToIps.get(packet1.getMacaddr()));
                    } else {
                        String ip = generateIp();
                        macToIps.put(packet1.getMacaddr(), ip);
                        packet2.setSiaddr(serverAddress);
                        packet2.setYiaddr(ip);
                    }
                    buffer = packet2.toString().getBytes();
                    packet = new DatagramPacket(buffer, packet2.toString().length(),
address, port);
                    socket.send(packet);
                    System.out.println("[INFO] Sent DHCP Offer Packet to address : " +
address + " port: " + port);
```

Code for the DHCP Request Packet Class

```
public class DHCPRequestPacket extends BasePacket {

    public static DHCPRequestPacket decode(String s) {
        String[] tokens = s.split("[;]+");
        DHCPRequestPacket packet = new DHCPRequestPacket();
        packet.setCiaddr(tokens[0]);
        packet.setYiaddr(tokens[1]);
        packet.setSiaddr(tokens[2]);
        packet.setGiaddr(tokens[3]);
        packet.setMacaddr(tokens[4]);
        packet.setOption(tokens[5]);
        return packet;
    }

    public static DHCPRequestPacket getDHCPRequestPacket(String macAddress) {
        DHCPRequestPacket packet = new DHCPRequestPacket();
        packet.setCiaddr("00");
        packet.setYiaddr("00");
        packet.setGiaddr("00");
        packet.setSiaddr("00");
        packet.setMacaddr(macAddress);
        packet.setOption(Integer.toHexString(3));
        return packet;
    }


    @Override
    public String toString() {
        String s = ciaddr + ";" + yiaddr + ";" + siaddr + ";" + giaddr + ";" + macaddr
+ ";" + option;
        return s;
    }
}
```

Code for configuring DHCP Request Packet

```
// Broadcast the DHCP Request message for the IP given
            DHCPRequestPacket packet2 =
DHCPRequestPacket.getDHCPRequestPacket(macAddress);
            packet2.setSiaddr(packet1.getSiaddr());
            packet2.setCiaddr(packet1.getYiaddr());
            buffer = packet2.toString().getBytes();
            sendPacket = new DatagramPacket(buffer, buffer.length, address, 8067);
            socket.send(sendPacket);
            System.out.println("[INFO] DHCPRequestPacket broadcasted");
```

Code for configuring DHCP Acknowledge Packet

Server Side

```
if (type.equals("DHCPRequest")) {

                    // Create a DHCP Ack packet
                    DHCPRequestPacket packet1 = DHCPRequestPacket.decode(received);

                    System.out.println("[INFO] Got DHCP Request packet from  port : " +
packet.getPort() + " : " + packet.getSocketAddress() + " with mac address  : " +
packet1.getMacaddr());
```

```
                    DHCPAckPacket packet2 =
DHCPAckPacket.getDHCPAckPacket(packet1.getMacaddr());
                    if (macToIps.containsKey(packet1.getMacaddr())) {
                        packet2.setSiaddr(serverAddress);
                        packet2.setYiaddr(macToIps.get(packet1.getMacaddr()));
                    } else {
                        System.out.println("[INFO] Invalid DHCP Request");
                        packet2.setSiaddr(serverAddress);
                        packet2.setYiaddr("00");
                    }
                    buffer = packet2.toString().getBytes();
                    packet = new DatagramPacket(buffer, packet2.toString().length(),
address, port);
                    socket.send(packet);
                    System.out.println("[INFO] Sent DHCP ACK Packet to address : " +
address + " port: " + port);
                }
```

Client Side

```
// Receive the DHCP ACK regarding the following IP is active.
            buffer = new byte[4096];
            sendPacket = new DatagramPacket(buffer, buffer.length);
            socket.receive(sendPacket);
            received = new String(buffer).substring(0, sendPacket.getLength());
            DHCPAckPacket packet3 = DHCPAckPacket.decode(received);
            System.out.println("[INFO] DHCPAckPacket received from : " +
packet3.getSiaddr() + " and confirmed IP :" + packet3.getYiaddr());

            System.out.println("\nIP confirmed by DHCP server : " +
packet3.getYiaddr());
```

Observations

- The DHCP is a dynamic method as compared to BOOTP and allows dynamic IPs to be configured on to the clients with random time outs.
- The following code can be adjusted to implement BOOTP to just allow the static allocation of IP to the client and not allowing the dynamic IP configuration.