DFTP

Datagram File Transfer Protocol

Nazmul Alam

TABLE OF CONTENTS

INTRODUCTION

I would like to implement a three-tier inter-process communication using Java’s Socket API. I will be following the protocol I have designed as part of this project as my reference to implement the system.

DESIGN

OBJECTIVES AND OVERVIEW

The objective of this exercise is to design a 3-tier client server application. Using Java and Maven, I would like to implement a micro-service based architecture, and demonstrate my understanding of implementing an inter-process communication application, by following the protocol document I have had drafted.

The client application should allow the user to login and logoff from the system, as well as upload and download a file to and fro the server. For simplicity, the server is going to maintain a separate folder unique to each client. A client may upload or download files only from his/her unique folder.

The server should accommodate all the functionalities documented within the RFC 768(N) protocol document.

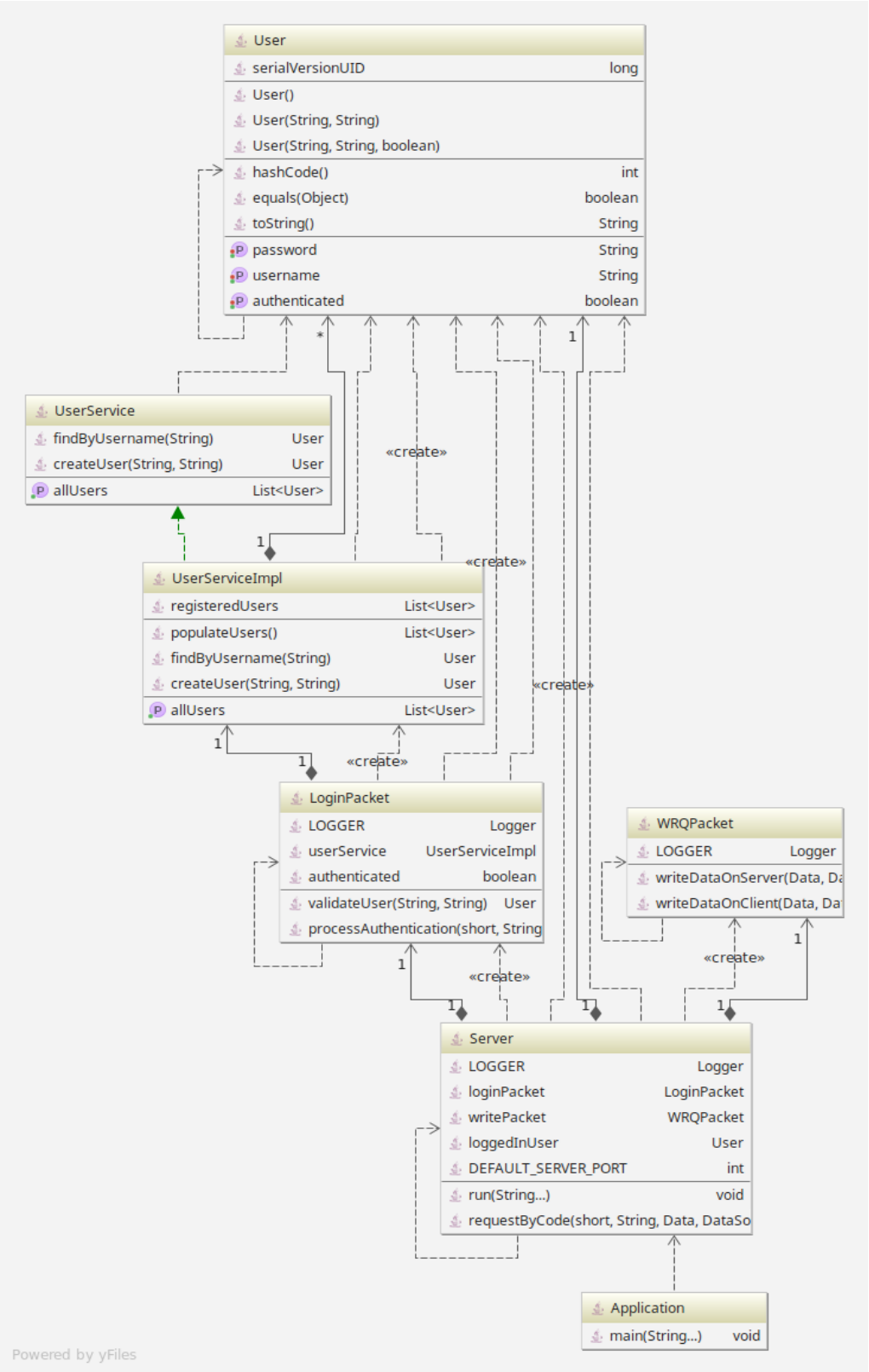
All the objectives will be achieved with DatagramSocket and Datagram Packet APIs from Java Socket API.

DESIGN PHILOSOPHY

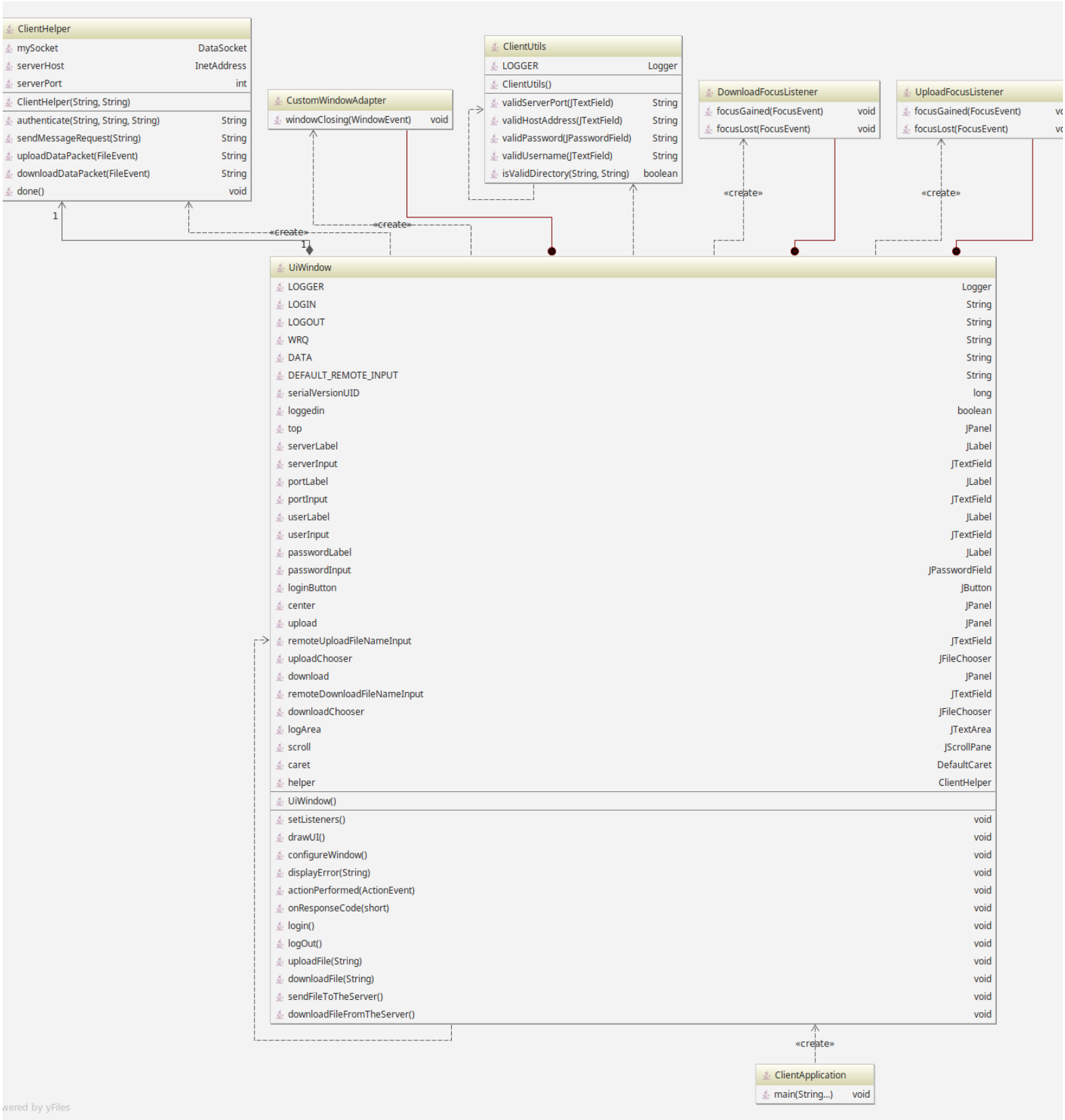
To avoid duplicated code, I am going to create a common layer. The common layer should consist of all the shared implementations. Classes, methods, and fields would be named as generic as possible to avoid code smells. I will also design the system with keeping all the Java's design patterns in mind.

- ***ftp-common:*** *the application layer for all the libraries, utility methods, and data model shared within the client and the server.*
- ***ftp-client:*** *the presentation layer for all the client logic to create the GUI to sending and receiving data. All the functionalities within the presentation layer should correlate with the protocol document*
- ***ftp-server:*** *the session layer to send and receive requests to and from the client to server or vice versa.*

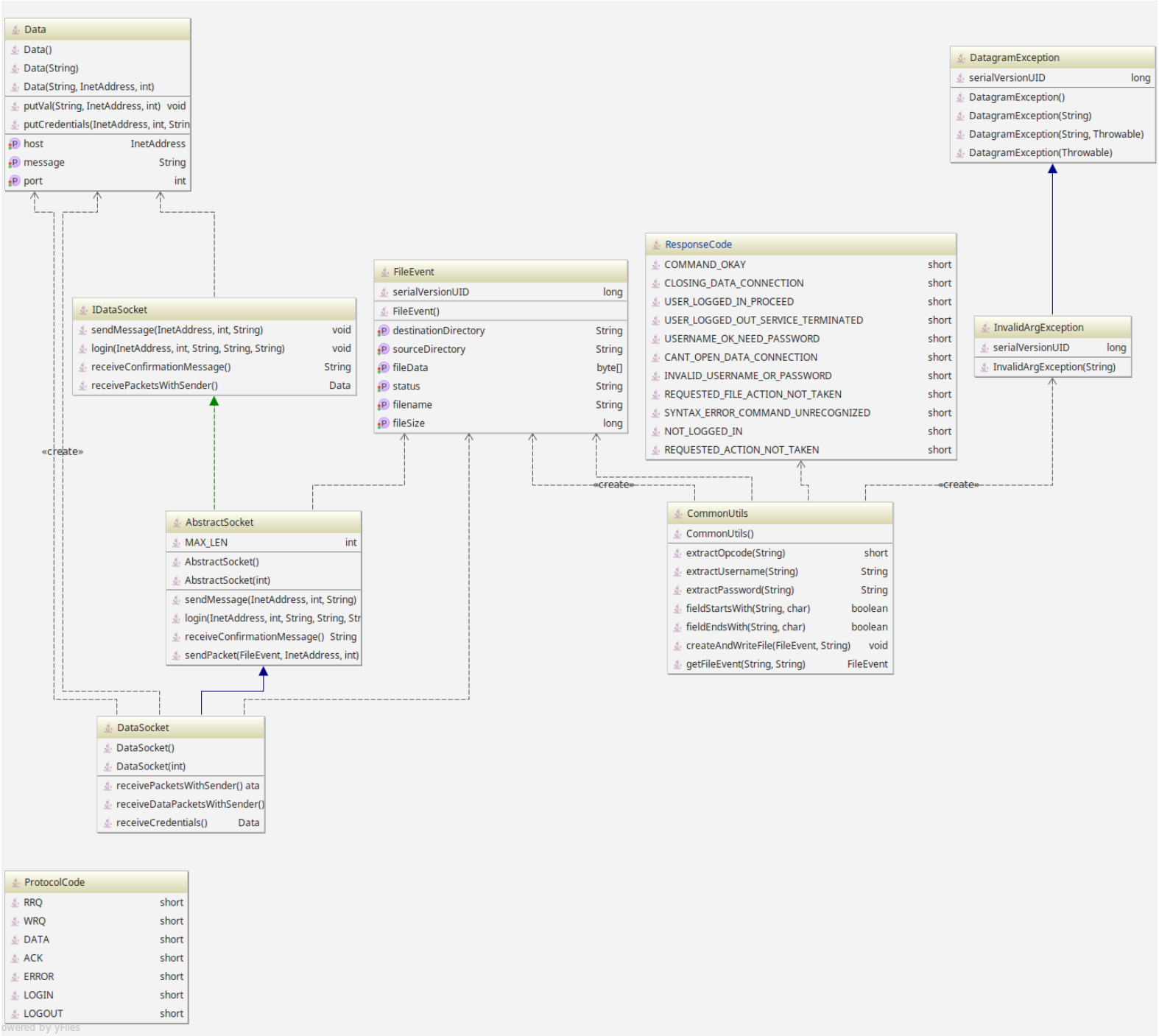
SERVER UML DIAGRAMS



CLIENT UML DIAGRAMS



COMMON LAYER UML DIAGRAMS



IMPLEMENTATION

OBJECTIVES AND OVERVIEW

As protocol in place, as well as the design for a 3-tier client-server inter-process application.

Using Datagram to send and receive files does not guarantee the delivery and duplicate protection. However, I am aiming to send a small binary file at once without breaking them into parts.

APPLICATION LAYER IMPLEMENTATION

In the Application/common layer, I have included two exception handler classes (DatagramException.java and InvalidArgException.java) within the exception package. These classes should be enough to handle all the custom exceptions.

For protocol codes, I have created ProtocolCode.java and ResponseCode.java classes, both of which are Enumeration classes to avoid magic number code smells.

ProtocolCode.java

```
public static final short WRQ = 111;

public static final short DATA = 300;

public static final short ACK = 400;

public static final short ERROR = 222;

public static final short LOGIN = 600;

public static final short LOGOUT = 700;
```

ResponseCode.java

```
public static final short COMMAND_OKAY = 200;

public static final short CLOSING_DATA_CONNECTION = 226;

public static final short USER_LOGGED_IN_PROCEED = 230;

public static final short USER_LOGGED_OUT_SERVICE_TERMINATED = 231;

public static final short USERNAME_OK_NEED_PASSWORD = 331;

public static final short CANT_OPEN_DATA_CONNECTION = 425;

public static final short INVALID_USERNAME_OR_PASSWORD = 430;

public static final short REQUESTED_FILE_ACTION_NOT_TAKEN = 450;

public static final short SYNTAX_ERROR_COMMAND_UNRECOGNIZED = 500;

public static final short NOT_LOGGED_IN = 530;

public static final short REQUESTED_ACTION_NOT_TAKEN = 550;
```

I have created inheritance based data modelling classes to handle the message sent to and from the client to server or vice versa. The aim of this interface is to follow the 'program to interface rather than implementation' paradigm.

```
void login(InetAddress host, int port, String opcode, String username, String password) throws IOException;

String receiveConfirmationMessage() throws IOException;

Data receivePacketsWithSender() throws IOException;

FileEvent receiveDataPacketsWithSender() throws IOException, ClassNotFoundException;

void sendPacket(FileEvent event, InetAddress host, int port) throws IOException;

Data receiveCredentials() throws IOException;
```

As I have already dealt with messages in the labs, sending binary packets needed different approach during implementation.

```java
@Override
public FileEvent receiveDataPacketsWithSender() throws IOException, ClassNotFoundException {

  byte[] incomingData = new byte[MAX_LEN * 1000 * 50];
  DatagramPacket incomingPacket = new DatagramPacket(incomingData, incomingData.length);
  receive(incomingPacket);
  byte[] data = incomingPacket.getData();
  ByteArrayInputStream in = new ByteArrayInputStream(data);
  ObjectInputStream is = new ObjectInputStream(in);
  return (FileEvent) is.readObject();
}
```

Finally, I have created a FileEvent.java class for data serialization and CommonUtils.class for utility methods.

```java
public static void createAndWriteFile(FileEvent fileEvent, String username)
        throws IOException, InvalidArgException {

  String destinationPath = fileEvent.getDestinationDirectory() + "/" + username;
  String outputFile = destinationPath + "/" + fileEvent.getFilename();
  if (!new File(destinationPath).exists()) {
    new File(destinationPath).mkdirs();
  }
  try {
    File dstFile = new File(outputFile);
    FileOutputStream fileOutputStream = new FileOutputStream(dstFile);
    fileOutputStream.write(fileEvent.getFileData());
    fileOutputStream.flush();
    fileOutputStream.close();
  } catch (FileNotFoundException e) {
    throw new InvalidArgException(String.valueOf(ResponseCode.REQUESTED_ACTION_NOT_TAKEN));
  }
}
```

createAndWriteFile() method takes the destination path and appends the logged in username with it. This forces the files to be written to the user's folder only.

File selected for upload and download needed to be serialized to a FileEvent object so it can be transferred.

```
73    public static FileEvent getFileEvent(String sourceFilePath, String destinationPath) {
74
75      FileEvent fileEvent = new FileEvent();
76      String fileName = sourceFilePath.substring(sourceFilePath.lastIndexOf("/") + 1, sourceFilePath.length());
77      String path = sourceFilePath.substring(0, sourceFilePath.lastIndexOf("/") + 1);
78      fileEvent.setDestinationDirectory(destinationPath);
79      fileEvent.setFilename(fileName);
80      fileEvent.setSourceDirectory(sourceFilePath);
81      File file = new File(sourceFilePath);
82      if (file.isFile()) {
83        try {
84          DataInputStream diStream = new DataInputStream(new FileInputStream(file));
85          Long len = file.length();
86          byte[] fileBytes = new byte[len.intValue()];
87          int read = 0;
88          int numRead = 0;
89          while (read < fileBytes.length && numRead >= 0) {
90            read += numRead;
91            numRead = diStream.read(fileBytes, read, fileBytes.length - read);
92          }
93          fileEvent.setFileSize(len);
94          fileEvent.setFileData(fileBytes);
95          fileEvent.setStatus("Success");
96        } catch (Exception e) {
97          fileEvent.setStatus("Error");
98        }
99      } else {
100       fileEvent.setStatus("Error");
101     }
102     return fileEvent;
103   }
104 }
```

PRESENTATION LAYER IMPLEMENTATION

Initially, I wanted to implement the GUI with JavaFX so it is future proofed. However, as I was used to implementing GUI with Swing, I later changed my mind.

I have always used FileZilla as my FTP client. My GUI was inspired from FileZilla's UI.

Login Request

I have applied the pseudocode I have had compiled in the protocol design phase as much as possible. I have tried to be as clean as possible. ClientUtils methods validates all the inputs. Once they are validated, the client a login request to the server and expects response back.

Based on the response received from the server, information is displayed in the console and GUI.

N.B. Logout request is very similar.

```
private void login() {

  String responseCode = "";
  try {
    String host = ClientUtils.validHostAddress(serverInput);
    String port = ClientUtils.validServerPort(portInput);
    String username = ClientUtils.validUsername(userInput);
    String password = ClientUtils.validPassword(passwordInput);

    helper = new ClientHelper(host, port);
    logArea.append("Status: Logging into " + host + "\n");
    responseCode = helper.authenticate(LOGIN, username, password);

  } catch (IOException | InvalidArgException io) {
    logArea.append("Status: " + io.getMessage() + "\n");
  } finally {
    // successfully logged in
    if (responseCode != null
            && responseCode
            .trim()
            .equals(String.valueOf(ResponseCode.USER_LOGGED_IN_PROCEED))) {

      loggedin = true;
      onResponseCode(Short.parseShort(responseCode.trim()));

    } else if (!responseCode.isEmpty()) {
      onResponseCode(Short.parseShort(responseCode.trim()));
    }
  }

}
```

Upload Request

```java
private void sendFileToTheServer() {

  String responseCode = "";
  try {
    String host = ClientUtils.validHostAddress(serverInput);
    String port = ClientUtils.validServerPort(portInput);
    String username = ClientUtils.validUsername(userInput);
    String password = ClientUtils.validPassword(passwordInput);

    // Send request to write data
    logArea.append("Status: Sending a request to write data\n");
    responseCode = helper.sendMessageRequest(WRQ + username + password);
    // if data write is allowed
    if (responseCode.trim().equals(String.valueOf(ResponseCode.COMMAND_OKAY))) {
      LOGGER.info(ResponseCode.COMMAND_OKAY + " Ready to upload");
      // send data
      String sourcePath = uploadChooser.getSelectedFile().getAbsolutePath();
      String destinationPath = downloadChooser.getCurrentDirectory().getAbsolutePath();
      FileEvent event = CommonUtils.getFileEvent(sourcePath, destinationPath);
      logArea.append("Status: File upload has started\n");
      responseCode = helper.uploadDataPacket(event);
    }

  } catch (InvalidArgException | IOException inval) {
    logArea.append("Status: " + inval.getMessage() + "\n");
  } finally {
    // if file was successfully uploaded
    if (responseCode != null && !responseCode.isEmpty()) {
      onResponseCode(Short.parseShort(responseCode.trim()));
    }
  }
}
```

The code above should be self-explanatory. Client initiates a call to the server for a handshake, if server responds with 200, client goes ahead and sends the data.

If anything goes wrong, exception handlers catches them and messages are displayed appropriately on the console and GUI.

Download Request

```
530  private void downloadFileFromTheServer() {
531
532    String responseCode = "";
533    try {
534      String host = ClientUtils.validHostAddress(serverInput);
535      String port = ClientUtils.validServerPort(portInput);
536      String username = ClientUtils.validUsername(userInput);
537      String password = ClientUtils.validPassword(passwordInput);
538
539
540      String curDirName = downloadChooser.getCurrentDirectory().getName();
541      String sysUsername = CommonUtils.extractUsername(username);
542      boolean validDirectory = ClientUtils.isValidDirectory(curDirName, sysUsername);
543
544      logArea.append("Status: Sending a request to download data\n");
545
546      if (validDirectory) {
547        // Send request to download data
548        responseCode = helper.sendMessageRequest(DATA + username + password);
549
550        if (responseCode.trim().equals(String.valueOf(ResponseCode.COMMAND_OKAY))) {
551          // send data source
552          String sourcePath = downloadChooser.getSelectedFile().getAbsolutePath();
553          String destinationPath = uploadChooser.getCurrentDirectory().getAbsolutePath();
554          FileEvent event = CommonUtils.getFileEvent(sourcePath, destinationPath);
555          logArea.append("Status: File download has started\n");
556          responseCode = helper.downloadDataPacket(event);
557        }
558
559      } else {
560        LOGGER.info(ProtocolCode.ERROR + " Restricted data access requested");
561        responseCode = helper.sendMessageRequest(DATA + username + password + ProtocolCode.ERROR);
562      }
563
564    } catch (InvalidArgException | IOException inval) {
565      logArea.append("Status: " + inval.getMessage() + "\n");
566    } finally {
567      // if file was successfully downloaded
568      if (responseCode != null && !responseCode.isEmpty()) {
569        onResponseCode(Short.parseShort(responseCode.trim()));
570      }
```

In download, at line 542, it checks for a valid directory. A valid directory is user's unique folder. The upload logic on the client side could be reversed, but as this is a full application, I have decided to implement it in the client. However, if time persists, I am going to move the logic to the server side.

SESSION LAYER IMPLEMENTATION

The first thing I had implement is to manage users. As, I am not required to use a database, or persist anything for this exercise, I decided to populate users in the memory at runtime.

```
12    static {
13      registeredUsers = populateUsers();
14    }
15
16    private static List<User> populateUsers() {
17
18      List<User> users = new CopyOnWriteArrayList<User>();
19      users.add(new User("admin", "admin", false));
20      users.add(new User("user", "user", false));
21      users.add(new User("demo", "demo", false));
22      return users;
23    }
```

The server runs on a default port 3000. Although, the default port for UDP is 7, it was not able to run it on my Linux distro. So, I chose the port the node server runs on.

```
while (true) {
  Data request = socket.receivePacketsWithSender();
  String message = request.getMessage();
  short opcode = CommonUtils.extractOpcode(message);

  if (loggedInUser != null && loggedInUser.isAuthenticated()) {
    loggedInUser = (User) requestByCode(opcode, message, request, socket);

  } else if (loggedInUser != null && !loggedInUser.isAuthenticated()) {
    loggedInUser = loginPacket.processAuthentication(opcode, message, request, socket);
  } else {
    socket
            .sendMessage(
                    request.getHost(),
                    request.getPort(),
                    String.valueOf(ResponseCode.CANT_OPEN_DATA_CONNECTION));
  }
```

The server keeps on listening to the localhost port 3000 and waits for a message to be received from the client. When a message is received, it extracts the operation code and performs accordingly.

When a login request is received, the server executes accordingly.

```
if (opcode == ProtocolCode.LOGIN) {
  LOGGER.info(ProtocolCode.LOGIN + " Authentication request received");
  try {
    loggedInUser = validateUser(username, password);
    dataSocket
            .sendMessage(
                    request.getHost(),
                    request.getPort(),
                    String.valueOf(ResponseCode.USER_LOGGED_IN_PROCEED));
    LOGGER.info(ResponseCode.USER_LOGGED_IN_PROCEED + " Authenticated");

  } catch (InvalidArgException exc) {
    dataSocket.sendMessage(request.getHost(), request.getPort(), exc.getMessage());
    LOGGER.info(exc.getMessage() + " Authentication unsuccessful");
  }
  return loggedInUser;
}
```

The server starts off validating the user populated at runtime, and if everything goes well, the server sends 230 response code to the client. **N.B.** The logout procedure is very similar.

When a download request is received, with an Error code 222, the server acknowledges the code, and responses with 550 code.

```
case ProtocolCode.DATA:
  //now send data to the client
  if (request.getMessage().trim().endsWith(String.valueOf(ProtocolCode.ERROR))) {
    LOGGER.warn(ProtocolCode.ERROR + " Restricted data access");
    socket
            .sendMessage(
                    request.getHost(),
                    request.getPort(),
                    String.valueOf(ResponseCode.REQUESTED_ACTION_NOT_TAKEN));
    LOGGER.warn(ResponseCode.REQUESTED_ACTION_NOT_TAKEN + " Requested action not taken");
```

However, if the request was successful with no error code was received, the server processes the data.

```
try {
  CommonUtils.createAndWriteFile(fileEvent, username);

} catch (FileNotFoundException file) {
  dataWritten = false;
  socket
          .sendMessage(
                  request.getHost(),
                  request.getPort(),
                  String.valueOf(ResponseCode.REQUESTED_FILE_ACTION_NOT_TAKEN));

  LOGGER.warn(ResponseCode.REQUESTED_FILE_ACTION_NOT_TAKEN + " Writing data was unsuccessful");
  LOGGER.warn(file.getMessage());

} catch (InvalidArgException e) {
  dataWritten = false;
  socket.sendMessage(request.getHost(), request.getPort(), e.getMessage());

  LOGGER.warn(e.getMessage() + " Writing data was unsuccessful");

} finally {
  // If file was created successfully
  if (dataWritten) {
    LOGGER.info(ResponseCode.CLOSING_DATA_CONNECTION + " Successfully written data");
    socket
            .sendMessage(
                    request.getHost(),
                    request.getPort(),
                    String.valueOf(ResponseCode.CLOSING_DATA_CONNECTION));
  }
}
```

The server uses the same implementation for an upload request as above.

USER MANUAL

BUILD AND RUN

Assuming you have the latest version of Java installed and the JAVA_HOME and PATH are set properly; please run these commands from the parent directory i.e. datagram-ftp

Windows

Build by running: mvnw clean install

First run the server: java -jar ftp-server/target/ftp-server-1.0-SNAPSHOT-jar-with-dependencies.jar

Run the client: java -jar ftp-client/target/ftp-client-1.0-SNAPSHOT-jar-with-dependencies.jar

*nix

Build by running: ./mvnw clean install

First run the server: java -jar ftp-server/target/ftp-server-1.0-SNAPSHOT-jar-with-dependencies.jar

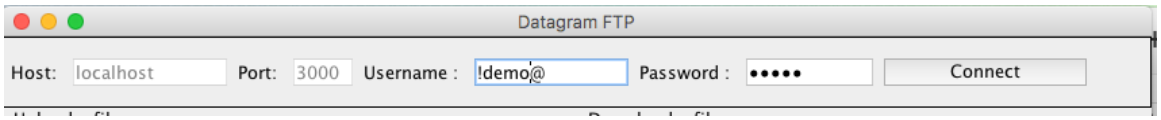
Run the client: java -jar ftp-client/target/ftp-client-1.0-SNAPSHOT-jar-with-dependencies.jar

LOGIN/LOGOUT

The application starts off with all the information prefilled. However, two other users can login to the system – admin and user. The password for each of them are the same as their username.

Login: demo

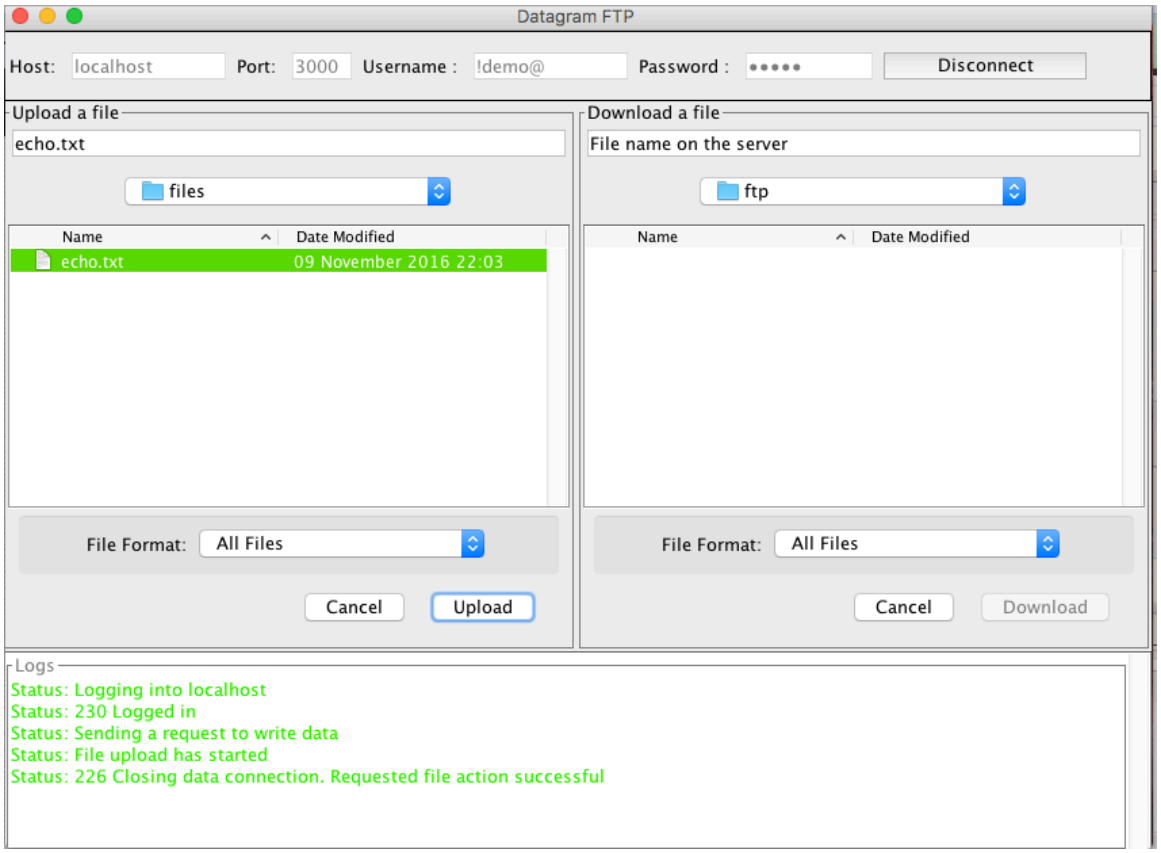
Password: demo



- *The username must start with '!' and end with '@' delimiters.*
- *The password must end with '!' delimiter.*
- *Host name and port cannot be changed before or after login*
- *Authentication fields are disabled after successful login*
- *The login button automatically behaves as a logout. The 'Connect' text changes to 'Disconnect' once logged in.*

UPLOAD

- *Upload and download buttons are disabled before login*
- *Cancel button does not do anything except printing a log information 'Upload was cancelled'*
- *Uploading a file can be selected from any directory*
- *Data size must not exceed 64 kilobytes*



When upload button is pressed, and all the conditions are satisfied, a new directory called with user name i.e. demo is created and the echo.txt file is uploaded to the user's directory.

Logs are also displayed in the console in both client and server sides. Here is an example of client side's log:

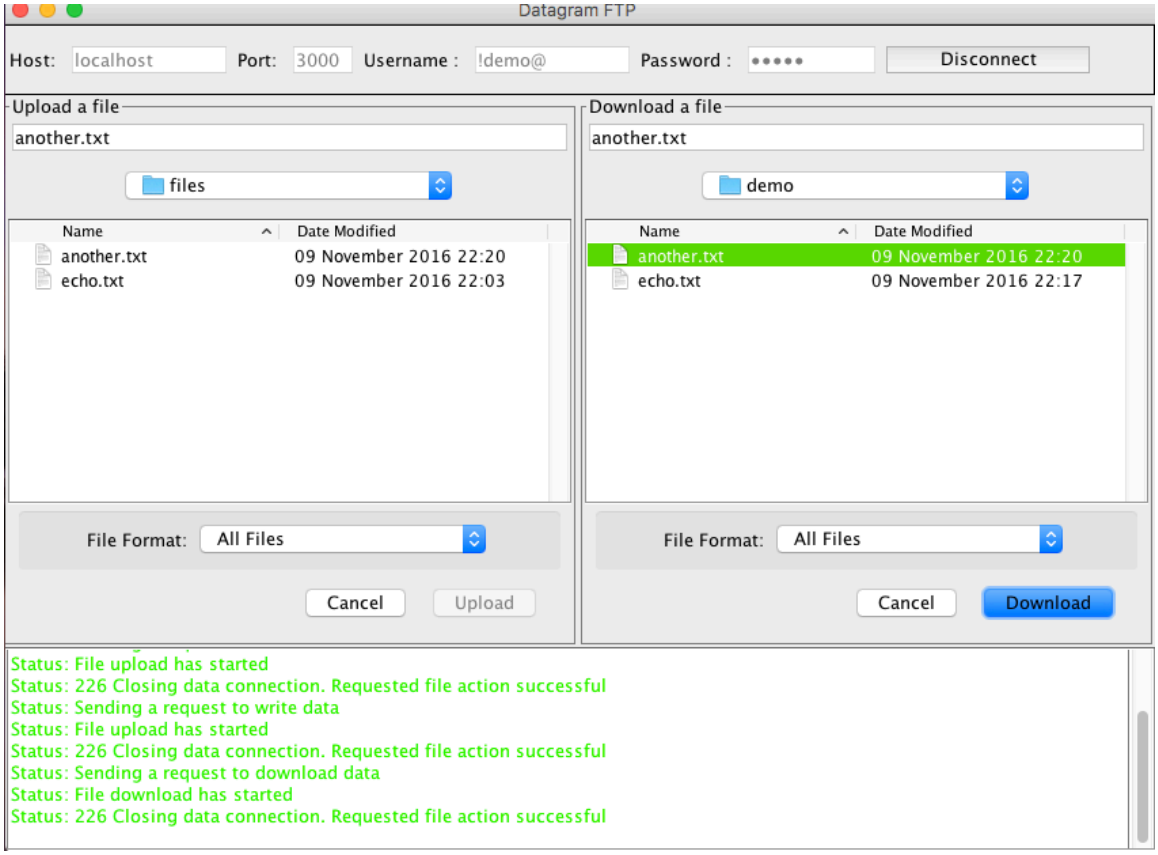
```
[Naz:~/Desktop/datagram-ftp]$ java -jar ftp-client/target/ftp-client-1.0-SNAPSHO
T-jar-with-dependencies.jar
2016-11-09 22:16:05 INFO  UiWindow:284 - Status: 600 Login request sent
2016-11-09 22:16:05 INFO  UiWindow:318 - Status: 230 Logged in
2016-11-09 22:17:23 INFO  UiWindow:459 - Status: Uploading echo.txt has started
2016-11-09 22:17:23 INFO  UiWindow:511 - Status: 200 Ready to upload
2016-11-09 22:17:23 INFO  UiWindow:344 - Status: 226 Closing data connection. Re
quested file action successful
2016-11-09 22:20:19 INFO  UiWindow:459 - Status: Uploading another.txt has start
ed
2016-11-09 22:20:19 INFO  UiWindow:511 - Status: 200 Ready to upload
2016-11-09 22:20:19 INFO  UiWindow:344 - Status: 226 Closing data connection. Re
quested file action successful
2016-11-09 22:20:46 INFO  UiWindow:490 - Status: Downloading another.txt has sta
rted
2016-11-09 22:20:46 INFO  UiWindow:344 - Status: 226 Closing data connection. Re
quested file action successful
```

And, on the server side:

```
[Naz:~/Desktop/datagram-ftp]$ java -jar ftp-server/target/ftp-server-1.0-SNAPSHO
T-jar-with-dependencies.jar
2016-11-09 22:15:54 INFO  Server:33 - Status: FTP server ready
2016-11-09 22:16:05 INFO  LoginPacket:48 - Status: 600 Authentication request re
ceived
2016-11-09 22:16:05 INFO  LoginPacket:56 - Status: 230 Authenticated
2016-11-09 22:17:23 INFO  Server:70 - Status: 111 Upload handshake received
2016-11-09 22:17:23 INFO  Server:77 - Status: 400 Acknowledgement sent
2016-11-09 22:17:23 INFO  Server:79 - Status: 111 Data upload has started
2016-11-09 22:17:23 INFO  WRQPacket:57 - Status: 226 Successfully written data
2016-11-09 22:20:19 INFO  Server:70 - Status: 111 Upload handshake received
2016-11-09 22:20:19 INFO  Server:77 - Status: 400 Acknowledgement sent
2016-11-09 22:20:19 INFO  Server:79 - Status: 111 Data upload has started
2016-11-09 22:20:19 INFO  WRQPacket:57 - Status: 226 Successfully written data
2016-11-09 22:20:46 INFO  Server:96 - Status: 300 Download handshake received
2016-11-09 22:20:46 INFO  Server:103 - Status: 400 Acknowledgement sent
2016-11-09 22:20:46 INFO  Server:105 - Status: 300 Data upload has started
2016-11-09 22:20:46 INFO  WRQPacket:57 - Status: 226 Successfully written data
```

DOWNLOAD

- *Downloading a file must be from users directory*
- *Data size must not exceed 64 kilobytes*



CONCLUSION

Designing a protocol for an inter-processing communication was fun. I am used to implementing a system, but designing a protocol for communication systems was my first experience.

This exercise allowed me to explore new areas of communication. I have taken a lot of the under-laying communications within a network for granted. After designing the protocol and implementing it, I have realised how much planning it requires to proceed with developing a three-tier system.