



**Department of Computer Science,
Electrical and Space Engineering**

**Network Programming and
Distributed Applications**

Lab 2 - Java Socket Programming and Security

Ameer Hamza, Otabek Sobirov

27th September 2021

Part I: Java GUI

Write a simple java GUI with two fields and one button. Where the first field is used to input commands and the second one is to display output.

The project has 3 classes. App class contains the window frame code and the program launching code. DetailPannel class has the design and event handling code while the Terminal class handles the command line execution code. Each command is executed in a separate thread. Our program is able to execute a wide variety of commands and supports pipelining.

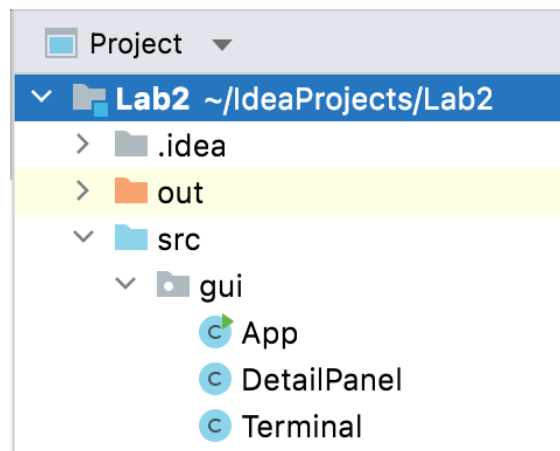


Figure 1.1: Project Structure

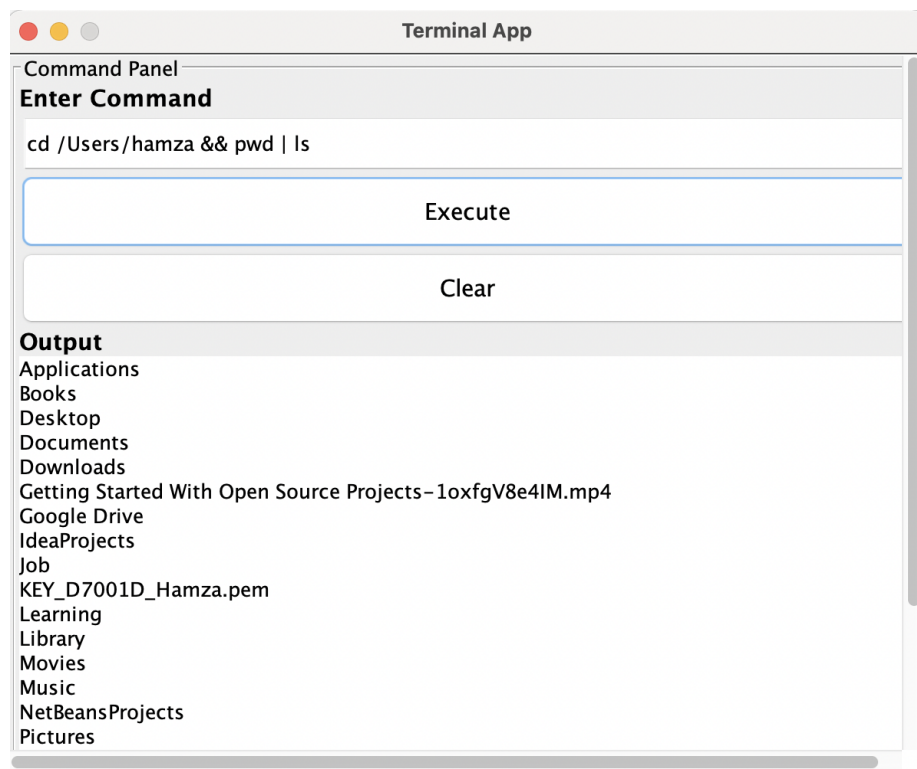


Figure 1.2: Mac-based GUI

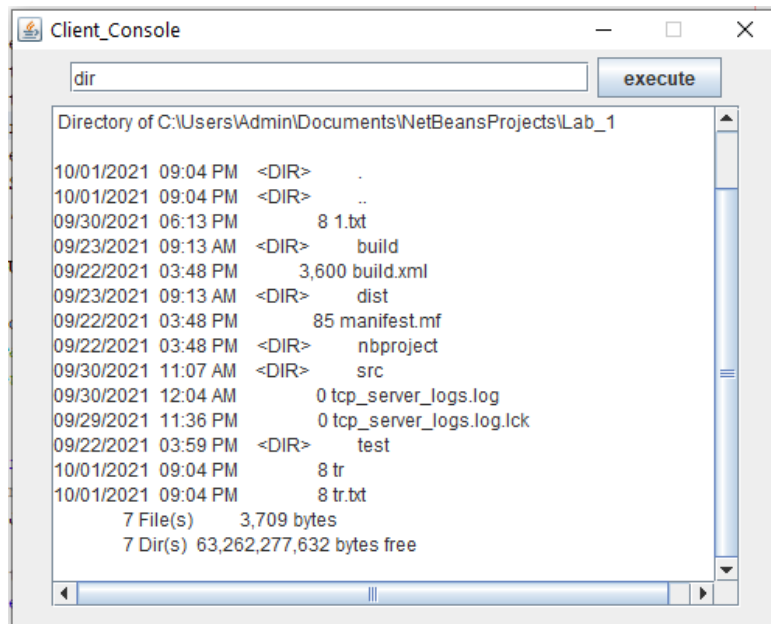


Figure 1.3: Windows-based GUI

Now write another Java class, which will execute an empty loop. The program should take the number of loop iterations as a parameter, print this in the output field, launch the loop. Launch this program from the GUI!

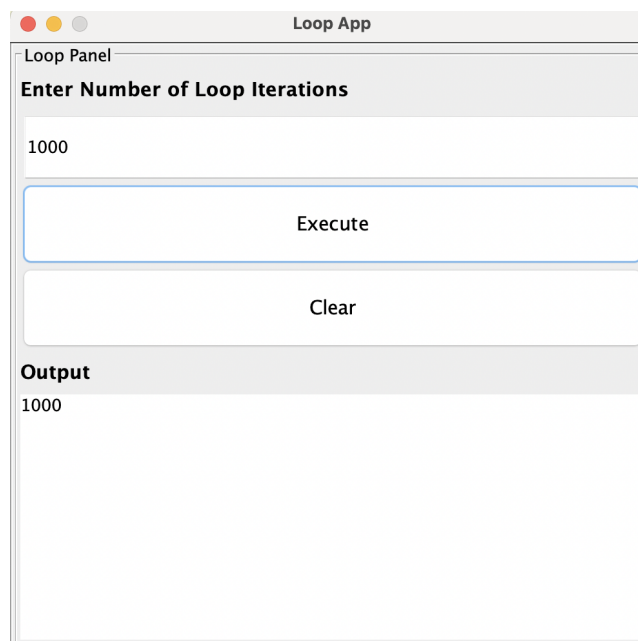
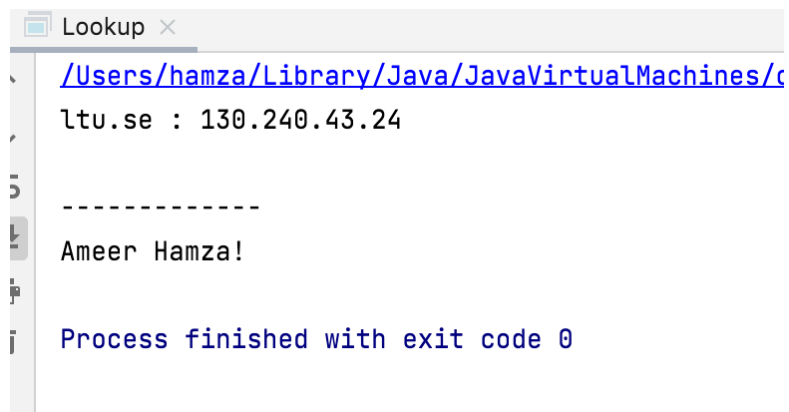


Figure 1.4: GUI for the Loop Program

Part II: Java netprog patterns – sockets & threads

Modify the Lookup class so that it outputs your name in addition to the input parameters that you supply at runtime.

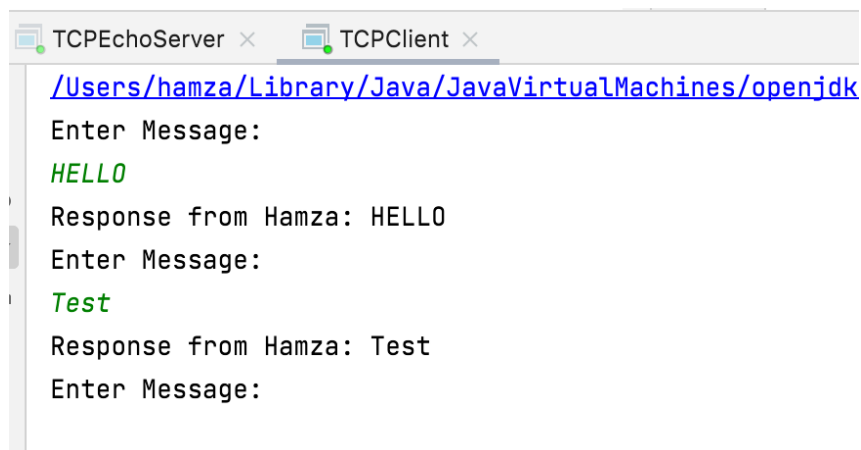
Lookup program returns the IP address of the given hostname.



```
Lookup x
/Users/hamza/Library/Java/JavaVirtualMachines/c
ltu.se : 130.240.43.24
-----
Ameer Hamza!
Process finished with exit code 0
```

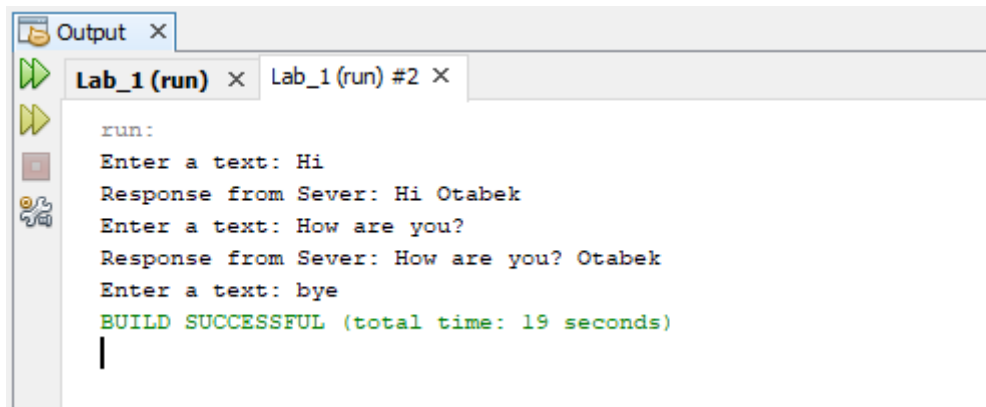
Figure 2.1: Lookup Program

Modify both the TCPEchoServer and UDPEchoServer classes so that in addition to echoed input symbols the server would send back your name.



```
TCPEchoServer x  TCPClient x
/Users/hamza/Library/Java/JavaVirtualMachines/openjdk
Enter Message:
HELLO
Response from Hamza: HELLO
Enter Message:
Test
Response from Hamza: Test
Enter Message:
```

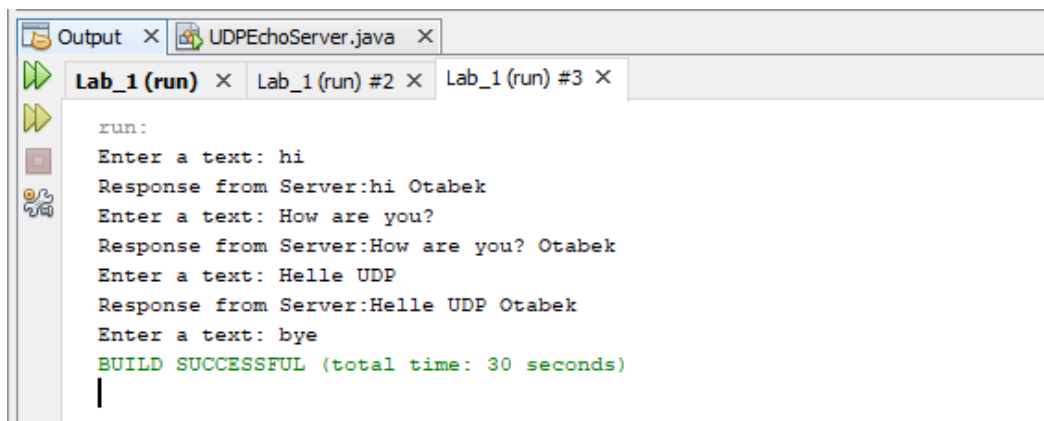
Figure 2.2: TCP Echo Client



```
Output x
Lab_1 (run) x Lab_1 (run) #2 x

run:
Enter a text: Hi
Response from Sever: Hi Otabek
Enter a text: How are you?
Response from Sever: How are you? Otabek
Enter a text: bye
BUILD SUCCESSFUL (total time: 19 seconds)
```

Figure 2.3: TCP Echo Server-Client communication



```
Output x UDPEchoServer.java x
Lab_1 (run) x Lab_1 (run) #2 x Lab_1 (run) #3 x

run:
Enter a text: hi
Response from Server:hi Otabek
Enter a text: How are you?
Response from Server:How are you? Otabek
Enter a text: Helle UDP
Response from Server:Helle UDP Otabek
Enter a text: bye
BUILD SUCCESSFUL (total time: 30 seconds)
```

Figure 2.4: UDP Echo Server-Client communication

For this task, we first wrote the client program which is able to send a message to the UDP/TCP server. Given server code was modified such that message was echoed with the user name.

Compile the class Race0 in the threads part of the project.

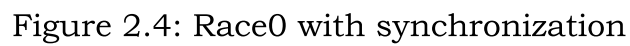
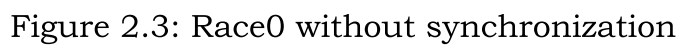
1. What kind of behavior do you observe?

Figure 2.3 shows that a race condition is observed. Since the main program and the thread have shared variables that are trying to modify at the same time,

2. Make now both dif() and bump() methods synchronize. Compile the classes and run.

3. How has the behavior changed now?

Figure 2.4 shows that now there is no race condition now as synchronization is being used. Synchronization locks the resources which prevent consistency errors.



Part III: Client-server application

1. The GUI runs as a client-side application. Add an additional field to your GUI where the user can enter the URL of the server-side application.
2. The entered commands should be executed and run as the server-side application. Implement your own multi-threaded TCP server

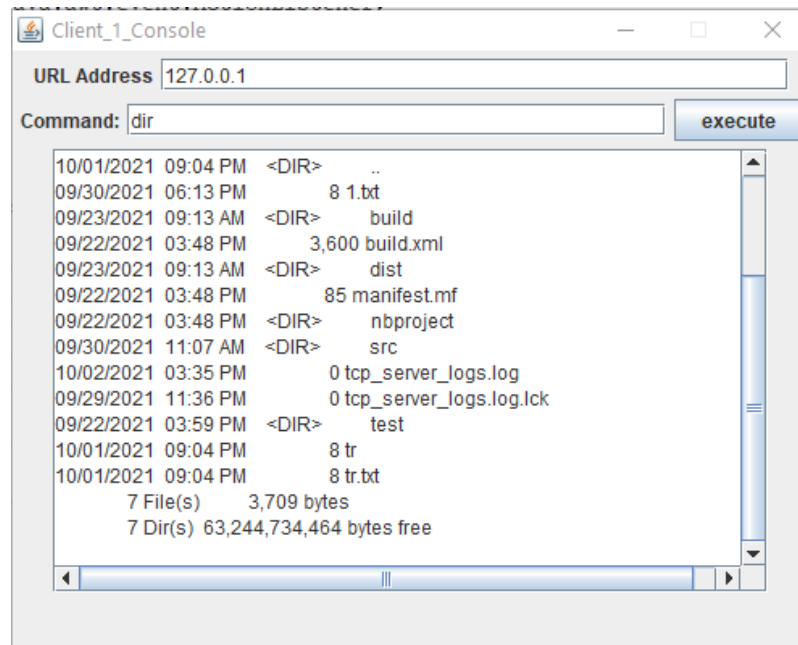


Figure 3.1 Multi Threaded Client-Server GUI (Windows)

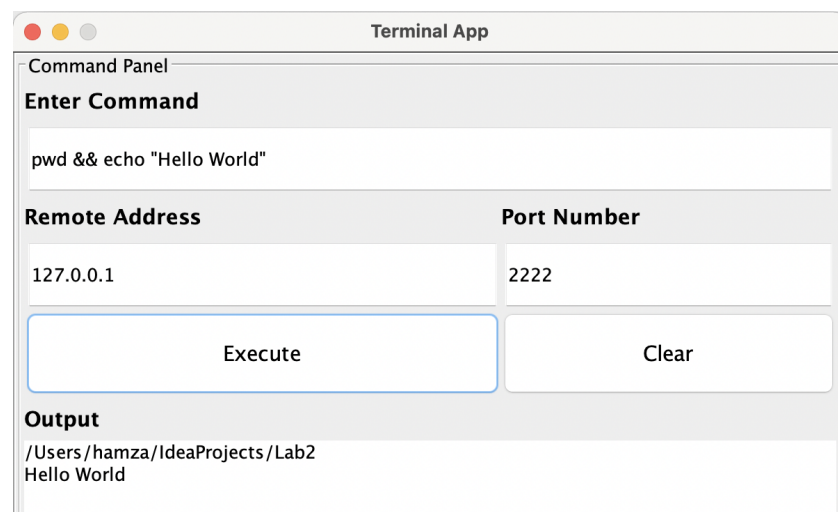


Figure 3.2 Multi-Threaded Client-Server GUI (Mac)

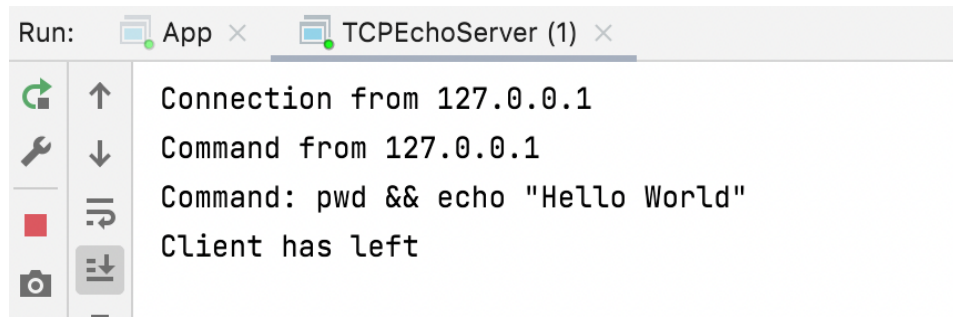


Figure 3.3 Multi-Threaded Server

Figure 3.1 & 3.2 shows that the GUI client takes the address and port number of the remote host and also takes the command from the user to be executed on the remote machine. Once the command is executed on the remote machine, a response from the remote machine is shown in the output area.

For this task, multithreading is used on the server-side which enables it to handle multiple connections/users simultaneously.

3. Logging

Logging is storing and/or analyzing events that have happened/been happening during a java application running. For this purpose Logger class is used. We created a logger and named it the same as the class name. Appenders(Handlers) are classes used to handle logger objects. We used FileHandler class to record the information that our global logger is sending.

4. Level of Logs

Logger class has the following default levels (however, custom levels can be created).

- a. SEVERE (Highest Level)
- b. WARNING
- c. INFO
- d. CONFIG
- e. FINE
- f. FINER
- g. FINEST (Lowest Level)

Logger class has the `setLevel()` method that sets the level by which the logger object records information. If it is set to INFO level, all the logs lower than this will be ignored by the logger object. We choose the INFO level, so we are interested only in errors, warnings, and some critical information logs that the server has.


```

public class ServerTCP extends Thread {

    private final static Logger LOGGER = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
    static final int BUFSIZE = 1024;
    private Socket s;
    static FileHandler fileHandler = null;

    public ServerTCP(Socket s) {
        this.s = s;
    }

    public static void main(String[] args) {
        try {
            LOGGER.setLevel(Level.INFO);
            fileHandler = new FileHandler("./tcp_server_logs.log");
            LOGGER.addHandler(fileHandler);

```

Figure 3.4. Setting up a logger object

5. Logging with different levels

Logging events can be done with two methods:

- a. `Logger.log(level, log_event);`
- b. `Logger.warning(log_event);`

In the first method, you need to specify the level by the user. On the other hand, the second one is a way of recording specific level logs. Each default level has its own methods with a corresponding name (e.g. `Logger.info(logs)`).

```

public void run() {
    try {
        handleClient(s);
        System.out.println("waiting for another client");
        LOGGER.setLevel(Level.INFO);
    } catch (IOException e) {
        //Logging the IO errors while running the client handler(Level is Severe)
        LOGGER.log(Level.SEVERE, "Exception occur", e);
    }
}

```

Figure 3.5. Recording Input/Output errors

```

61 while (in.read(buff) != -1) {
62     String request = new String(buff).trim();
63     String result;
64     System.out.println("Request from Client: " + request);
65     // Recording the command requested by a user (with its IP address)
66     LOGGER.log(Level.INFO, "Recording Client Command: ".format(request, s.getInetAddress()));
67

```

Figure 3.6. Recording the command requested by a user(with IP address)

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2021-10-03T17:56:02</date>
  <millis>1633276562621</millis>
  <sequence>0</sequence>
  <logger>global</logger>
  <level>INFO</level>
  <class>multi_thread_tcp.ServerTCP</class>
  <method>handleClient</method>
  <thread>11</thread>
  <message>dir</message>
</record>

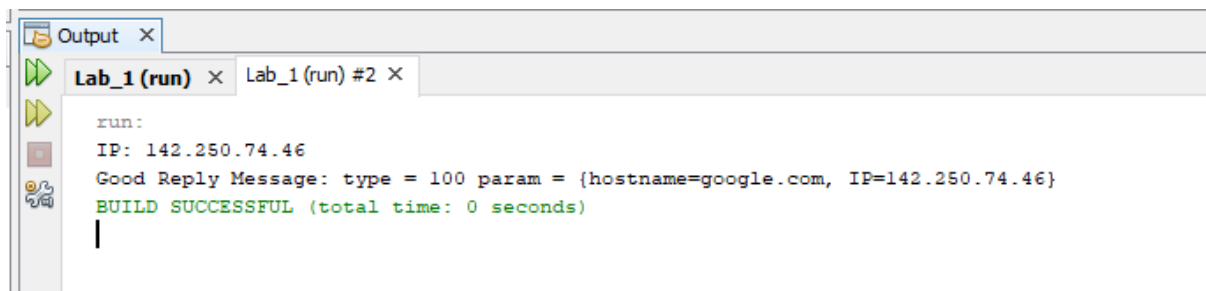
```

Figure 3.7. Recorded Logs from the server application

Part IV - Simple Messaging Architecture

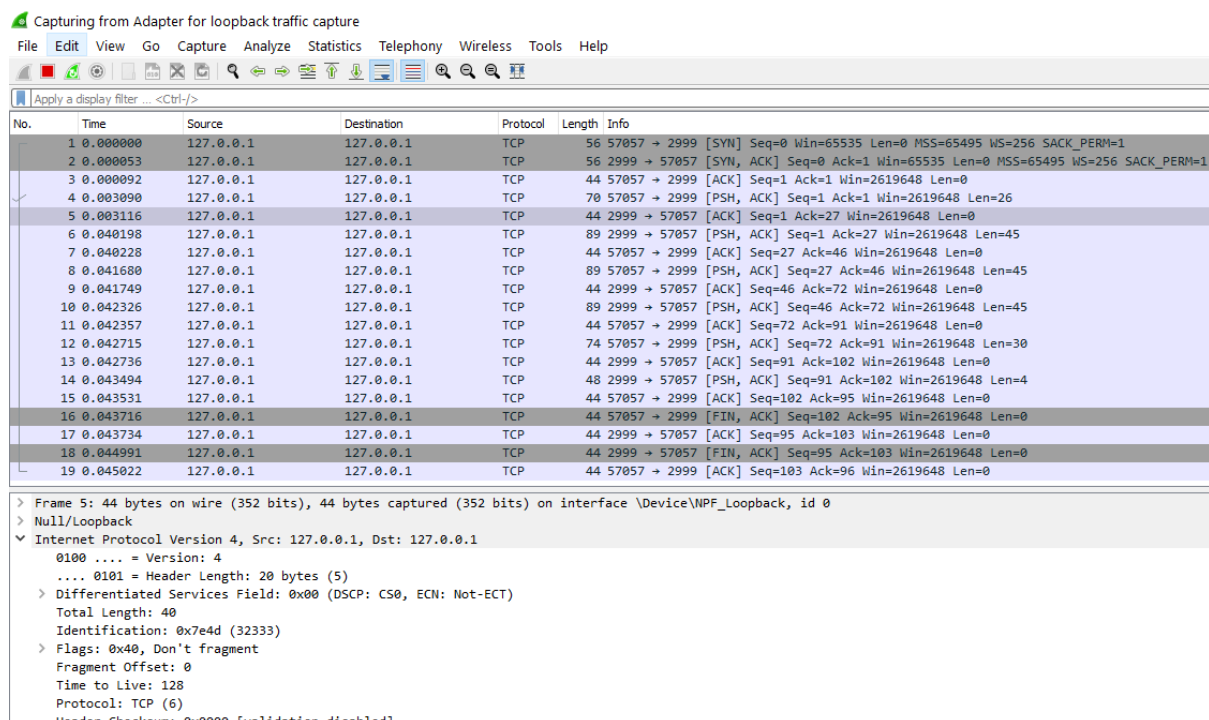
We ran the code given in the lab and analyzed it. We added the DNS resolving function (a function that takes DNS name and retrieves its public IP address) as the DNS Service Server class. The architecture given to exchange messages has as follows:

- DNS Service Server takes requests from users and replies.
- Message Server handles the message communication between DNS Service Server and DNS Service Clients.
- Message class provides the message structure for communication.
- Message Server Dispatcher creates a thread for each client communication.
- Deliverable.java provides an interface for DNS Service Server.
- DNS client requests the server to resolve DNS names into IP addresses.



```
run:
IP: 142.250.74.46
Good Reply Message: type = 100 param = {hostname=google.com, IP=142.250.74.46}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4.1. DNS Service Client request and response



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	56	57057 → 2999 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
2	0.000053	127.0.0.1	127.0.0.1	TCP	56	2999 → 57057 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
3	0.000092	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
4	0.003090	127.0.0.1	127.0.0.1	TCP	70	57057 → 2999 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=26
5	0.003116	127.0.0.1	127.0.0.1	TCP	44	2999 → 57057 [ACK] Seq=1 Ack=27 Win=2619648 Len=0
6	0.040198	127.0.0.1	127.0.0.1	TCP	89	2999 → 57057 [PSH, ACK] Seq=1 Ack=27 Win=2619648 Len=45
7	0.040228	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [ACK] Seq=27 Ack=46 Win=2619648 Len=0
8	0.041680	127.0.0.1	127.0.0.1	TCP	89	57057 → 2999 [PSH, ACK] Seq=27 Ack=46 Win=2619648 Len=45
9	0.041749	127.0.0.1	127.0.0.1	TCP	44	2999 → 57057 [ACK] Seq=46 Ack=72 Win=2619648 Len=0
10	0.042326	127.0.0.1	127.0.0.1	TCP	89	2999 → 57057 [PSH, ACK] Seq=46 Ack=72 Win=2619648 Len=45
11	0.042357	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [ACK] Seq=72 Ack=91 Win=2619648 Len=0
12	0.042715	127.0.0.1	127.0.0.1	TCP	74	57057 → 2999 [PSH, ACK] Seq=72 Ack=91 Win=2619648 Len=30
13	0.042736	127.0.0.1	127.0.0.1	TCP	44	2999 → 57057 [ACK] Seq=91 Ack=102 Win=2619648 Len=0
14	0.043494	127.0.0.1	127.0.0.1	TCP	48	2999 → 57057 [PSH, ACK] Seq=91 Ack=102 Win=2619648 Len=4
15	0.043531	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [ACK] Seq=102 Ack=95 Win=2619648 Len=0
16	0.043716	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [FIN, ACK] Seq=102 Ack=95 Win=2619648 Len=0
17	0.043734	127.0.0.1	127.0.0.1	TCP	44	2999 → 57057 [ACK] Seq=95 Ack=103 Win=2619648 Len=0
18	0.044991	127.0.0.1	127.0.0.1	TCP	44	2999 → 57057 [FIN, ACK] Seq=95 Ack=103 Win=2619648 Len=0
19	0.045022	127.0.0.1	127.0.0.1	TCP	44	57057 → 2999 [ACK] Seq=103 Ack=96 Win=2619648 Len=0

> Frame 5: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface \Device\NPF_{...} id 0

> Null/Loopback

▼ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

- 0100 = Version: 4
- 0101 = Header Length: 20 bytes (5)
- > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
- Total Length: 40
- Identification: 0xe4d (32333)
- > Flags: 0x40, Don't fragment
- Fragment Offset: 0
- Time to Live: 128
- Protocol: TCP (6)
- Header Checksum: 0x0000 [validation disabled]

Figure 4.2. Capturing the packets for DNS resolving function

Part V - Java netprog patterns – Security

1. JCE - Java Cryptography Extension - EncipherDecipher

EncipherDecipher.java extends Frame class. It uses JCE (Java Cryptography Extension), an optional java package. The program uses password-based encryption with MD5 and DES. Users choose a password and the program generates a password for encryption.

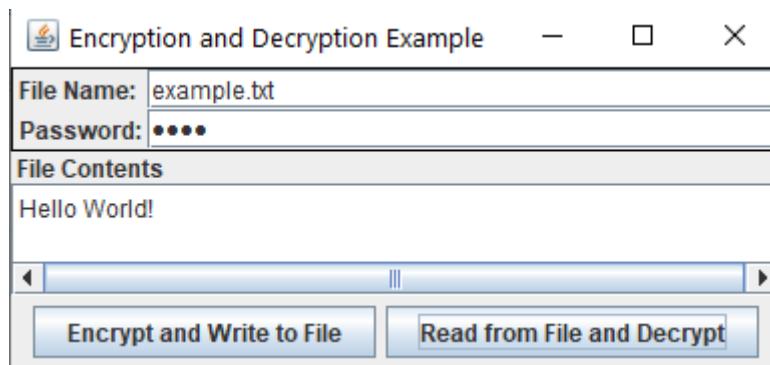


Figure 5.1. The program encrypts the text and writes to the file

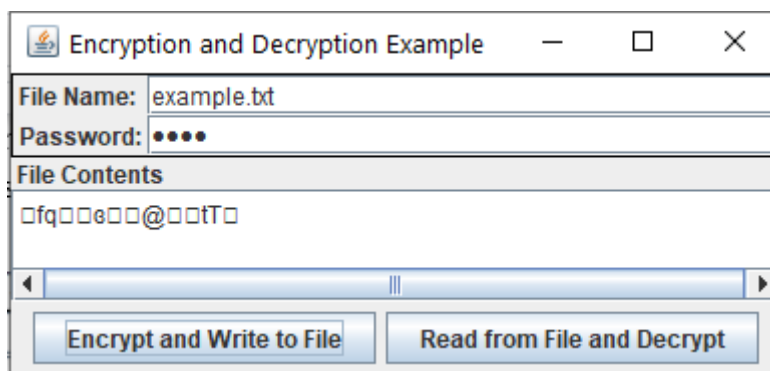


Figure 5.2. Encrypted text

2. JSSE - Java Secure-Socket Extension

This program makes use of SSL to communicate securely between client and server. We make use of certificates to establish a secure connection. JSSE is used for privacy and integrity protection.

For this task InstallCert didn't work for us. Hence, we used an alternative approach as follows:

Generating the certificate:

```
keytool -genkey -keystore yourKEYSTORE -keyalg RSA
```

For server certificate:

```
java -Djavax.net.ssl.keyStore=yourKEYSTORE  
-Djavax.net.ssl.keyStorePassword=123456  
-Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol  
-Djavax.net.debug=ssl LoginServer.java
```

For client certificate:

```
java -Djavax.net.ssl.trustStore=yourKEYSTORE  
-Djavax.net.ssl.trustStorePassword=yourPASSWORD SSLClient
```

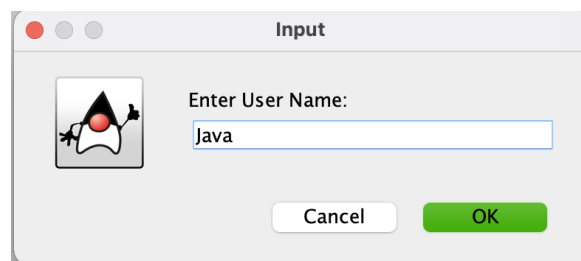


Figure 5.3. Login Screen

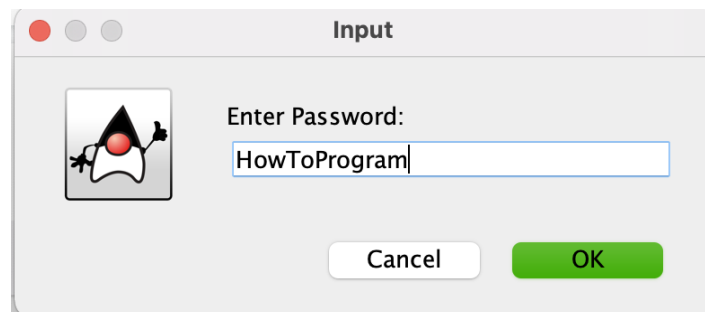


Figure 5.4. Login Screen

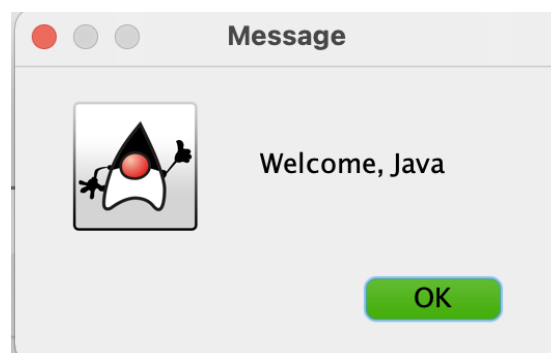


Figure 5.5. Welcome Screen