Distributed Computing

Distributed Computing

Client-server File Management System using Datagrams

Katie Griffiths

T00175748

Table of Contents

[Datagram FTP Protocol 2](#_Toc499317557)

[Introduction 2](#_Toc499317558)

[Objectives 2](#_Toc499317559)

[Data Format 3](#_Toc499317560)

[Sequence Diagram 4](#_Toc499317561)

[Message Format 6](#_Toc499317562)

[Functionality of Messages 8](#_Toc499317563)

[Pseudo Code of Protocol Messages 9](#_Toc499317564)

[Design 11](#_Toc499317565)

[Introduction 11](#_Toc499317566)

[Objectives & Overview 11](#_Toc499317567)

[Design Philosophy 11](#_Toc499317568)

[Server-side Design Classes 12](#_Toc499317569)

[Client-side Design Classes 13](#_Toc499317570)

[Implementation 14](#_Toc499317571)

[Objectives & Overview 14](#_Toc499317572)

[Presentation Layer Implementation 15](#_Toc499317573)

[Service Layer Implementation 18](#_Toc499317574)

[User Manual 21](#_Toc499317575)

[Conclusions 26](#_Toc499317576)

# Datagram FTP Protocol

## Introduction

The User Datagram Protocol (UDP) is a transport layer protocol defined for use with the IP network layer protocol. It is defined by RFC 768. It provides a best effort datagram service to an End System (IP host). This service provided by UDP is an unreliable service that provides no guarantee for full delivery and no protection against duplication. The simplicity of UDP reduces the overhead from using the protocol and the services may be adequate in many cases. What UDP provides is a minimal, unreliable, best-effort, message-passing transport to applications and upper-layer protocols. Compared to other transport protocols, UDP are unique in that it does not need to establish end-to-end connections between communicating end systems. UDP communication consequently does not incur connection establishment and teardown overheads and this minimal associated end system state. Another characteristic of UDP is that it provides no inherent on many platforms, applications can send UDP datagrams at the line rate of link interface which is often much greater than the available path capacity and so this would contribute to congestion along the path applications therefore it would need to be designed responsibly.

UDP does not provide any communication security e.g. tampering or message forgery, therefore it needs to separately provide security services using additional protocol mechanisms.

## Objectives

The objectives are:

* To design and implementation of a message passing protocol.
* To handle basic authentication functionality
* To handle the transferring of data through a local network.

## Data Format

UDP encapsulates the raw payload with its header.

0 15 16 31

|  |  |
| --- | --- |
| Source Port (16 bits) | Destination Port Number (16 bits) |
| Length(UDP Header + Data) 16 bits | UDP checksum (16 bits) |
| Application Data (Message) | |

Source port number: This is a port number used by source host, who is transferring data. It is 16 bit longs. So port numbers range between 0 to 65,535.

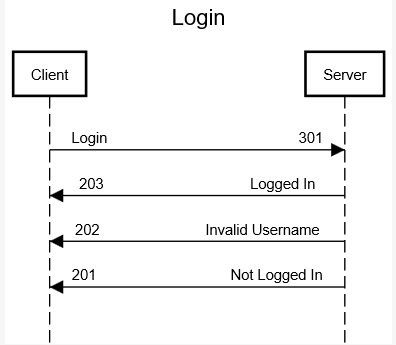
Destination port number: This is a port number used by Destination host, who is getting data. It is also 16 bits long and also same number of port range like source host.

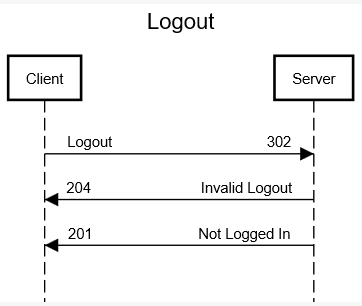
Length: Length field is a 16 bits field. It contains the total length of the user datagram, header and data.

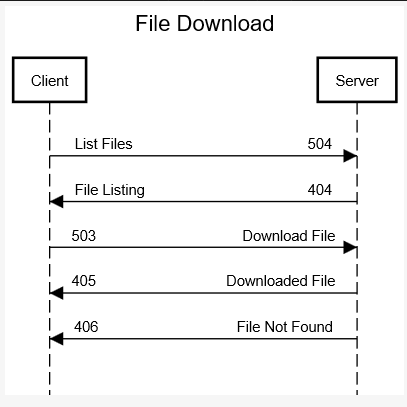
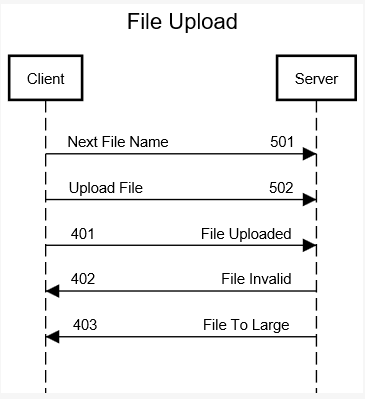
Checksum: The UDP checksum is optional. It is used to detect error from the data. If the field is zero then checksum is not calculated. And true calculated then field contains 1.

## Sequence Diagram

These are sequence diagrams for the implemented protocol codes for the communication between the client and server. The Client will have a set of unique source port numbers.







## Message Format

**Message:** Login

**Description:** The user can log into the server with a username

**Code/Number:** 301

**Message Parameters:** username (text String)

**Response Message**

**Code:** 203

**Text:** login successful

**Response Message**

**Code:** 201

**Text:** login unsuccessful

**Message:** Logout

**Description:** The user can logout of the server

**Code/Number:** 302

**Message Parameters:** username (text String)

**Response Message**

**Code:** 201

**Text:** logout successful

**Response Message**

**Code:** 204

**Text:** logout unsuccessful

**Message:** Upload

**Description:** The user can upload a file into their folder

**Code/Number:** 502

**Message Parameters:** file (byte array as string)

**Response Message**

**Code:** 401

**Text:** Upload Successful

**Response Message**

**Code:** 402

**Text:** Upload Unsuccessful

**Message:** Download

**Description:** The user can download a file from their folder

**Code/Number:** 503

**Message Parameters:** username (text String)

**Response Message**

**Code:** 405

**Text:** Download Successful

**Response Message**

**Code:** 406

**Text:** Download Unsuccessful

## Functionality of Messages

**Login**

Implement a connection message on the server as a function called Login()

//save username

//if a folder does not exist for username on login create said folder

//return 203: login successful

//if folder cannot be created return 201

**Logout**

Implement a connection message on the server as a function called Logout()

//check username logged in

//return 201: logout successful

//if logout is unsuccessful return 204

**File Upload**

Implement a connection message on the server as a function called Upload()

//check username is logged in

//check file is available for upload

//send 501: file name

//send 502: byte stream representation of the file

//return 401: upload successful

//if file cannot be uploaded return 402 upload unsuccessful

**File Download**

Implement a connection message on the server as a function called Download()

//check username is logged in

//check file is available for download in user’s folder

//return 405: download successful and the file as a bite array

//if file cannot be downloaded return 406 download unsuccessful

## Pseudo Code of Protocol Messages

**Login**

If code == 301 <String> (username)

If !exist(filename)

Create folder

Save username to state

Return 203

**Logout**

If code == 302

If logged in

Return 201

Else

Return 204

**Download**

If code == 503 <String> (File name)

If file exsits

Convert file to byte array

Return 405 (String) byte array

Else

Return 406

**Upload**

If code == 501 <String> (file name)

Save file name to state

If code == 502 <String> (file as byte array)

If file name in state

Save file

Return 401

Else

Return 402

**List files**

If code == 504

Return 404 list of files as ; separated string

# Design

## Introduction

To implement an inter process communication using Java’s Socket API. The protocol designed above will be used as the reference in designing and implementing the system.

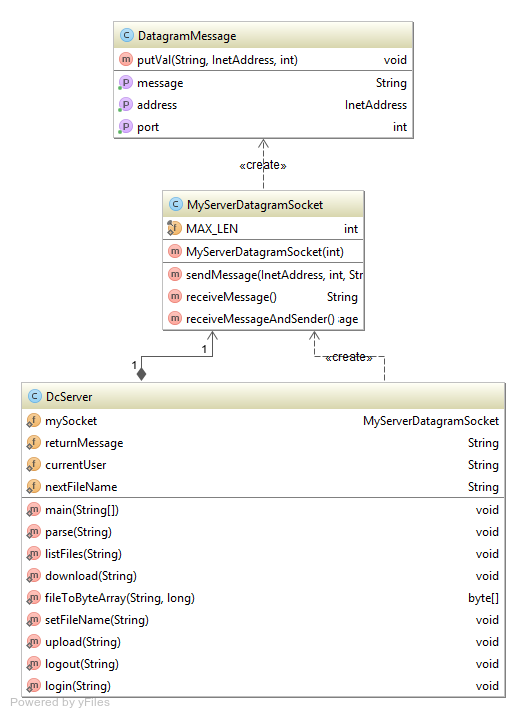
## Objectives & Overview

The objective of this project is to design a client-server file management system using datagrams. Using Java as the programming language I intend to implement this system using the protocol I have documented above. The client application will allow a user to login to the server through a GUI, if this is a new user a folder is created on the server with the user’s chosen login name, if a folder already exists with this name, they are treated as a returning user. Once a user has logged into the server a welcome protocol message will appear, then the user is brought to a file upload and download GUI, this will allow the user to choose a file locally to upload onto the server. The GUI also shows a file list of all the files that the current user has on the server already, once a file has been selected from the list, the user can then download that file to a download folder. The user also has the ability to log out at any point from the server by pressing logout, this will then send the user a protocol log off message.

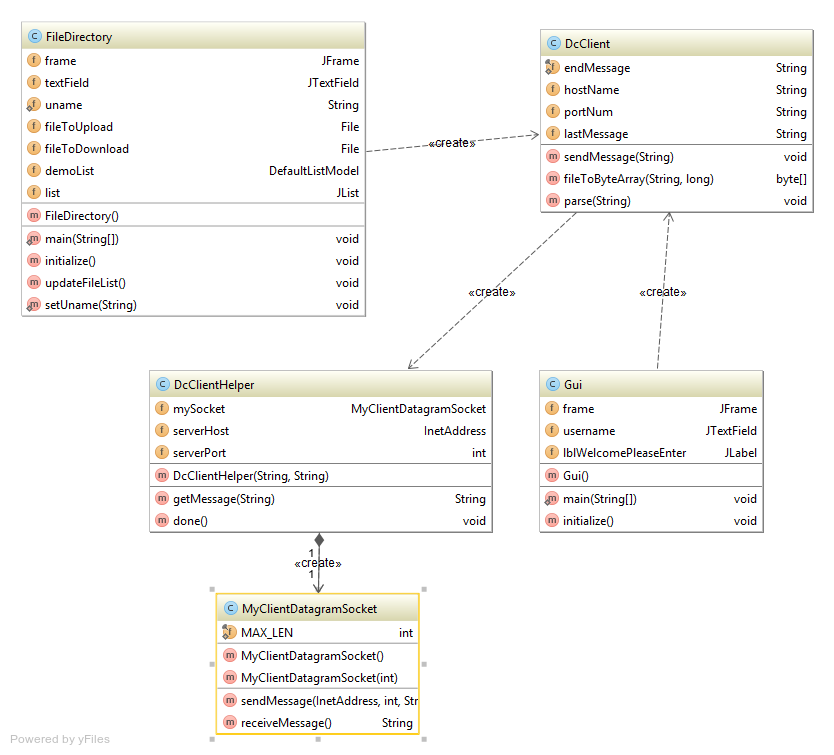
## Design Philosophy

* Presentation layer
  + Client application which formats the data to be sent out to the server it also stores all the client logic to send and receive data through the GUI
* Session layer
  + Server which includes the acknowledgements of data being received during the session as well as the sending and receiving of requests to and from the client and the server and to and from the server and client. It also allows for the termination of the session.

## Server-side Design Classes



## Client-side Design Classes



# Implementation

## Objectives & Overview

To implement a client-server file management system using datagrams to send and receive files. Using this type of socket it can support connectionless and connection-orientated communications at the application layer. Datagrams are sent or received without the notion of connections at the transport layer, the runtime support of the socket API can create and maintain logical connections for datagrams exchanged between two processes.

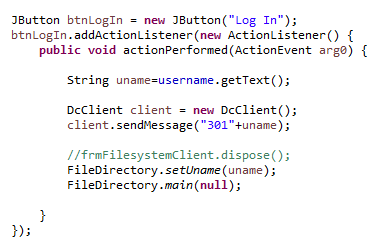
The application process wishing to send and receive data using this API must instantiate a datagram socket (socket), each of these sockets are bound to a UDP port on the local machine. For the application to receive datagrams to the socket, the process creates a datagram packet object which references a byte array and calls a receive method in the datagram socket object, which specifies the argument referenced to the datagram packet object.

This implementation to use UDP for this application does not provide any communication security e.g. tampering or message forgery, therefore it needs to separately provide security services using additional protocol mechanisms.

## Presentation Layer Implementation

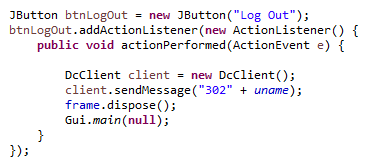
**Login Request**

Through the GUI class the log in protocol is triggered by the user clicking a JButton, this gets the username and instantiates the DC client class which sends a 301 message with the username. If it’s a new user the FileDirectory class is run, which opens a new JFrame.



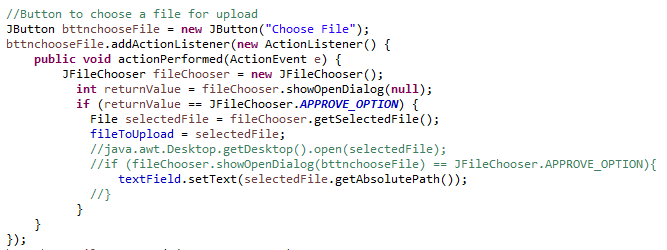
**Logout Request**

The log out request works the same as the login, as when the log out JButton is pressed the DcClient is used to send a protocol message 302 to the server with the username attached and closes the file directory frame and calls for the login frame to open.

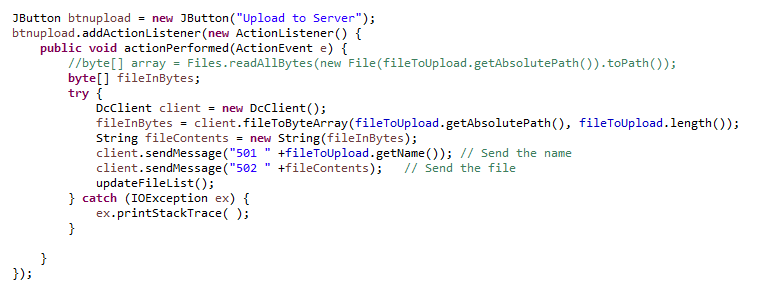


**Upload Request**

For a file upload the code snippets below show when the Choose File button is pressed the file chooser opens with files on the local machine, when the user selects a file the file path information goes into a text field.



When the upload button is pressed, first the file name is retrieve from the upload field, and then the the file is processed into an array of bytes by the filetobytearray function in DC client, the size of the file is also sent to this function to ensure that the bytearray is of the correct size. The protocol request 501 sends the name of the file to upload and then protocol 502 sends the file to the user’s folder. As a convenience, the program then requests a new list of all files present in the folder in order to update the file list to include the file that was just uploaded.



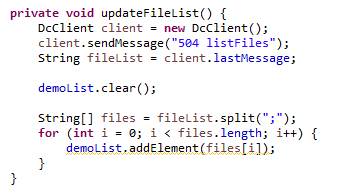
**Download Request**

For the download to be initialised the code below shows when the download button is pressed the Dc client is sent a protocol message 503 along with the selected files. The file that has been selected is turned into a byte array by the server, much like the upload functionality above, a new file is created on the c drive and the selected file is the outputted to the downloads folder.



**Update File List**

To allow the user a convenient method of seeing all available files in their folder in the file directory GUI an updateFileList method was created. A protocol number of 504 is sent to with the list of files. The data is returned in a semi-colon separated string from the server, so this function also includes some functionality to split it into separate items, which are then populated into a DefaultListModel which is used to fill the JList element in the GUI.



## Service Layer Implementation

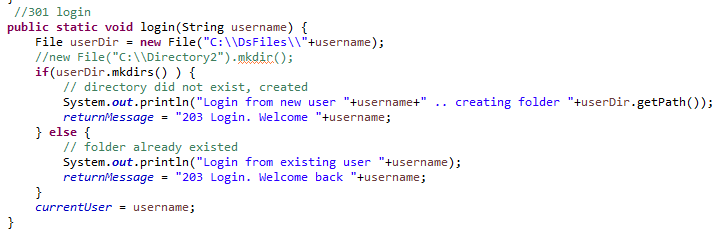
**Parse Request**

As most messages are in the format of <Three digit code> <Message>, string parsing functionality is required to interpret this. This method gets the first three symbols of the message being received i.e. the protocol number for the relevant action and separates the remainder of the string. This is then trimmed. This allows for the protocol action to be interpreted, which is done with a switch statement.



**Login Request**

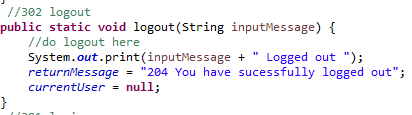
On the server the login method is created which takes the username as an argument. The mkdirs() method of the file class will return true only if the folder is actually created, so if it returns false, we know that it already exists and can tailor the application response appropriately. A 203 response is generated with either a welcome, or a welcome back message as required.



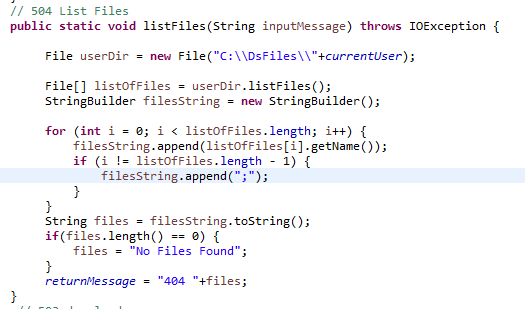
**Logout Request**

On the server a logout request method is called, it takes a string, inputMessage as the parameter , this displays a simple message with 204 protocol being called and a successful log out message.

**List Files**

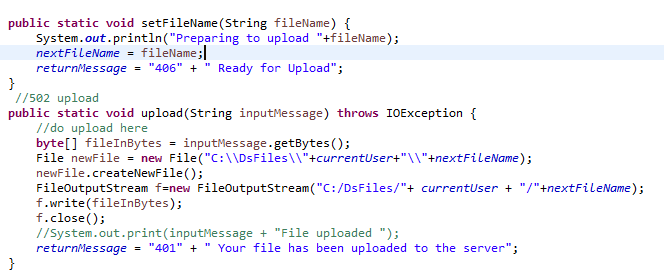


The protocol method 504 is called on the server, this takes the imputMessage as the parameter. The file class is first used to generate a list of files in the selected directory, and a StringBuilder class is then used to combine these filenames into a semi-colon separated string. Should the directory be empty, the message “No Files Found” is returned. The method uses the file directory of the current user who is currently logged into the application.



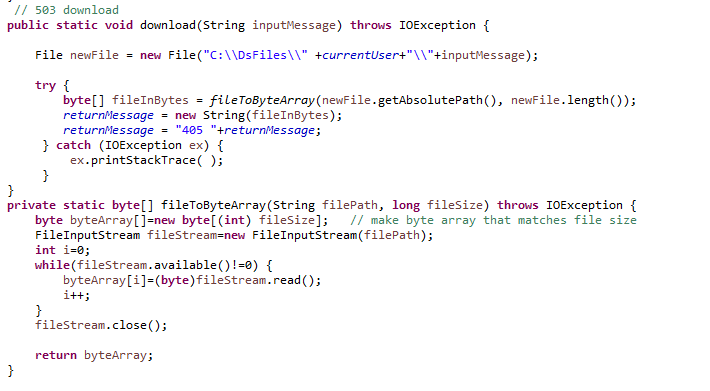
**Upload Request**

The upload method on the server is called when a user chooses to upload a file. Uploads require two separate server calls. First, the file is sent to the server, followed by the file. Once the file is ready for upload the file is turned into a byte array and output into the current users folder. A message is returned with the protocol number and a confirmation message to say that the file has been uploaded to the server.



**Download Request**

The download method, which takes the filename as an argument, loads the file using an instance of the File class. This is then converted into an Array of bytes using a FileInputStream. This is then cast into a String to allow it to be sent over the socket. I had originally defaulted the size of the array to 1024, to match the length of the datagram messages, but I found that this padded the array with NULL bytes, which could cause display issues when saving files on the other end, so I also take the file size via File.length() to ensure that the array of bytes is the correct size.



# User Manual

This java application was developed using Eclipse IDE.

* First ensure there is a folder on the C drive named DsFiles
* Secondly create a folder inside of the DsFiles folder called downloads

To run the application, run the server class first this message will be displayed in the console window.

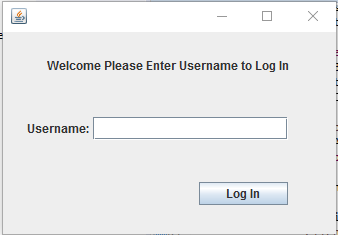
**Acknowledgement on Console Window**



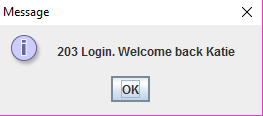
Once this message is displayed run the GUI class, this will generate the login screen below.

**Login**

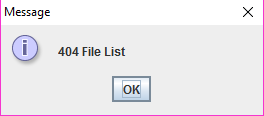
When the Login screen is running, enter in a name into the text field and then press the Log in button.



When the user has logged in a confirm message appears showing the protocol number. If a new user is logging onto the server a file will be added to the named folder (DsFiles) on the C drive with the name of the person that has logged in. Just press the OK button to continue.



A protocol message 404 File List will appear, this states that the list of files is being generated on the server to display. Just press the OK button to continue.

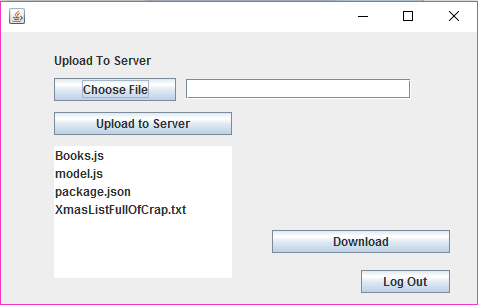


**File Directory GUI**

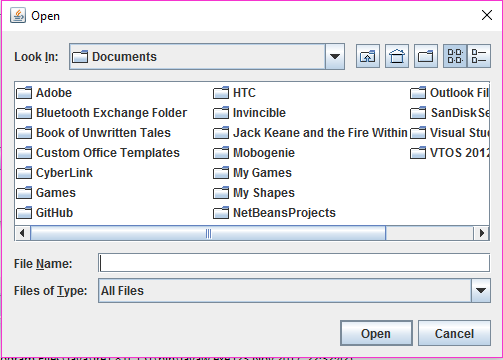
After Login is complete the FileDirectory GUI will run automatically, this allows for the Upload and Download.

**Upload**

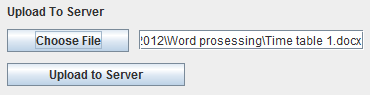
To upload to the server click on the Choose File button, this will open a file directory for you to choose the file you would like uploaded.



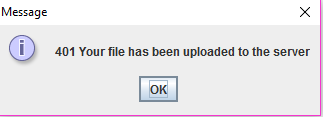
This is the file directory which appears when the Choose File button is pressed, this allows you to choose a file that is locally on your computer. Once you have selected a file, press the Open button.



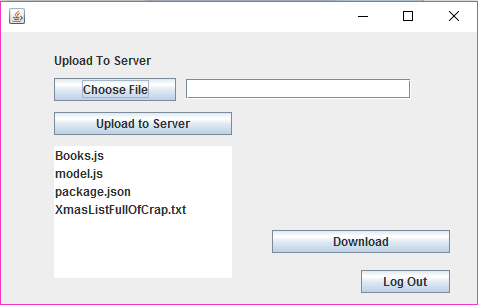
Once a file has been chosen the path name of the file will display in the textbox next to the choose file button. Once you are happy with your choice of file, you can press the Upload to server Button.



When the Upload to Server Button has been pressed an acknowledgement message will appear showing the protocol number with a message saying that your file has been uploaded to the server. Just press OK to continue.



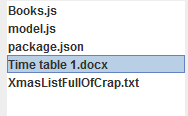
Once a file has been uploaded to the server it will display in the list menu as well as in the folder that was created on the C drive.



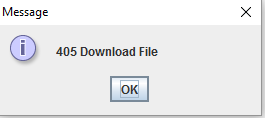
**Download**

From the list you can select a file that you would like to download, just by pressing on the file name. This will highlight the selected file that you have chosen. When a file has been selected press the Download button.





Once the download button has been pressed a protocol acknowledgement message will appear showing that the file has been downloaded. Just Press OK button to continue.

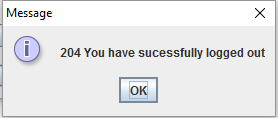


**Logout**

To Log out of the server press the Log Out button.



Once the Log Out button has been pressed a protocol acknowledgement message appears with a successfully logged out message.



Once you have logged out the Login screen will appear, this allows for a new user to log in to the server or for the same user to log back in again.

To quit the log in screen press the x on the top right hand side of the top bar, this will stop the gui from running.

To stop the server from running press the terminate button on the console and then press the remove launch button, this will clear the console window and stop the server from running so the application can be started again if desired.

# Conclusions

My conclusion on this project is that UDP protocol has a potential to be used in the future, seeing that emerging technologies are now having a tendency to choose UDP over TCP even for a file transfer makes this more confirming. The Link layer technologies have an improved speed and reliability in transfers. Though the reliability of UDP is dependent on the transfer media, a conclusion can be made that in time UDP may be used more liberally with the reliability not being much of an issue. The main problem using UDP is that it allows for multiple routes of unequal trip time to the same destination which means that packets can arrive in arbitrary order, if there was not an alternative routing or the load balancing was not in use, UDP can been seen as being too far away from the required reliability to carry the packets in the right order. UDP doesn’t establish a connection before sending data, it will just send. The reason for it being fast is there is no form of flow control, no error checking, error correction or acknowledgement is done by UDP. The only concern for UDP is the speed. There are a number of advantages and disadvantages to using UDP.

The many advantages of using UDP are:

* More Efficient
  + Given an application where packet size is small and frequent guaranteed overhead may not be required.
* Reduced Overhead
  + In centralised system with single server overhead of multiple TCP socket connections may be an issue.
* Reduced Delay
  + With UDP there are fewer delays due to error checking and flow control.

The many disadvantages of using UDP

* No Guaranteed Packet Delivery
  + Packets sent via UDP may be lost. If a file is being transferred a loss in packet data is unacceptable.
* No Packet Sequencing
  + Packet sequencing must be included for those applications that require sequential data access. No default sequence number as in TCP.
* No Flow Control
  + Flow control shapes internet traffic to prevent the network from overload. We may limit the rate of packets sent per second or the number of packets that are unacknowledged.