

To: CPE 400.600.1001

From: Ryan Dahan, Jason Liang, Denielle Oliva

Date: May 2, 2022

Subject: Implementation of a File Transfer Application Technical Report

## **Functionality**

### **Two files:**

Client.java

Server.java

### **Client.java Classes:**

Client

### **Client Functions:**

main

sendFile

sendAdler32

concurrentSend

### **Server.java Classes:**

Server

### **Server Functions:**

main

receiveFile

receiveAdler

## **Introduction**

There exist two files, Client.java and Server.java which will hold our client java code and server java code respectively. Each one will be run in a different terminal to simulate a client and server communicating with each other. To properly simulate a connection, the server must be run first followed by the client.

### **Client.java main**

Within Client.java's main, it starts with a command-line check if the correct number of arguments is provided. These checks do not include a check if the source folder path can be opened due to the folder check already existing within our concurrentSend function. If no arguments are provided, it will print "Error: source folder path not provided", if 1 argument is provided, it will set the source file path to the one specified by the user and the concurrency pack

size to 1, if 2 arguments are provided and the concurrency file pack size is within the size limit of 1 to 10, then it will set the source file path to the one specified by the user as well as setting the concurrency pack size to the user-defined size. If the arguments are 3 or more, it will print “Error: number of arguments exceed 2” and the final else statement will output “Error: concurrency file transfers must be between 1 and 10” to catch an incorrect concurrency pack size declaration.

The file will then output the source file path and concurrency pack size to the terminal and attempt to try the local host port for file transfer through a try and catch statement. If it successfully connects to the localhost 5000 socket, it will attempt to open the input and output files as well as run the concurrentSend function. If it does not successfully connect, then it will output an exception message. The main will then exit once this is completed.

### **Client.java concurrentSend**

ConcurrentSend is where the file directory is opened and calculations are made on how many connections will be needed for the file transfer. Once the calculations are completed, the files will then be transferred in packets until all the files are transferred through the usage of the sendFile function. Unfortunately, multithreading requires the usage of declaring a new function with Thread object manipulation and we were unable to fully implement this due to how far integrated our current file output function is. What we were able to do is to enforce a system that creates packets and enforces pseudo-multithreading where files will be sent together in packets and will wait until the previous packet is completed. If there are no files within the directory then an error message will be outputted stating that the directory is empty. This function will return the number of connections back to main for output.

### **Client.java sendFile**

SendFile opens up the file path and passes the file contents as long as the buffer limit is not hit. This is to ensure that we are not overriding any other memory blocks. It will also create a checksum for each file through the usage of the function sendAdler32. Once the file transfer is complete it will close the file. This function only completes file transfers one at a time and returns nothing. There is no error catch in this function due to the fact that this function will only run if the concurrentSend function catches no errors through its error catch.

### **Client.java sendAdler32**

SendAdler32 is a simple function and creates a checksum through the usage of the Adler32 import. It runs a try and catch statement on if the file can be assigned a checksum and will output an exception message if an error occurs and the file cannot be given a checksum. The function returns nothing and will complete after this try and catch statement is completed.

### **Server.java main**

Within Server.java's main, it will attempt to immediately connect to the port 5000 and will print out a message stating that the server is now listening for a client. This is done in a try and catch statement and will output an exception message if an error is detected. If a connection is established with a client, then it will print out a message stating that a connection is established along with what socket, address, port, and local port it is connected to. File input and output variables will be assigned and the program will read in the number of files that it is attempting to accept. Once it reads these values in, the receiveFile function is run. The files are then closed and the main file exits.

### **Server.java receiveFile**

ReceiveFile opens up both the input and output files and will first create checksum variables before beginning the file transfer. The file transfer will be run until either all the files are transferred or if the buffer or file size restriction is broken. Once either of these cases are hit, the function will then close the file and end returning nothing.

### **Server.java receiveAdler**

ReceiveAdler runs a simple try and catch statement that reads in the original checksum from the client as well as the checksum created within the server. If they match, it will output "Verification complete: file integrity secure" but if they do not match, it will output "Verification complete: file integrity compromised." If any errors occur within this step, the catch will output an exception message detailing the error.

### **Conclusion**

In regards to the number of concurrency file transfer rates the user can enter, we limited it from 1 to 10 as too many concurrent files within a package may lead to a higher chance of dropping files. If no concurrency file transfer rate is specified, then the rate will default to 1. Once the concurrentSend and receiveFile functions are complete in client and server respectively, both programs will exit and the file transfer will be complete. Any errors will have been printed out to the terminal and confirmations will also be printed out.

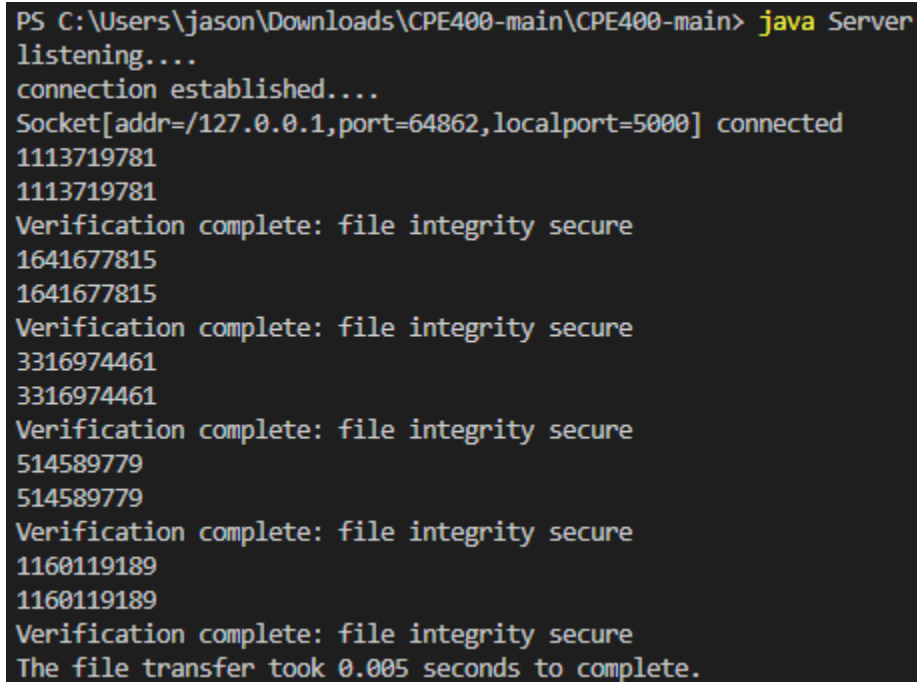
### **Novel Contribution**

#### **File Transfer Timer**

We implemented a file transfer time tracker at the end of the file transfer to inform the user of the rate it takes to send or receive files through this system. This gives them a better estimate of how much time a file transfer will take and can adjust their usage accordingly. It requires the java.util.concurrent.TimeUnit import to function properly and essentially takes the start time when a client connects to the server and subtracts it from the end time when all the file transfers are complete. This time is calculated in milliseconds and to simply convert it to seconds we

divide by 1000. Since our data type is a double, it supports decimals and will output the time it took in seconds even if it is less than 1 second.

Screenshot Example:



```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> java Server
listening....
connection established....
Socket[addr=/127.0.0.1,port=64862,localport=5000] connected
1113719781
1113719781
Verification complete: file integrity secure
1641677815
1641677815
Verification complete: file integrity secure
3316974461
3316974461
Verification complete: file integrity secure
514589779
514589779
Verification complete: file integrity secure
1160119189
1160119189
Verification complete: file integrity secure
The file transfer took 0.005 seconds to complete.
```

### Adler32 Implementation

Usage of Adler32 for our checksum to reduce the number of lines of code we need and increase code readability while keeping run time performance the same. The following screenshots will show the implementation we chose, as well as a different implementation that does not use Adler32.

Screenshot 1 (Our implementation with Adler32):

```
113 // sendAdler32 function
114 // Creates a checksum for the file
115 private static void sendAdler32(byte[] buffer)
116 {
117     // Checksum variable declaration
118     Checksum checksum = new Adler32();
119     checksum.update(buffer, 0, buffer.length);
120
121     // Grabs a checksum for the file
122     try{
123         outputStream.writeLong(checksum.getValue());
124     }
125     // Error catch to prevent bugs if response was not expected
126     catch(Exception e)
127     {
128         e.printStackTrace();
129     }
130 }
```

Screenshot 2 (Byron K's Implementation with MessageDigest):

```
08 public static void main(String args[]) throws Exception {
09
10     String filepath = "C:\\Users\\nikos7\\Desktop\\output.txt";
11
12     MessageDigest messageDigest = MessageDigest.getInstance("SHA1");
13
14     FileInputStream fileInput = new FileInputStream(filepath);
15     byte[] dataBytes = new byte[1024];
16
17     int bytesRead = 0;
18
19     while ((bytesRead = fileInput.read(dataBytes)) != -1) {
20         messageDigest.update(dataBytes, 0, bytesRead);
21     }
22
23
24     byte[] digestBytes = messageDigest.digest();
25
26     StringBuffer sb = new StringBuffer("");
27
28     for (int i = 0; i < digestBytes.length; i++) {
29         sb.append(Integer.toString((digestBytes[i] & 0xff) + 0x100, 16).substring(1));
30     }
31
32     System.out.println("Checksum for the File: " + sb.toString());
33
34     fileInput.close();
35
36 }
```

Courtesy of: Byron Kiourtzoglou

[<https://examples.javacodegeeks.com/core-java/security/messagedigest/generate-a-file-checksum-value-in-java/>]

As we can see, the Adler32 implementation requires around 17 lines of code while the message digest implementation will require around 28 lines. The usage of the Adler32 saves us roughly 11 lines of code while also containing a try and catch statement to catch errors and display an error message as needed which the message digest implementation does not contain. Adler32 also contains mainly single-line commands, while the message digest contains a while and for loop within the function, showing a much simpler complexity for Adler32 which further saves time.

### Metapacket Usage

Furthermore, we discovered the names of files are not kept when received on the other end. Therefore, we decided to use what we are calling a “metapacket” which gives information about the file name, size, and anything else that may be necessary. This technique allows implementations to be unique and invite a broader audience to its use.

Screenshot 3 (Our implementation of metapacket)

```
101      outputStream.writeUTF(file.getName());
102      outputStream.writeLong(file.length());
103
104      // Sets buffer
105      byte[] buffer = new byte[4*1024];
106
107      // While the buffer is not hit, transfer file contents
108      while((bytes=fileIn.read(buffer))!=-1){
109          outputStream.write(buffer, 0, bytes);
110          outputStream.flush();
111          sendAdler32(buffer);
112      }
```

In our implementation, lines 101 and 102 send information regarding the file length and name. The position of the Adler32 computation makes it convenient to use the byte array that has been extracted from the file exactly before it. The sendAdler can be a part of the “metapacket” information in a future implementation if the byte array were initialized earlier which would benefit through readability at the cost of flow.

## **Results & Analysis**

### **Command line argument results:**

#### **Case 1: No given command line arguments in client**

Case 1 represents the scenario where no command-line arguments were given to the client while the server response is listening or not listening. Both will result in an error message being printed in the client terminal stating “Error: source folder path not provided.” A minor issue with this response is that the server will continue to stay open and listen until the user closes out of the server terminal. We could implement a timeout function within the server terminal to fix this if needed.

Command line client response

```
C:\Users\jason\Downloads\CPE400-main\CPE400-main>java Client
Error: source folder path not provided
```

Command line server response

```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> java Server
listening....
```

#### **Case 2: File name command line argument given along with server listening**

Case 2 represents the scenario where a file name command-line argument is given but no concurrency pack size is specified. It will default to a concurrency pack size of 1 and then output the name of the source file as well as the concurrency pack size. The program will then start sending packets if the server connects. For the user’s convenience, we will output the packet numbers as well as file numbers to ensure that the packets and files are sent in the correct order.

On the server-side of things, we have the standard listening output once the server is online and waiting for a connection as well as the connection established output once we run the client and the server connects. It will then output the socket address, port, and localport along with the original and new checksum when the files are transferred. Taking a look at the file1 within the sendFolder along with the new file1 created from the server, we can see that the file contains the same contents.

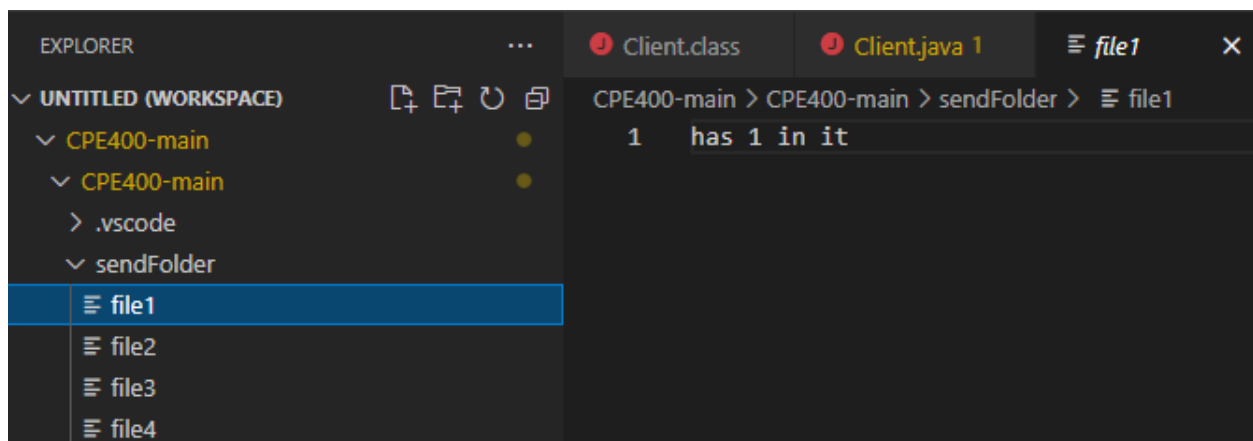
## Command line client response

```
C:\Users\jason\Downloads\CPE400-main\CPE400-main>java Client sendFolder
sendFolder | 1
Sending packet number: 1
Sending file number: 1
Sending packet number: 2
Sending file number: 2
Sending packet number: 3
Sending file number: 3
Sending packet number: 4
Sending file number: 4
```

## Command line server response

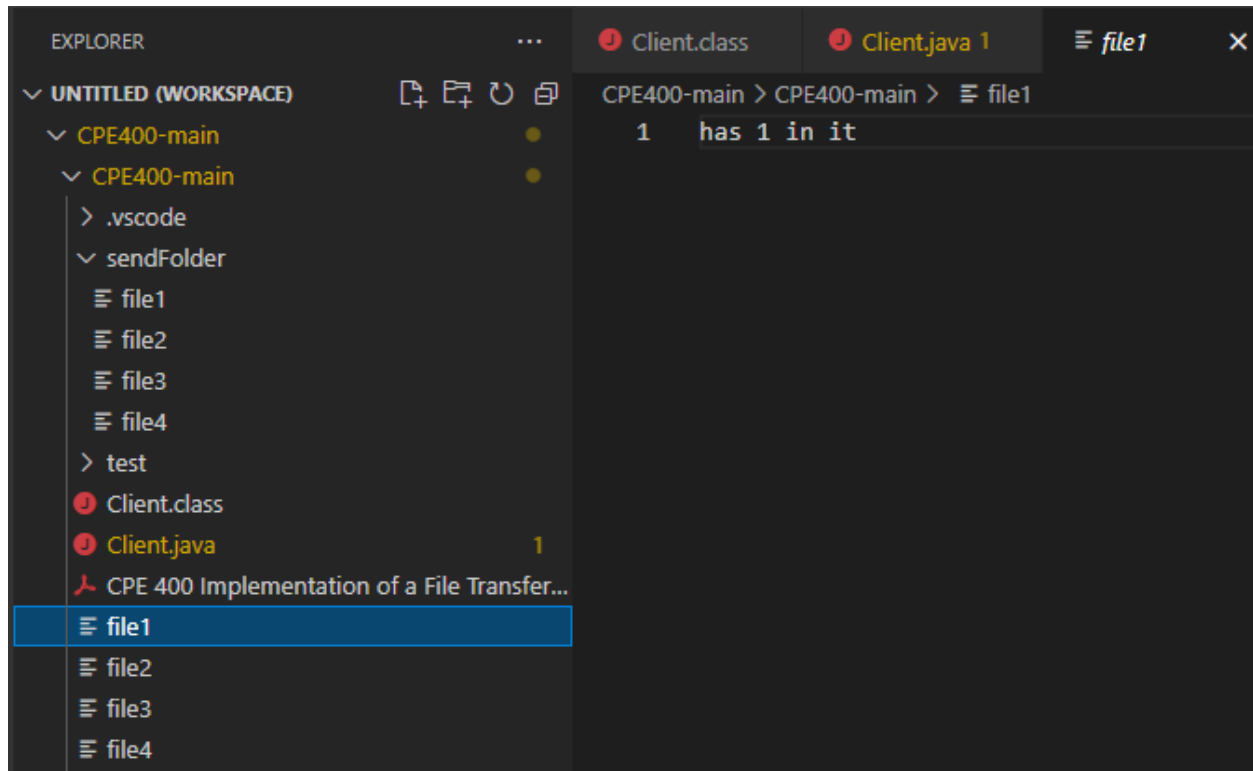
```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> java Server
listening....
connection established....
Socket[addr=/127.0.0.1,port=64823,localport=5000] connected
294388610
294388610
Verification complete: file integrity secure
562561923
562561923
Verification complete: file integrity secure
830735236
830735236
Verification complete: file integrity secure
1098908549
1098908549
Verification complete: file integrity secure
The file transfer took 0.002 seconds to complete.
```

## sendFolder contents





server output files



### Case 3: Command line client response when given file name and server is not listening

Case 3 occurs in the scenario where the client attempts to connect to a server that is not listening. It will output the same response regardless of if a source file name is given or if a source file name and a concurrency file pack size are given. It will simply output the source file name and concurrency file pack size followed by a connection refused error message. An additional feature we could add to clarify the output message to the user is to add a label above the sendFolder and 1 to signify that it represents the source file and concurrency file pack size.

Command line client response

```
C:\Users\jason\Downloads\CPE400-main\CPE400-main>java Client sendFolder  
sendFolder | 1  
java.net.ConnectException: Connection refused: connect
```

Command line server response

```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> 
```

#### Case 4: Command line client response when given file name and concurrency file packet size

Case 4 represents the scenario where we specify both the source file name and the concurrency file packet size between 1 and 10. In our example, we input the source file test with a concurrency file packet size of 2 which resulted in 3 packets being sent, 2 files being sent each time until the final packet where only 1 file was sent. This is the result we are looking for as there were 5 total files in the test file, and each file contains the same contents. An improvement that we could make here is to place all of the server files into their own folder for organization purposes or ask the user to specify a folder to put them in.

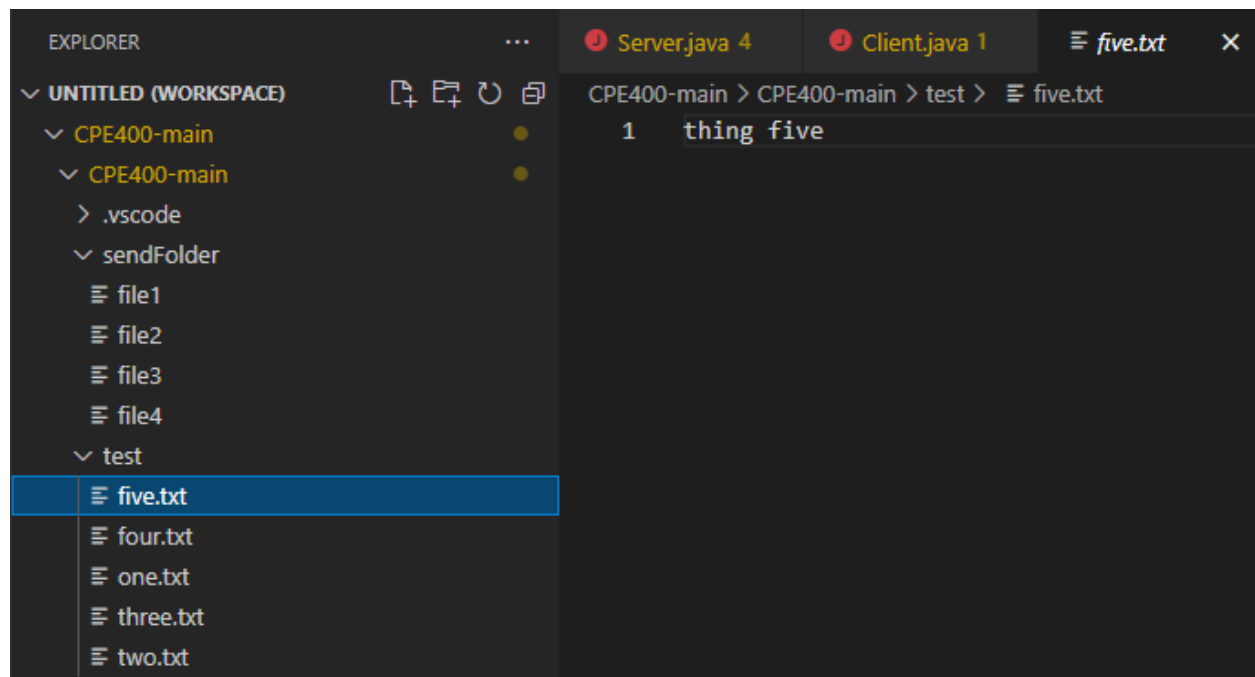
##### Command line client response

```
C:\Users\jason\Downloads\CPE400-main\CPE400-main>java Client test 2
test | 2
Sending packet number: 1
Sending file number: 1
Sending file number: 2
Sending packet number: 2
Sending file number: 3
Sending file number: 4
Sending packet number: 3
Sending file number: 5
```

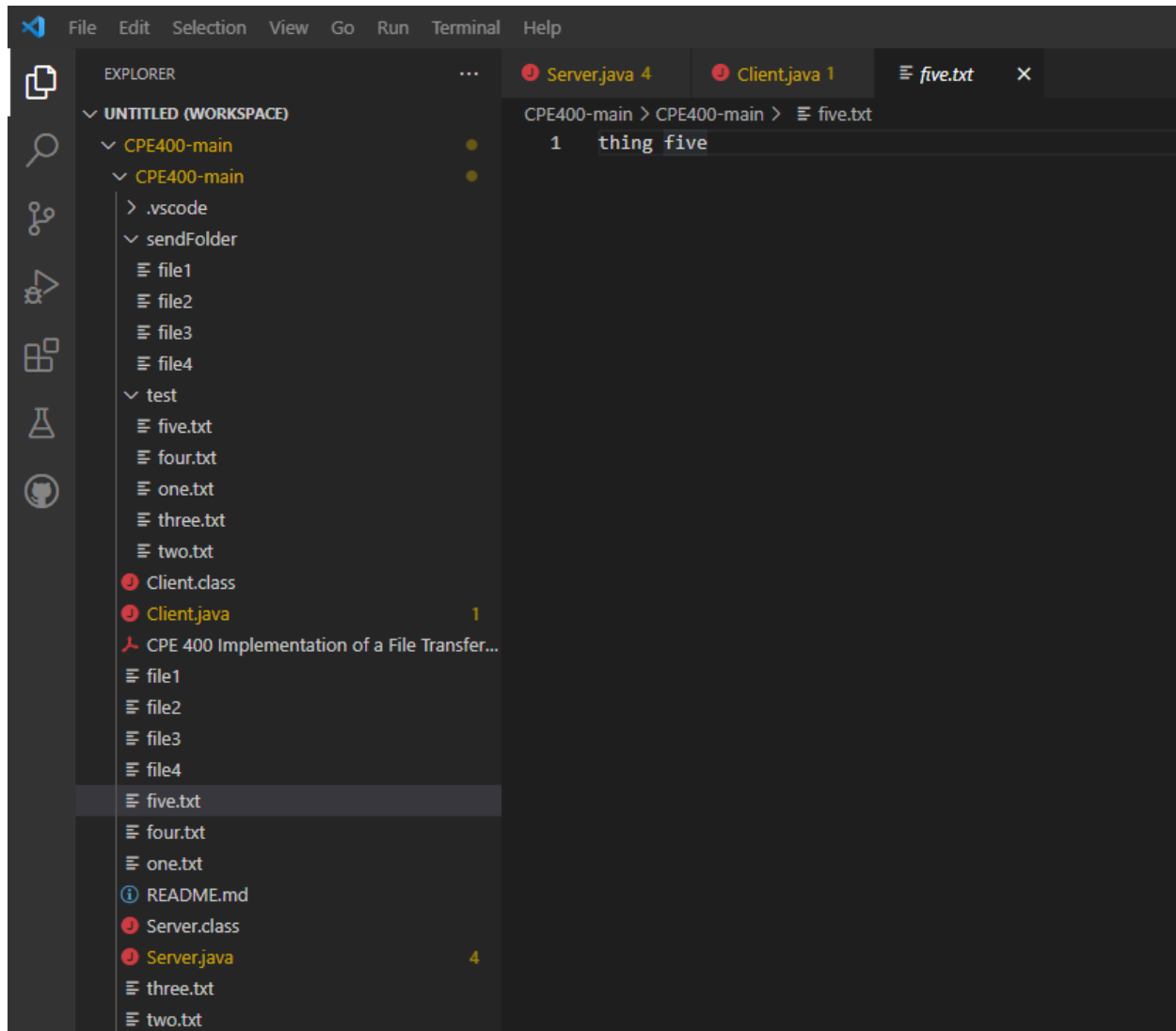
##### Command line server response

```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> java Server
listening....
connection established....
Socket[addr=/127.0.0.1,port=64862,localport=5000] connected
1113719781
1113719781
Verification complete: file integrity secure
1641677815
1641677815
Verification complete: file integrity secure
3316974461
3316974461
Verification complete: file integrity secure
514589779
514589779
Verification complete: file integrity secure
1160119189
1160119189
Verification complete: file integrity secure
The file transfer took 0.005 seconds to complete.
```

test contents



server output files



### Case 5: Command line client response when given file name and a concurrency file size that is too large

Case 5 represents the scenario where the user enters a source folder name as well as a concurrency file size, but the size given is too large or too small. It will simply print an error message stating: “Error: concurrency file transfers must be between 1 and 10.” This will output the same message regardless of whether the server is listening or not due to it only interacting with the command line of the client terminal. It is also a built-in security feature as it will not connect to an open server without having the correct parameters, preventing the waste of server resources or port usage until the client is ready.

Command line client response

```
C:\Users\jason\Downloads\CPE400-main\CPE400-main>java Client sendFolder 11
Error: concurrency file size transfers must be between 1 and 10
```

Command line server response

```
PS C:\Users\jason\Downloads\CPE400-main\CPE400-main> java Server
listening....
□
```