An analysis of the File Transfer Protocol

Inzamam Rahaman

810006495

Nicholas Mendez

811002795

Jherez Taylor

812003287

Kimberly Brideglal

811000371

David Charles

811000385

November 2014

COMP 3150 - COMPUTER NETWORKS GROUP PROJECT

Department of Computing and Information Technology

University of the West Indies

Lecturer: Dr. Michael Hosein

Marker: Mr. Sterling Ramroach

Preface

A small office corporation requires a simple FTP Client and Server so that its employees can remotely access their files. Administrators would assign the users of the application a specific folder on the server. Users would be given permissions based on their user type that is, Administrator, Supervisor and Employee that would allow them to carry out file operations.

Contents

Chapter 1

Introduction

The File Transfer Protocol, hereafter referred to as FTP, is used to facilitate the transfer of files between computers over a network. It also allows for the transfer files between FTP servers. FTP is implemented using a client-server architecture with distinct control and data connections between the client and the server. This report provides an overview of an implementation of FTP. A subset of the commands within the protocol will be defined and examined. This overview also covers the benefits and drawbacks of the protocol.

1.1 Background

User groups and organizations rely on FTP to send files directly between computers on a network, bypassing the need to use physical media such as flash memory or optical discs to transfer said files. As such, a FTP client/server pair allows a user to carry out file operations with minimal knowledge of the implementation.

1.2 Objectives

The main objectives of this Computer Networks project are to

1. Provide an overview of FTP
2. Implement an FTP Client and Server
3. Facilitate authentication
4. Provide the user with a file explorer
5. Facilitate the upload and download of files
6. Manipulate files on a server
7. Provide functionality to allow a user to mirror folders

1.3 FTP Overview

Most FTP sessions begin with authentication of user on a given server. Authentication involves the user providing the hostname of the remote host which initiates a TCP connection to the FTP server in the remote host. The username and password are then sent over the TCP connection via the FTP commands USER and PASS.

Once authenticated, the user is then free to carry out file operations based on the level of permissions granted by the Administrator of the given file server. The permissions determine the files and folders that can be accessed and whether they have read, read/write and access to delete files. FTP facilitates the transfer of files between local and remote file systems.

1.4 FTP Detailed

FTP runs on TCP and uses two parallel TCP connections to transfer a file, a control connection and a data connection. Sending control information between the two hosts is handled by the control connection. This control information includes user identification, user password, commands to change the remote directory and also commands to put and get files. All FTP commands are sent via the control connection. The data connection handles the actual transfer of the file between hosts. Due to the use of a separate control connection, FTP is said to sends its control information out of band.

When an FTP session is started, the client initiates a control TCP connection with the server on port 21. The data channel is opened on any unused port set by the application programmer or administrator. Upon receipt of control commands from the client, the server initiates a TCP data connection to the client side. Only one file is sent per data connection, that is, the file is sent and then the data connection is closed. The control connection remains open for the duration of the FTP sessions and subsequent files are sent over new data connections that are opened.

There are two ways to send files over the data channel. The first is the Active Mode where the client specifies to the server how the transfer will be done. The client chooses a local port and mandates that the server send data to that port. A connection on port 20 is initiated by the server and the data is sent. Issues can arise from this setup as it becomes a requirement for firewalls to allow incoming connections on port 20 to a vast range of ports on on client machines. A vulnerability can be exploited by initiating connections from port 20 in order to scan internal machines.

The second method of data transfer is Passive Mode. The client requests a file from the server and specifies how the transfer will be done. The server selects an unreserved port which is incremented for each file transferred during that FTP session. The server then notifies the client to connect to the specified port to receive the file. The upside of this is that since the client initiates the connection then additional ports don't need to be opened in the firewall.

The FTP server is required to maintain the state about the user. It is necessary to associate the control connection with a specific user account, the current directory being accessed by the user must also be monitored. There is a significant overhead associated with tracking this state information for each client and thus the amount of connections that can be maintained simultaneously is limited.

Chapter 2

Technical Analysis

The FTP server has been implemented in Scala. Scala is an objected oriented, functional programming language. Like all JVM languages, Scala compiles to Java Virtual Machine bytecode. Consequently, Scala is interoperable with Java, meaning that code written in Java can be easily used and referenced by Scala code with little or no overhead.

Scala, being a functional language inspired by members of the ML family languages, has facilities to pattern match on algebraic data types by using their class based representations that implement a special purpose unapply method. There were two primary benefits of using this pattern matching:

1. Scala's standard library facilitates the conversion of strings representing regular expressions to a objects that implement the unapply method, thereby allowing for the easy dispatch of server activity based on the commands received by the server, as well as the extraction of arguments from the commands without the aide of explicitly defined parsers
2. Scala's algebraic types allow for the usage of Option types and for the usage of types that resemble Tagged Unions to facilitate dispatch based on the results of computations

The usage of Scala also illustrates that pieces of a Server/Client application can be built using different implementation languages so long as the contracts specified by the underlying protocol of the application is adhered to.

The FTP Client was written in Java. The FTP commands that were implemented will be explored.

2.1 Authentication

The FTP Client sends the commands USER <insert username>, PASS <insert password> to the FTP Server in order to establish a session and authenticate the incoming user. The method of authentication simply accepts the incoming parameters and checks a Map of users where the username and the password match.

Listing 2.1: Database of users

```
private val users = Map(
"Inzi" -> User("Inzi", "InziPass", None, "admin", "research"),
"Nicholas" -> User("Nicholas", "NicholasPass", None, "admin", "development"),
"Jane" -> User("Jane", "JanePass", None, "admin", "accounts"),
"John" -> User("John", "JohnPass", None, "normal", "accounts")
)
```

Here we can see that the database stores a user name, password, user type and their department.

Listing 2.2: Authentication

```
def authenticate(username : String, password : String) = {
users.get(username)
}
```

On the client side, we simply parse the user input and send the data to the server.

Listing 2.3: Send authentication data to the server

```
@Override
public ServerResp<String> login() throws IOException,
PoorlyFormedFTPResponse, InvalidFTPCodeException {
writeString("USER " + username);
String resp = readString();
System.out.println(resp);
FTPParseProduct prod = this.respParser.parseResponse(resp);
Status stat = this.fact.getStatus(prod.getCode());
System.out.println("Username sent and response received");
if(stat == Status.USERNAME_ACCEPTED) {
writeString("PASS " + password);
System.out.println("Set password to server");
String resp1 = readString();
System.out.println("Got response from server for password " + resp1);
FTPParseProduct prod1 = this.respParser.parseResponse(resp1);
Status stat1 = this.fact.getStatus(prod1.getCode());
return new ServerResp<String>(prod1.getBody(), stat1);
}
return new ServerResp<String>(prod.getBody(), stat);
}
```

2.2 Renaming Files

RNFR (Rename From) and RNTO (Rename To) allows a user to edit the name of a given file. The server must receive a RNFR command before a RNTO. To deal with this error, the server simply returns an error code and a relevant message. The following code snippet covers this case.

```
def rnto(filename : String) : String = "530 Invalid command sequence,
RNFR must be followed by RNTO"
```

The case where a RNTO does not follow a RNFR is handled as well.

Listing 2.4: Rename error checking

```
def rnfr(filename : String) : String = {
val command = this.readString
command match {
case RNTO(newname) => rnto(filename, newname)
case _ => "530 RNTO must follow RNFR"
}
}
```

When the commands are entered in the correct sequence, RNFR followed by RNTO then then the operation can be performed.

Listing 2.5: File rename operation

```
def rnto(oldFilename : String, newFilename: String) = {
val oldFile = new File(user.dept + "\\" + oldFilename)
val newFile = new File(user.dept + "\\" + newFilename)
oldFile.renameTo(newFile) match {
case true => "250 File rename successful"
case false => "450 Action not taken"
}
```

2.3 Folder Mirroring

In order to demonstrate the principle of retreiving files functionality was implemented to allow a client to mirror the contents of a folder on the FTP server. Firstly, let us take a look at the server code to retrive a file.

Listing 2.6: File rename operation

```
def retr(filename: String): String =
{
  println("Trying to send file")
  val filepath = user.dept + "\\" + filename
  val file = new File(filepath)
  file.exists() match
  {
    case true =>
      {
      try
      {
        val fileReader = new FileInputStream(file)
        var count = 0

        val size = fileReader.getChannel().size().toInt
        this.sendString("125 Expect " + size)
        var arr = new Array[Byte](size)
        outStream.flush()
        println("Telling to expect file")
        count = fileReader.read(arr)
        outStream.write(arr)
        /*
        while(count > 0)
```

```
          {
            println("Read " + count + " from file ")
            outStream.write(arr)
            println("Sent file piece")
            count = fileReader.read(arr)
          } */
          println("Awating acknowldegement")
          val r = this.readString
          println("Finished transimiit")
          outStream.flush()
          fileReader.close()
          "250 File send successful"
        }
        catch
        {
        case e: Exception => "450 Unsuccesful"
        }
    }
    case false => "553 Doesn't exist"
    }
}
```

Given the method above, the client method to mirror the folder first retrieves the list of files in the folder and stores the names of the files in a data structure. An interation through the data structure is then performed with a RETR (retrive) call being executed for each file. The method can be seen on the following page.

Listing 2.7: File rename operation

```
public void clientMirror(String dir) throws IOException,
InvalidFTPCodeException, PoorlyFormedFTPResponse, ClassNotFoundException
{
  ServerResp<String[]> filesResp = this.pwd();
  if(filesResp.getStat() == Status.SUCCESSFUL_FILE_OPERATION)
    {
    String[] fileNames = filesResp.getItem();
    HashSet<String> fileSet = new HashSet<String>();
    File folder = new File(dir);
    File[] filesInFolder = folder.listFiles();
    HashSet<String> names = new HashSet<String>();
    for(File f : filesInFolder)
    {
      names.add(f.getName());
    }
    Set<String> filesToCopy = setDifference(fileSet, names);
    for(String filename : filesToCopy)
    {
      ServerResp<String> resp = this.retr(dir, filename);
    }
  }
}
```

2.4 WIP2

Chapter 3

Summary and Recommendations for Further Work

In this final chapter you should sum up what you have done and which results you have got. You should also discuss your findings, and give recommendations for further work.

3.1 Summary and Conclusions

Here, you present a brief summary of your work and list the main results you have got. You should give comments to each of the objectives in Chapter 1 and state whether or not you have met the objective. If you have not met the objective, you should explain why (e.g., data not available, too difficult).

This section is similar to the Summary and Conclusions in the beginning of your report, but more detailed—referring to the the various sections in the report.

3.2 Discussion

Here, you may discuss your findings, their strengths and limitations.

3.3 Recommendations for Further Work

You should give recommendations to possible extensions to your work. The recommendations should be as specific as possible, preferably with an objective and an indication of a possible approach.

The recommendations may be classified as:

- Short-term
- Medium-term
- Long-term