

Grand Valley State University

Project 1 Report

Colton Bates, Matthew Kennedy, Parker Skarzynski, Noah Verdeyen

CIS 457 10

Professor El-Said

October 16, 2018

Introduction

The object of this project was to build a multi-threading, text-based FTP client & server application. The FTP server allows multiple clients to connect and interact with the server simultaneously. The main features of the application include listing the files on the server's working directory, retrieving files from the server directory, and retrieving files from the client, and storing them on the server. The application makes use of 2 TCP connections—one for the control messages (which are the commands) and another for the data transfer.

The language-of-choice for this application was Java because it easily handles multithreading, socket programming, and input and output streams. The program has three main Java files—FTPClient.java, FTPServer.java, and FTPServerThread.java. The FTP client is a basically a main method that creates a control connection, gets user input commands, and creates and closes data connections as necessary. The FTP server continuously accepts client connections. For each client, an FTPServerThread object is created. The run method of the thread is where most of the server logic resides

The following commands are implemented:

list - The client sends this command to the server, and receives a list of the files on the server's working directory.

retr <filename> -The client sends this command with a file name to the server, and the server reads the file in as bytes to an array, which is then sent to the client. The client then writes the array of bytes to a file with a FileOutputStream. A JDialogChooser is used here on the client side just to determine the directory to save the file in. If nothing is selected, the program defaults to the client's working directory.

stor - The client sends this command to the server, and a JFileChooser is opened. If the client does not select a file, then no command is sent. If the client selects a file, then the file is read in as an array of bytes. This array of bytes is then sent to the server, which writes the array of bytes to a file in the server's working directory.

quit - Closes the client connection and stops the client application. The control socket is closed on the server side, and the FTPServerThread application is stopped for that specific connection.

Program Logic and Implementation

1. Client

a. User Arguments

When the client program starts, there are no command line arguments. Instead, it waits for a user to enter input through the console, then puts the input into a StringTokenizer object. In hindsight, this is slightly inconvenient, although not really an issue, as a string tokenizer parses a string based on the default space delimiter and doesn't provide the user with access to a size. After input is received and parsed, the client will try to connect with the provided arguments to the server. If the port is < 1024, then the port will default to 11230, as per project guidelines (Last 4 of G# + 10000).

b. Connection to Server

If the first argument is "connect" and the last argument is an integer, then the client will attempt to connect to a socket with the given information. This creates the Control Socket that the client and server use to send commands to each other.

c. Client Commands

Once the control socket is created and the user is successfully connected, the client application enters an infinite loop where the client can enter any of the commands listed above until the "quit" command is entered.

- i. If the user enters "list", the server will gather a list of files (NOT directories) in the working directory of the server and send this list to the client.
- ii. If the user enters "stor", a JFileChooser prompt will appear to select a file. If no file is chosen, no command is sent. If a file is chosen, then the client reads the file in through a FileInputStream as bytes, reads the bytes into an array, sends this over the DataOutputStream to the server, where the server reverses the process. This is slightly different than project guidelines, but follows the same principle - using bytes instead of lines allows any file to be sent instead of just text files.
- iii. If the user enters the "retr" <filename> command, then the process is the same as the stor command, just reversed: the server searches for the file requested, reads it in as bytes, then sends it to the client.
- iv. If the user enters the "quit" command, then the client closes all sockets and streams, then tells the server thread to do the same.

2. Server

a. Server Class

This class does not do much, functionality-wise. The only purpose this class has is to accept new connections from clients on the bound port. This object will enter

an infinite loop where the welcome socket will use the **accept()** method to block until a new connection is received. After that, an instance of the runnable **FTPServerThread** is created so that multiple clients can be handled.

b. FTPServerThread run() Method

The main server logic is done in the run method. At the start of the run method, the server uses the **readLine()** method from the control socket's `InputStreamReader` to block until the client sends a command. Once a message is received, the client command is parsed. The first argument is always the port, unless the "quit" command is sent. The method uses an `ArrayList` to hold the arguments given. If the command is list, retr, or stor, then a data connection is made.

- i. If the command is "list", the server will gather a list of files (NOT directories) in the working directory of the server and send this list to the client.
- ii. If the command is "stor", the server will wait until the client sends the file name with the **writeUTF()** command. After the filename is received, the server reads in the file as bytes from the client, then writes it to the working directory of the server. Note: because we are using a `FileOutputStream` to create the file, there is no need to check if the file already exists, as the `FileOutputStream` will automatically override the file if it does already exist.
- iii. If the command is "retr", the server will check for a file with the given filename. If there is no file supplied, or the file doesn't exist, then a status code of 550 is sent to the client, which will terminate the data connection before any data is sent. If the file is found, the status code is set to 200, and the file is sent as bytes from the server to the client.
- iv. If the command is "quit", then the control socket is closed as well as the input and output stream objects, and the loop variable is set to false.

Resolving Issues

This project has successfully implemented all requirements, as well as improved on some, such as being able to send any file type. Additionally, multithreading was successfully implemented in the application. The multithreading lab helped to confirm the code we had already written for this project.

One issue that was faced early on was trying to read multiple lines of data from the server in the “list” command. The **readUTF()** method on the client was executing faster than the **writeUTF()** method on the server. The way this was solved was utilizing the **available()** method, which determines if there are any bytes of data available to be read.

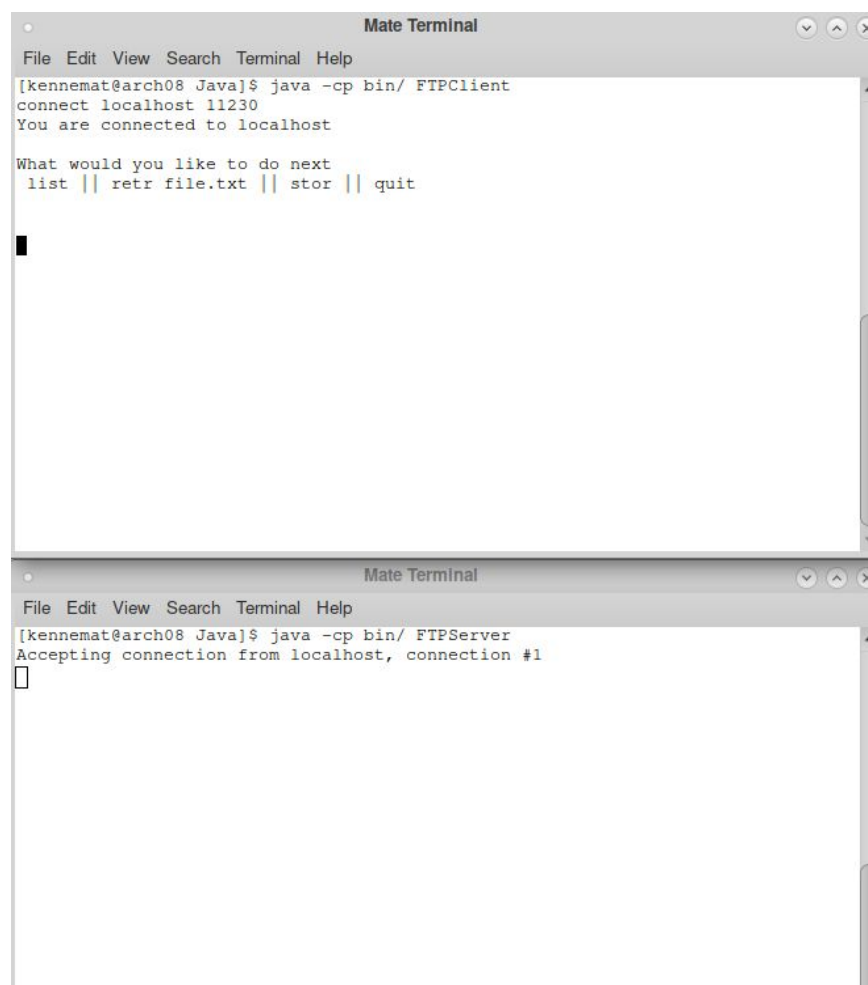
Another issue we faced was the quit command, and how to cleanly exit the server thread and the client. Because both client and server use blocking methods to wait for input (client blocks waiting for input, server blocks waiting for client), quitting could sometimes cause an exception to be thrown instead of closing resources and exiting. To get around this, we utilized if statements to verify if the input in the server was null from the client. Additionally, we used a try-catch block on the client to check if the connection was still open with the server. This way, there are no errors being pasted to the terminal.

One last issue we faced was writing bytes to an OutputStream and getting an exception. This resulted from the OutputStream object trying to close before all bytes were read out. The way we solved this was implementing the **flush()** method on the OutputStream.

Overall, the project took roughly 10-15 hours to complete. This includes all time to write code, research solutions, testing, and write this report.

Screenshots

Connect:

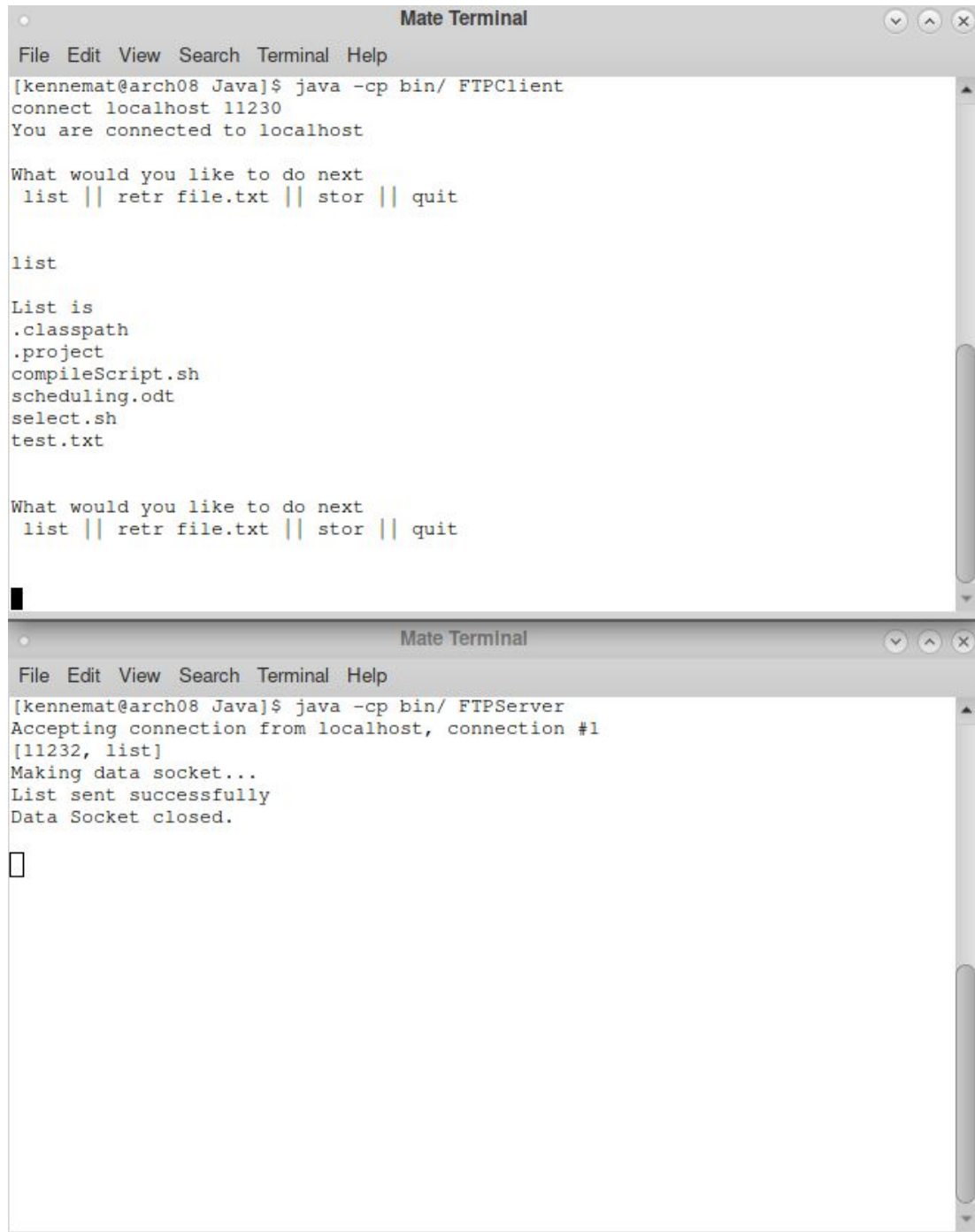


```
Mate Terminal
File Edit View Search Terminal Help
[kennemat@arch08 Java]$ java -cp bin/ FTPClient
connect localhost 11230
You are connected to localhost

What would you like to do next
list || retr file.txt || stor || quit

[kennemat@arch08 Java]$ java -cp bin/ FTPServer
Accepting connection from localhost, connection #1
```

List:



The image shows two terminal windows from the Mate desktop environment. The top window, titled 'Mate Terminal', shows the execution of the FTPClient application. It starts with the command 'java -cp bin/ FTPClient', followed by 'connect localhost 11230'. The application responds with 'You are connected to localhost' and a menu: 'What would you like to do next' with options 'list || retr file.txt || stor || quit'. The user enters 'list', and the application outputs a list of files: '.classpath', '.project', 'compileScript.sh', 'scheduling.odt', 'select.sh', and 'test.txt'. The bottom window, also titled 'Mate Terminal', shows the execution of the FTPServer application with the command 'java -cp bin/ FTPServer'. It outputs 'Accepting connection from localhost, connection #1' and '[11232, list]'. When the client sends the 'list' command, the server responds with 'Making data socket...', 'List sent successfully', and 'Data Socket closed.'.

```
[kennemat@arch08 Java]$ java -cp bin/ FTPClient
connect localhost 11230
You are connected to localhost

What would you like to do next
list || retr file.txt || stor || quit

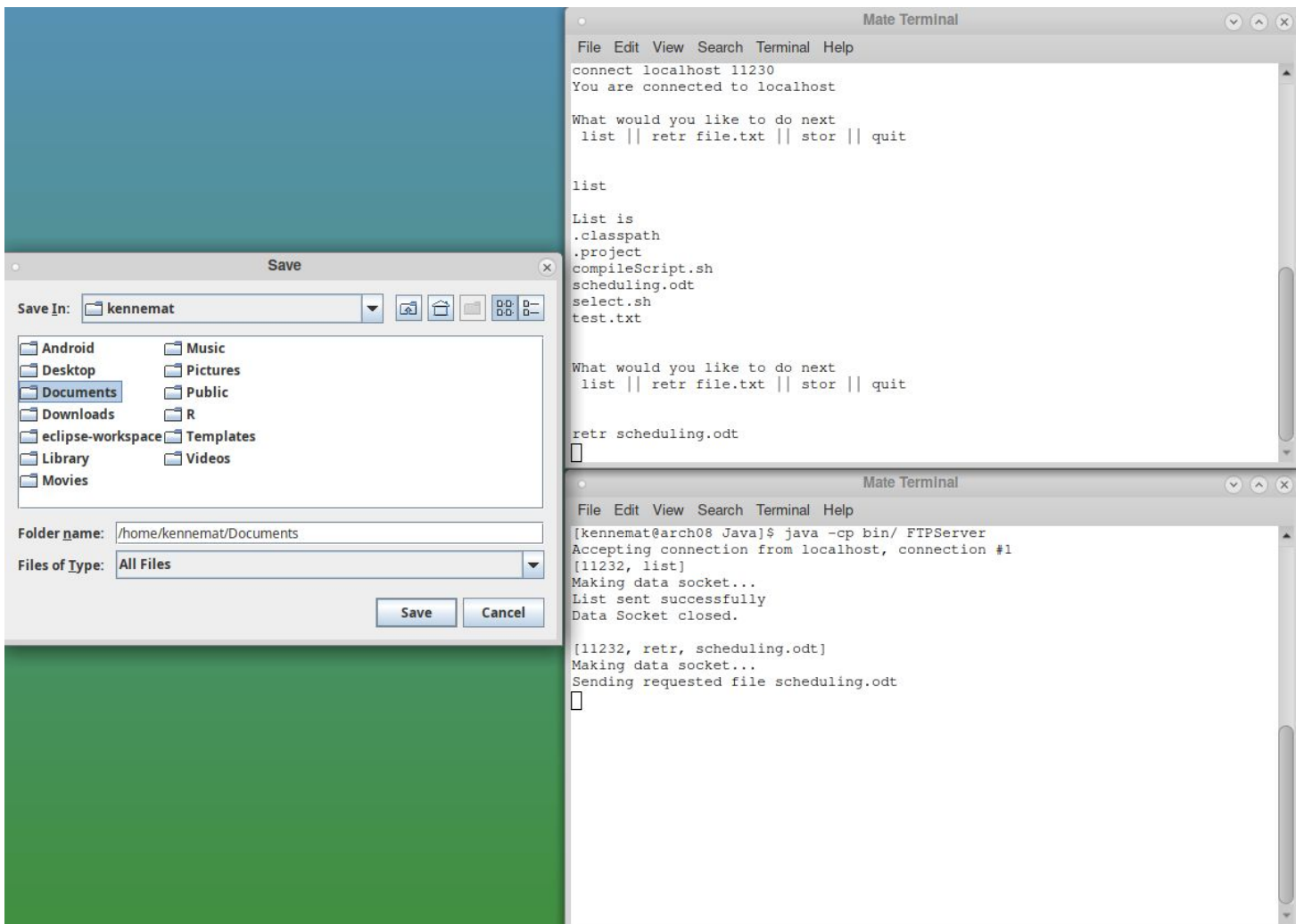
list

List is
.classpath
.project
compileScript.sh
scheduling.odt
select.sh
test.txt

What would you like to do next
list || retr file.txt || stor || quit
```

```
[kennemat@arch08 Java]$ java -cp bin/ FTPServer
Accepting connection from localhost, connection #1
[11232, list]
Making data socket...
List sent successfully
Data Socket closed.
```

Retr:



```
Mate Terminal
File Edit View Search Terminal Help

list

List is
.classpath
.project
compileScript.sh
scheduling.odt
select.sh
test.txt

What would you like to do next
list || retr file.txt || stor || quit

retr scheduling.odt
Saving file at /home/kennemat/Documents/scheduling.odt

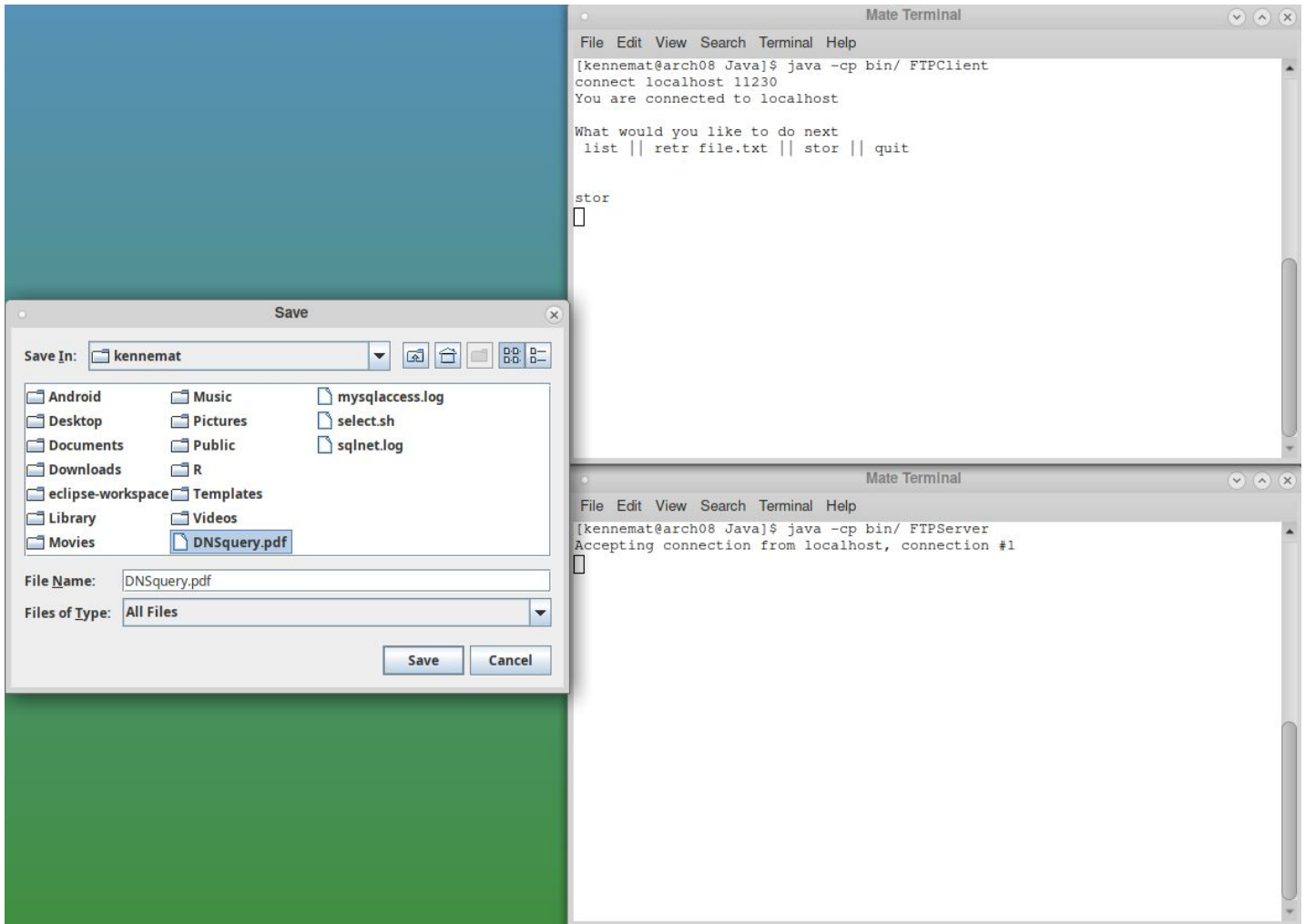
What would you like to do next
list || retr file.txt || stor || quit

Mate Terminal
File Edit View Search Terminal Help

[kennemat@arch08 Java]$ java -cp bin/ FTPServer
Accepting connection from localhost, connection #1
[11232, list]
Making data socket...
List sent successfully
Data Socket closed.

[11232, retr, scheduling.odt]
Making data socket...
Sending requested file scheduling.odt
█
```


Stor:



```
Mate Terminal
File Edit View Search Terminal Help
[kennemat@arch08 Java]$ java -cp bin/ FTPClient
connect localhost 11230
You are connected to localhost

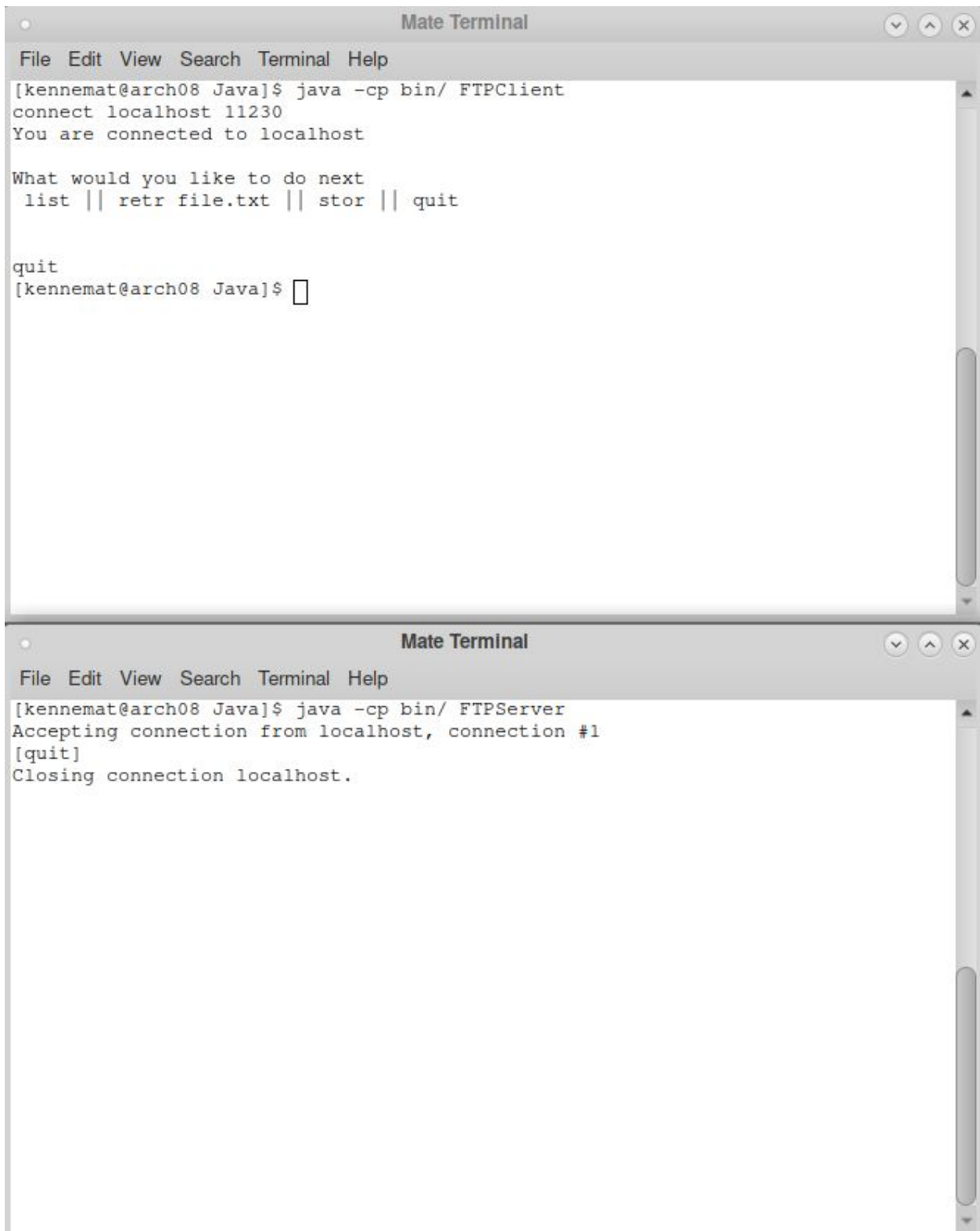
What would you like to do next
list || retr file.txt || stor || quit

stor
Storing /home/kennemat/DNSQuery.pdf

What would you like to do next
list || retr file.txt || stor || quit
```

```
Mate Terminal
File Edit View Search Terminal Help
[kennemat@arch08 Java]$ java -cp bin/ FTPServer
Accepting connection from localhost, connection #1
[11232, stor]
Making data socket...
Storing DNSQuery.pdf in working directory
█
```

Quit:



The image displays two screenshots of a 'Mate Terminal' window. The top screenshot shows the execution of the 'FTPClient' program. The user runs 'java -cp bin/ FTPClient', which connects to 'localhost' on port '11230'. The program prompts the user for the next action, listing options: 'list || retr file.txt || stor || quit'. The user enters 'quit', and the prompt returns to '[kennemat@arch08 Java]\$'. The bottom screenshot shows the execution of the 'FTPServer' program. The user runs 'java -cp bin/ FTPServer', which outputs 'Accepting connection from localhost, connection #1'. The user enters '[quit]', and the program outputs 'Closing connection localhost.'

```
Mate Terminal
File Edit View Search Terminal Help
[kennemat@arch08 Java]$ java -cp bin/ FTPClient
connect localhost 11230
You are connected to localhost

What would you like to do next
list || retr file.txt || stor || quit

quit
[kennemat@arch08 Java]$
```

```
Mate Terminal
File Edit View Search Terminal Help
[kennemat@arch08 Java]$ java -cp bin/ FTPServer
Accepting connection from localhost, connection #1
[quit]
Closing connection localhost.
```

Multithread:

The image displays four terminal windows arranged in a 2x2 grid, illustrating the operation of a multithreaded FTP client. Each window has a title bar 'Mate Terminal' and a menu bar 'File Edit View Search Terminal Help'.

Top-Left Terminal: Shows the initial command prompt and the first command entered.

```
[kennemat@arch08 Java]$ java -cp bin/ FTPClient
connect localhost 11230
You are connected to localhost

What would you like to do next
list || retr file.txt || stor || quit
```

Top-Right Terminal: Shows the output of the 'list' command.

```
list

List is
.classpath
.project
compileScript.sh
select.sh
test.txt
DNSQuery.pdf
mysqlaccess.log

What would you like to do next
list || retr file.txt || stor || quit
```

Bottom-Left Terminal: Shows the output of the 'stor' command.

```
stor
Storing /home/kennemat/mysqlaccess.log

What would you like to do next
list || retr file.txt || stor || quit

quit
[kennemat@arch08 Java]$
```

Bottom-Right Terminal: Shows the server-side processing of the commands.

```
Accepting connection from localhost, connection #1
Accepting connection from localhost, connection #2
Accepting connection from localhost, connection #3
[11232, list]
Making data socket...
List sent successfully
Data Socket closed.

[11232, retr, test.txt]
Making data socket...
Sending requested file test.txt
[11232, stor]
Making data socket...
Storing mysqlaccess.log in working directory
[quit]
Closing connection localhost.
[11232, list]
Making data socket...
List sent successfully
Data Socket closed.
```