I decided to go with a GUI design for both the server and client. I went with this approach because I thought I would learn more and it would be easier to use. I will ignore most of the code in this write up as the GUI design and implementation is not part of the requirements.

Looking at the server code first we see

```
// MULTI-THREADING
new Thread(() -> {
  try {
     ServerSocket serverSocket = new ServerSocket(8675);
     textArea.appendText("Multi-Threaded Server Started at " + new Date() + "\n");
     try {
        while (!directorySet) {
           Thread.sleep(1000);
           // Wait for the user to set an initial working directory
        }

     } catch (InterruptedException e) {
        Thread.currentThread().interrupt();  // set interrupt flag
        System.out.println("Failed to hold thread for user input");
     }
```

A new Thread is constructed for the server and that thread is held until a current working directory is set for the server. The server code then creates a new thread for each user.

```
// Create a new thread for each connection
new Thread(new HandleAClient(socket)).start();

// Thread class for handling new client connections
class HandleAClient implements Runnable {
  private Socket socket;
  private boolean clientConnected = true;

  // construct a thread
  public HandleAClient(Socket socket) {
     this.socket = socket;
  }

  // run a thread
  public void run() {
     try {
```

In the run method we perform all actions required for that thread (client). Additional clients will cause the server to construct additional threads. Overall, the server is approximately 470 lines of code and the client is about 550 lines of code.

The following algorithm is how the server distinguishes client commands.

```
private String whatDoesTheClientWant() {
  String actionToPerform = "";

  if (clientCommand instanceof String && clientCommandString.startsWith("/") && isValidDirectory(clientCommandString))
     actionToPerform = "updateCWD";
  else if (clientCommand instanceof String && clientCommand.equals("index"))
     actionToPerform = "showFiles";
  else if (clientCommand instanceof String && clientCommand.equals("Quit"))
     actionToPerform = "quit";
  else if (clientCommand instanceof String && isInteger(clientCommandString))
     actionToPerform = "sendDirectory";
  else if (clientCommand instanceof String && clientCommandString.endsWith("$"))
     actionToPerform = "createDirectory";
  else if (clientCommand instanceof String && clientCommandString.endsWith("#"))
     actionToPerform = "removeDirectory";
  else if (!(clientCommand instanceof String))
     actionToPerform = "overwriteFile";

  return actionToPerform;
}
```

As you can see I appended special characters to the end of commands on the client side so that the server could determine specific user requests. Below is the client code for creating a directory you can see the “$” is appended without the client's knowledge.

```
create.setOnAction(e -> {
  if (connected) {
     String directoryToCreate = PopUpWindows.create();
     if (isInteger(directoryToCreate)) {
        // user did not want to create a new directory
        textArea.appendText("Directory creation cancelled\n");
     }
     else {
        // user clicked yes end the string with a dollar sign
        // so we can distinguish this command from others
        String tmp = directoryToCreate + "$";

        command = tmp;
        writeToServer();
        Object reply = readFromServer();
        textArea.appendText((String) reply + "\n");
     }
  }
  // not connected
  else {
     textArea.appendText("Please connect to the server:\n");
  }
});
```