Allegheny College
Department of Computer Science

Senior Thesis

The Capabilities of Peer-to-Peer Technologies

by

Charles Misback

ALLEGHENY COLLEGE

COMPUTER SCIENCE

Project Supervisor: **Professor Bonham-Carter**
Co-Supervisor: **Professor Kapfhammer**

7 March 2021

Abstract

Peer-to-Peer (P2P) networks have largely been neglected in favor of traditional Client-Server networks which typically rely on large companies to purchase and control massive server farms as hosting. In contrast, P2P networks rely on the every day person to "crowd-source" power to complete tasks that even Client-Server networks have trouble with. The neglect towards P2P networks has made it difficult to demonstrate their numerous advantages over Client-Server networks. This paper presents a lightweight P2P network which implements many of the features found in the popular P2P file sharing network known as Bittorrent. Along with this, a sister client-server network was developed with the goal of being a suitable means of comparison during testing alongside the P2P network. Some of the parts that the testing focused on is performance, scalability, and security. The results show a download speed proportionate to the amount of peers the user is downloading from. This benefit is worth exploring in future research and eventually would benefit from real world testing and implementation.

Acknowledgment

I would like to thank Professor Mohan for aiding me in this entire process. From brainstorming the complicated idea of P2P networks to helping choose an existing project within the realm of P2P networks to aid in the project's goal.

Though not directly contacted, Professor Nadeem Abdul Hamid of MIT aided with their github project titled PeerBase, "A lightweight framework for peer-to-peer programming". Without a framework to build around this project would not have been possible.

My first and second readers, Professor Bonham-Carter and Professor Kapfhammer both helped me a lot finishing the process out. Without them I would never have gotten this project fully flushed out.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Outline

This senior thesis project consists of five main chapters: Introduction, Related Work, Methodology, Experiments, and Conclusion. The first chapter (this one) will present the problems which P2P networks can solve, the motivation behind doing so, how the problem will be solved, verification of correctness of the solution presented, and finally the applications of the solution.

1.2 Problem and Motivation

The problem is best explained after learning the organizational structure of the most common network, Client-Server (CS). A CS network consists of many clients and a single server from which users request data from. Clients do not directly interact with other clients because the server is always the middle man. For some applications this method of interaction may be very relevant and efficient, but not all applications. For example, if many users were to request the exact same file from the same server it can get overwhelmed when trying to deal with all these requests. What if other users already have the file downloaded? Wouldn't it be faster to download the file from multiple sources at the same time? This is exactly how a P2P file sharing network operates, through directly connecting users to each other where each user can also function as a server.

This question poses the problem of scaling. Client-Server networks have terrible difficulty scaling up to meet the requests of all their users. So far, this problem has been solved one way: purchasing more powerful servers. Wasting hundreds of thousands of dollars on compute power only needed for a few hours everyday. P2P networks solve this problem but utilizing bandwidth from hundreds of peers across the network. Instead of waiting for your turn to download a file, you should be able to request the file from one of the hundreds of users that have already downloaded it. The load is shared by hundreds of users across the network so each user only contributes a small fraction of the overall load. A P2P network's power is not held by any one user, it is the entire user-base as a whole. Hundreds of sources for bandwidth is more reliable and powerful than one single massive source.

This project is meant to give another reason for people to use P2P networks. By showcasing the potential benefits of P2P file sharing networks I hope to convince the reader that they can easily support the needs of users all over the world.

While there are a few well established P2P file sharing networks they all have lots of overhead. This project shows anyone how to use a lightweight P2P file sharing network to share files of any size. It has minimal overhead so it allows you and your peers to share files at extremely high speeds anytime you want when client-server file sharing networks are bogged down by needless overhead. Not only do P2P networks have extremely high download speeds, but they also are extremely reliable. If the server in a client-server network goes offline, every single user on the network also is forced to go offline. In a P2P network like the one described in this project, every single user can act as a server at a moments notice. There is no more waiting for the server to come back up, just connect to a new peer and continue downloading.

P2P networks offer so many benefits that have been passed up in favor of centralized servers in a traditional client-server network and this project is attempting to make everyone acquainted to these precious benefits.

1.3 Goals of the Project

This project further explores the issue and presents an example P2P file sharing tool to demonstrate the benefits that it shows while competing against a CS file sharing tool.

The goal is to effectively demonstrate and convince the reader that P2P technology can effectively replace CS networks in many part of the Internet.

I believe the use of P2P technology could greatly decrease the need of large server farms. Humanity creates a lot of waste and wasting computing power is a large one. Networks that could be optimized by utilizing new, more efficient technologies continue to use outdated and increasingly wasteful software. On top of this many users will benefit from the increased download speeds which can be sought from a P2P file sharing network.

1.4 Current State of the Art

To understand the intricacies of P2P networks, it is important to understand what a decentralized network is and what this unique network architecture means for users. A centralized network is structured how you would expect considering its name, users communicate with one central server. All network activity passes through this server. Another name for this model is Client-Server (CS). To contrast, a decentralized network has no "central" server; each user communicates with at least one (usually more) other client that is currently on the network.

In order for a decentralized network to operate successfully, each node (a single user) must be of equal importance to every other node; they all possess the means to single-handedly operate the network. If a single node is shut down, the network will continue to persist. If that node happened to be the central server of a client-

server network, the whole network would be shutdown. Stability like this is the main advantage of a decentralized network.

Something that goes hand in hand with stability is security. It is more difficult to target multiple nodes than one central server so malicious activities such as hacking and DDOSing become less effective. This also enables a very reliable means of verification which utilizes the multitudinous nodes to ensure the validity of requests made by users.

P2P networks can be split up into three main groups: unstructured, structured, and hybrid. If a P2P network is "structured" that means there is a protocol which enables nodes to quickly search for one another, usually in the form of a distributed hash table (DHT). Unstructured simply means the lack of a protocol, each node randomly connects to other nodes. With no protocol, unstructured P2P networks struggle with connecting specific users to each other, having to search through the whole network, consequently consuming lots of processing power. However this lack of protocol also allows users to quickly connect and disconnect without having to jump through any hoops. Hybrid models combine both P2P and client-server architecture through a central server acting as a sort of air traffic controller, directing which node for a user to connect to.

Initial research shows that there are mainly two different types of P2P applications that exist currently: file sharing and computing. File Sharing is currently the most popular with BitTorrent leading the P2P charge; providing high bandwidth downloads for popular files with no server overhead. P2P Computing is a very different story. The idea of a P2P computing is to use a user's device when they're not in use, running computations to fulfill the needs of the entire network.

1.5 Ethics

When it comes to the Internet and the users on it that access content everyday and are affected by it, ethics is a very important discussion to have. Currently, one of the first concerns that people have about P2P network is security. Discussions about security can go many different ways, but I think the most important one to have is relating to efficacy of requests.

Efficacy of requests means how a user knows that the information being sent to them is truly the information that they requested. In a CS network this is much easier to keep track of because every single client is requesting information form one and only one source, the central server. This way every single user knows exactly where and who is supplying the information that is being sent. They can then directly make a decision of whether or not to trust the server and in the end, whether or not to accept the data they requested. The server identity is known at all times. In a P2P network, how do you know the identity of the users that you requested data from and more importantly, knowledge that the data you requested and received is actually the data you requested in the first place? In a CS network there is only one entity that you communicate with, the server. In a P2P network there can be hundreds of entities (peers) that you connect to simultaneously so it's unrealistic for a user to manually decide to trust each peer.

If a user does not 100 percent know that the integrity of their request and data

was upheld, why should they use a P2P network over a CS network which they use all the time? P2P networks solve this through cryptographically hashing the contents of a file.

File hashing is a process where an input file is sent through a complex set of equation which output an effectively unique string. Every single time the same file is run through the same file hashing algorithm, it will output the exact same string. If this is implemented into a P2P network, users never have to worry about file integrity. One of the downsides of this method relates to where they initially find the unique output string. Many modern P2P networks support a method where users supply a file containing these cryptographic hashes as a means to verify the file they download.

Another important ethical discussion to have is the content being distributed by these P2P networks which consequently brings the topic of who controls these networks? A traditional CS network is of course operated and controlled by the current owner of the server. They decide what their users can and cannot download because they can manipulate the server however they like. On the other hand P2P networks struggle with this as a fundamental concept. By their very nature decentralized networks can't be controlled by just one person, but this is by design. P2P networks are effectively immune to broad Internet restrictions that affect CS networks. They're also very difficult for one company to control and profit off of because of this. While this is bad for monetization, it gives the people what they want devoid of any intervention by the government. In the Internet of today where our every move is tracked by websites trying to serve ads to use the anonymity provided by P2P networks is very important.

Although, this might not always be a good thing. When people hear the word "torrent" they most commonly think of illegally pirated content. It's much easier for the government to go after individual hosting websites because once the website is taken down, no one else can access it. A P2P network stays active until the very last user is removed. Every single user has the ability to host the network by themselves. The effects of digital media piracy is a multi-faceted issue and it is nearly impossible to obtain conclusive data in favor or against due to the inherent reclusive nature of piracy [1].

Chapter 2

Related Work

2.1 Bittorrent

The related work in P2P networks is not as numerous as standard networks obviously so I will choose to explore the work of Bram Cohen, the creator of Bittorrent. Cohen wrote a research paper going into depth on some of the key algorithms that are integral to the success of the network. These are focused on the unique idea of splitting up a data file into numerous pieces, storing them on a user's computer once downloaded, an assortment of incentives to sustain users in a network, and methods of searching for files directly on the network [2].

The easiest way to understand the Bittorrent protocol is to start with a torrent file. A user creates a .torrent file which contains information about the file, the length, name, hashing information, and the URL of a tracker. A tracker is a user which directs incoming leechers (people wanting to download the file) to those which are known to have the desired file. Those wishing to download use the Bittorrent protocol to request this list of possible peers from the tracker. The protocol runs on top of HTTP and contains necessary information such as useful networking information to link peers together. From here, downloaders send diagnostics about the file which they have and which part. SHA hashes are supplied to the user via the .torrent file to ensure the integrity of the incoming data file which prevents seeders (peers which share the data) from spoofing any downloads for the user, possibly with some kind of malicious intent.

Sometimes this does not work as intended, as the file itself can already be malware infected. One solution to this is creating a percent satisfaction that is raised by positive feedback and lowered by negative feedback [3]. Seeders send data in quarter megabyte pieces which enables a user to not become independent on just one user and instead spreading out the download among all available peers. This continues while a file is being downloaded by a user.

Some optimizations which help Bittorrent have such a great reputation for download speeds is the process of breaking pieces of a file down further down into sub pieces, roughly 16 Kb in size, and always having at least five requests queued at once. The size of the queue has been calculated based on an average value which saturates most connections and uses the network in the most efficient way possible. A user will also prioritize requesting these rare sub-pieces over more common ones in order to maximize the chance of downloading the entire file since a seeder could terminate at any point

[4]. In addition to this, the very first piece which is sent to a user should be quick to download in an attempt for the downloader to get a whole piece as quick as possible so they can start seeding that piece to other peers. Though as soon as this is done, rarest first is resumed. As a download is close to finishing, endgame mode is triggered. This forces a downloader to request all the sub-pieces of the piece that they're currently on at once from all peers, which may seem to be redundant but makes sure that a download doesn't hang at the very end. A measure such as this one may seem wasteful since some requests will already be fulfilled when a peer gets around to it but the amount of bandwidth wasted is negligible.

2.2 Incentives

This brings us to one of the most important aspects of maintaining a P2P network's longevity, incentives. When a user stops seeding a download to a peer but still accepts downloads from it, this is called choking. This may seem backwards, but doing so always strives for the highest possible download/upload speed through taking advantage of TCP's built-in congestion control and stopping freeloaders (users which refuse to upload) [5]. Choking is a tit-for-tat policy which is based on the idea that a user should be able to request as much data as they have downloaded for themselves. If a peer's upload speed to you is the lowest out of all other peers, it is choked. The issue is created when a user can only have a certain amount of unchoked peers (default is four), so in order to maximize download speed the user must search for peers which upload to the user the quickest. One glaringly large issue with tit-for-tat is the restriction of the network, punishing peers with low bandwidth even if it was only for a short time. To fix this, Bittorrent clients use optimistic unchoking to unchoke a random peer no matter its performance once every 30 seconds. For reference, tit-for-tat choking and unchoking occurs every 10 seconds. This period of 10 seconds is long enough for a peers upload to ramp up to full capacity so as to not bog down users with constant choking and unchoking [6].

A very interesting strategy that some Bittorrent client implement is anti-snubbing, caused when a user hasn't received data from a peer for a minute. The user will then refuse to upload to that peer until it is optimistically unchoked, which consequently forces the user to find other peers and not waste time seeding data to a peer which won't reciprocate [7]. This only continues while no data has been uploaded so if that previously ignored peer optimistically unchokes the user, the tit-for-tat relationship is restored. This feature is typically only used when lots of other peers are available to prevent total snubbing. Overall, Bittorrent is a very promising and extremely interesting network and in many ways, a large experiment for larger, more widespread and mainstream P2P networks. It is an example of a P2P network without any central server whatsoever, except for distribution of the .torrent file [2].

Tit-for-tat is not the only way to incentivize peers to upload to the network. There is lots of research on this topic. This includes round-robin based selection of peers, an auction based system [8], the clustering of similar-bandwidth peers [9], and a new proposed method to directly improve upon tit-for-tat [10]. When developing systems such as these, it is prudent to attempt to break them; simulate exactly what people may

attempt in the real world [11]. Doing so finds weaknesses and strengths of methods. The above comments on some of the avenues that Bittorrent takes to attempt to reduce this.

2.3 Distributed Hash Table

Mentioned briefly in the introduction, a Distributed Hash Table (DHT) is a replacement for the aforementioned .torrent file. A DHT is a means for peers to index the current files which they are willing to upload and share among the P2P network. A peer can send a search query for a file which they or their directly associated peers may not have, and the query will propagate throughout the whole network. (Alg. 1) When a peer confirms that they have the requested file, they start to upload it to the original peer which requested it. There are varying methods of implementation for this but the most well known is using a Kademlia DHT [12]. A particularly interesting solution to this is cubit, a scalable and efficient P2P system which finds files on other peer's systems closely similar to a search query [13].

Algorithm - 1 Handle Query

Input - reqPeerid, query, hops
Output - None

```
1: for (fname in self.files.keys()) do
2:     if (query in fname) then
3:         self.connectandsend(peerid, fname)
4:         return
5:     end if
6: end for
7: if (hops <= 4) then
8:     for (peer in self.peers.keys()) do
9:         self.connectandsend(peer, reqPeerid, query)
10:    end for
11: end if
```

2.4 Peer-to-Peer Computing

Though this is not the main focus of the proposed project, there are many other interesting aspects of P2P networks such as crowd-sourcing compute power in order to process millions and even billions of different protein folding patterns to find a suitable one for attacking a virus. Qmachine was developed to bring this ability to modern day web browsers, stressing the importance of large cross platform mobility to reach as many people as possible. The authors found that the average amount of compute power which comes from the daily traffic of a high traffic website is well within the HPC (High Performance Computing) range, so if implemented into such a website could supply super computer level performance on demand [14]. It's important to mention that this

is most certainly not an unstructured P2P network so it does not come with all the privacy capabilities of unstructured P2P networks such as Bittorrent which everyone is a stranger to each other.

Going back to protein folding, folding@home is one of the largest movements and applications that users can download in order to help further protein research. [15] It's called distributed computing or volunteer computing in this case. A central server that controls and stores all folding@home results sends a computation to a volunteer's computer which is then instructed to compute the result and then sends the result to the central server. Surprisingly, the largest obstacle in distributed computing is finding enough storage for the computed results when one might expect it to be complicated networking. This is why the next goal of this effort is to use distributed storage, so instead of gathering compute power from volunteer's devices a user would volunteer their local storage.

Overall, Peer-to-Peer networks are still a fairly unresearched field due to it's inherent reclusiveness but the possibilities of it are endless.

Chapter 3

Method of Approach

This section will begin with a straightforward and non-exhaustive description of the design of the P2P application and the thought that went into it. An exploration into how the original motivation to complete the project came about as well as how this ended up affecting the final product will be explained. Every part of this project aims to satisfy a curiosity in the field of P2P technology. After that, select functions from each python file have been selected to describe in further detail to aid the reader in a deeper understanding of the project. There are three main python files in this project: btpeer.py, btfiler.py, and filergui.py. The first file contains basic communication protocols such as creating sockets between peers. The second file utilizes btpeer.py and adds a communication protocol complete with requests and responses. Finally, filergui.py is what creates the UI and allows the program to be human readable and controllable. The github repository for this project can be found here.

The decision of which programming language to develop this project was simple. The language that makes this project possible is python with it's numerous libraries that aided in the creation of this project. Python sockets were exclusively used for communication between peers. The GUI was built with the tkinter library, a multi-platform graphical user-interface.

The most important part of any project is the groundwork, an integral step that paves the way for the complicated features that will be implemented. In the case for this project, a solid network is needed. This network should have the ability to connect and communicate with multiple peers at a moments notice. To aid with this, an open source project called PeerBase was used utilized. This network is described by the author, Nadeem Abdul Hamid, to be "A lightweight framework for peer-to-peer programming". It utilizes the TCP sockets native to python as well as multi-threading to create multiple established connections to peers.

The arguments to launch are as follows, "python filergui.py server-port max-peers peer-ip:port". The "server-port" argument is the port which you want your client to be on. Your IP is automatically assumed to be the one you are on natively. The next argument, "max-peers", allows the user to decide how many peers they would like to be connected to at one time. Finally, the last argument, "peer-ip:port", is the client to which your are connecting. As soon as your client connects to this peer your client will request and add all of this peer's peers to your peer list.

Below is figure 3.1 which is a flowchart of a basic workflow that is happening in the

backend while a user interacts with the GUI. In order to successfully download a file a user must connect to peers, search and find the specific file on the network, and then request chunks of the file from each peer. The reader should keep this in mind while reading through the methods that make this process possible.

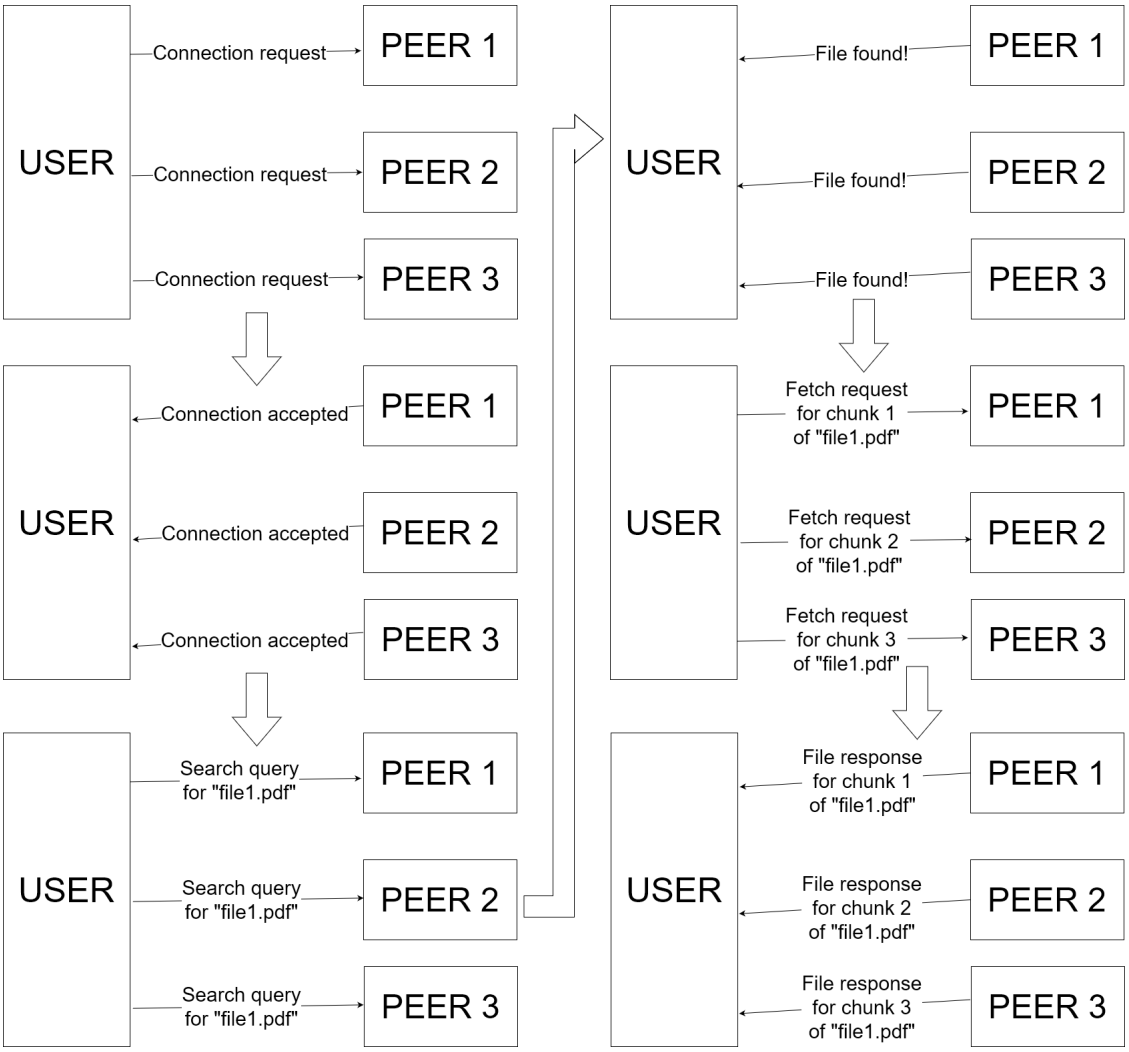

Figure 3.1: Flowchart of peer connection, file searching, and file transferring

3.1 Networking Groundwork

The first python file which will be visited is btpeer.py. The majority of the groundwork is created here for the communication between peers. There are methods such as makeserversocket (takes a port number as input and creates a socket which is binded to the input port number and is set to listen for incoming connections) and addpeer (takes a peer's id as well as host and port number and adds it to the local dictionary which holds all the information related to nearby peers) which are relatively simple and are seen in almost any modern network. Some of the more complicated and specific to python socket networking methods in btpeer.py are shown in algorithm 2, 3, and 4.

Algorithm - 2 mainloop (btpeer.py)

Input - self, serverport
Output - None

```
1: s = self.makeserversocket( serverport )
2: s.settimeout(2)
3: while (not self.shutdown) do
4:    clientsock, clientaddr = s.accept()
5:    t = threading.Thread( target = self.handlepeer, args = [ clientsock ] )
6:    t.start()
7:    if (KeyboardInterrupt) then
8:       self.shutdown = True
9:       continue
10:   end if
11: end while
```

The second algorithm is the groundwork for connecting to new peers. The goal of this method is to accept any incoming connection requests. Lines 1 and 2 create the server socket which allows a user to create client sockets which connect a user to peers. The main loop starts on line 3 which only stop if the user closes the application. Line 4 accepts any incoming request for connection and saves the client socket to the variable clientsocket and the client address to the clientaddress variable, though the latter is never used. Line 5 creates a new thread that calls the handlepeer method with the client socket as an argument as soon as it is started. The line after starts the thread. The next method, handlepeer, is easiest to understand after the next methods are introduced.

Algorithm - 3 sendata (btpeer.py)

Input - self, msgtype, msgdata
Output - None

```
1: msg = self.makemsg( msgtype, msgdata )
2: self.sd.write( msg )
3: self.sd.flush()
```

A network does not work correctly if data can not be sent between peers. Algo-

rithm 3 and 4 are called senddata and recvdata respectively because they send and receive data from nearby peers. To understand this best, it's important to know what the self.sd variable is. This is a necessary file that is created specific to each socket connection for each peer. The code used to create it is, "self.sd = self.s.makefile('rwb', 0)". The self.s is the specific socket that connects a user to a peer. makeFile with the args 'rwb' and 0 is what creates the aforementioned file. 'rwb' specifies that the file should be open in read, write, and and bytecode mode. All data transmitted between a user and peer is first encoded into bytes. This can be easily seen in algorithm 3 and 4. In algorithm 3 (sendata) at line 1 you can see that the input msgtype and msgdata are sent to the method makemsg. Line 6 of algorithm 5 shows this conversion of a message into bytecode. The msg variable is encoded with the message length, message type, and message data. All these elements are required for the receiving peer to be able to read the data and send it to the relevant functions.

Algorithm - 4 recvdata (btpeer.py)

Input - self
Output - (msgtype, msg)

```
 1: msgtype = self.sd.read( 4 ).decode('utf-8')
 2: lenstr = self.sd.read( 4 )
 3: msglen = int(struct.unpack( "!L", lenstr )[0])
 4: msg = ""
 5: if (msgtype == 'FILE') then
 6:    msg = b""
 7:    while (len(msg) ! = msglen) do
 8:       data = self.sd.read( min(2048, msglen - len(msg)) )
 9:       if (not len(data) then
10:          break
11:       end if
12:       msg + = data
13:    end while
14: else
15:    while (len(msg) ! = msglen) do
16:       data = self.sd.read( min(2048, msglen - len(msg)) )
17:       if (not len(data) then
18:          break
19:       end if
20:       msg + = data.decode('utf-8')
21:    end while
22: end if
23: return (msgtype, msg)
```

Line 2 and 3 of algorithm 3 write the encoded message (which is bytes) to the sd makefile and flushes it. Flushing a file clears out the buffer. In the case of sockets, it saves the data to the file without needing to close it. This data is send through the socket, and shows up in algorithm 4, recvdata. This method does not have any

inputs because it reads everything from the makefile established earlier. It reads the bytes where each of the variables were encoded before, and decodes them. The first two adhere to a standard protocol of containing 4 bytes. The third variable is msglen which is found by unpacking the lenstr and turning it into an int. The highest possible number represented with 4 bytes is around 4.3 billion so the file size able to be transmitted can be very large. From here there are two main blocks, one for messages of type "FILE" and the rest. The reason for this can be seen on line 6 and 20. The messages that contains file contents are kept in bytecode. This will be explained later. In both of these cases there is a simple while loop that essentially reads the self.sd makefile until there is nothing left to read. Finally, the message type and message itself are returned. These methods are necessary for fluid communication between a user and their peers. Now that the groundwork for basic communication has been established, more complicated communications are set up in btfiler.py. This file extends upon btpeer.py and adds a communication protocol with defining message types.

Algorithm - 4 makmsg (btpeer.py)

Input - self, msgtype, msgdata
Output - None

```
1: msglen = len(msgdata)
2: if (msgtype != "FILE") then
3:     msgdata = msgdata.encode('utf-8')
4: end if
5: msgtype = msgtype.encode('utf-8')
6: msg = struct.pack( "!4sL%ds" % msglen, msgtype, msglen, msgdata )
7: return msg
```

3.2 Networking Protocol

NAME	Requests a peer's ID
LIST	Requests a peer to list their neighbor's IDs
JOIN	Request to become a peer's neighbor
QUER	Request to search for a file on the network
RESP	Response to a search
FGET	Request a file
FILE	Reponse to a file request
QUIT	Request to be removed from a peer's neighbors
PING	Request to check if a peer still exists
ERRO	(Response) Notify a peer that there was an error with their request
REPL	(Response) Notify a peer their request was successfull

Table 3.1: Message types (btfiler.py)

Table 3.1 shows all of the Message types supported. Each of these are either requests which require some kind of response from a peer or user or responses in reaction to these requests. There are currently four responses and seven request message types available. They can be divided into three main categories, basic connection, file searching and downloading, and checking for validity. NAME, LIST, JOIN, PING, and QUIT all relate to establishing basic connecitons; QUER, RESP, FGET, and FILE all contribute to searching and delivering files; ERRO and REPL are used to notify the requestee if their request was a success or not. When put together all of these create a P2P network capable of finding and retaining peers to download files from.

Algorithm - 5 handlepeer (btpeer.py)

Input - self, clientsock
Output - None

```
1: host, port = clientsock.getpeername()
2: peerconn = BTPeerConnection( self.myid, host, port, clientsock, debug=False )
3: msgtype, msgdata = peerconn.recvdata()
4: if (msgtype notin self.handlers) then
5:    self.debug( 'Not handled: %s: %s' % (msgtype, msgdata) )
6: else
7:    self.handlers[ msgtype ]( peerconn, msgdata )
8: end if
9: peerconn.close()
```

Algorithm 5 is from btpeer.py but it is the method that seperates the message types from each other and calls the correct methods in btfiler.py that pertain to the message type. The first couple lines get the necessary variables required to call these methods such as msgtype, msgdata, host, and port. The peerconn variable uses the socket to create a peerconnection object which adds the senddata and recvdata methods. On line 4 it checks to see if the message type is a valid one. If it is not, a message to

the debugger is written describing the message type that is not handled (line 7). If it does exist, the method equal to the message type is called with the peercon and msgdata as input. The self.handlers variable is initialized when the peer is created and it is a dictionary that associates the message type codes in Table 1 to their respective method.

One of these such methods is called handlequery. It is algorithm 6 and it is responsible for receiving search queries and propogating them to nearby neighbors. Threads are created for each search to allow the current thread to receive more incoming requests. One of the main methods that handlequery calls is processquery which does the actual file search of the user's local files. Since most of the processing is done in processquery, this is known as a helper function and is required for multi-threading to work correctly.

3.3 File Searching

Algorithm - 6 handlequery (btfiler.py)

Input - self, peerconn, data
Output - None

```
1: peerid, key, ttl = data.split()
2: peerconn.senddata(REPLY, 'Query ACK: %s' % key)
3: if (error) then
4:    peerconn.senddata(ERROR, 'Query: incorrect arguments')
5: end if
6: t = threading.Thread(target=self.processquery, args=[peerid, key, int(ttl)])
7: t.start()
```

Below is algorithm 7 as mentioned above. This is quite a lengthy function and the goal of it is to facilitate file search requests and propagate them to other peers if the file is not found. Firstly, the main for loop shuffles through all the files that are locally stored. The next line checks to see if the key (the search query) matches the file name. If it does, fpeerid (file peer id) is set equal to key value pair of self.files at fname. Essentially, it makes sure everything is consistent between fpeerid and the locally stored file. Also it adds on any peer id's that it is aware of that also own the file. This will be explained more later. Line 12 is where the user sends their response in the nature of the RESP message type. On line 11 fpeerid is checked to see if it is None. The only way this is the case is if the key never matched any of the local files. It will also exit if it comes to this point because there is no need to propagate if the file has been found. In the case that does not happen, the message will be propagated to all known peers of the user and the ttl will be reduced by one. This is to make sure that a search query does not propagate endlessly. For larger networks this can be raised to a higher number but if users have a sufficient amount of peers, a number higher than around 5 or 6 is not needed due to the nature of exponential growth. It was mentioned before that line 12 sends a RESP typed message to the peer which was searching for the file. This received by qresponse and is explored in the next paragraph.

Algorithm - 7 processquery (btfiler.py)

Input - self, peerid, key, ttl
Output - None

```
1: fpeerid = None
2: for (fname in self.files.keys()) do
3:    if (key == fname) then
4:       fpeerid = " ".join(str(x) for x in self.files[fname])
5:       if (len(self.files[fname]) == 1) then
6:          fpeerid + = " " + str(self.myid)
7:          host,port = peerid.split(':')
8:       end if
9:    end if
10: end for
11: if (fpeerid) then
12:    self.connectandsend(host, int(port), QRESPONSE, '%s %s' % (fname, fpeerid),
       pid=peerid)
13:    return
14: end if
15: if (ttl > 0) then
16:    msgdata = '%s %s %d' % (peerid, key, ttl - 1)
17:    for (nextpid in self.getpeerids()) do
18:       self.sendtopeer(nextpid, QUERY, msgdata)
19:    end for
20: end if
```

Algorithm 8 is qresponse and is triggered when handlepeer receives a RESP typed message. This method is very similar to processquery but instead of sending a message that a file was found, it saves the file to self.files and makes sure to list all the peers which own that file.

Algorithm - 8 qresponse (btfiler.py)

Input - peerconn, fname, fsize, fpeerids
Output - None

```
1: if (fname in self.files) then
2:    for (fpeerid in fpeerids) do
3:       if (fpeerid notin self.files[fname]) then
4:          self.files[fname][1].append(fpeerid)
5:       end if
6:    end for
7: else
8:    self.files[fname] = [fsize, fpeerids]
9: end if
```

3.4 File Transferring

Now we arrive at the third and final file, filergui.py. This file contains all of the code that creates the window GUI and the methods that are activated when certain buttons are pressed on said UI. The main method to launch the whole program is also here. An important method related to the capabilities of file sharing is onFetch (algorithm 9). This method essentially requests every single file chunk required to build a full file. The second line has the main while loop that continues until the maximum index has been achieved. This is the same as all the file chunks being requested. Each loop of the while loop, a for loop starts which creates a new thread and sequentially requests the next file chunk from the next peer which has the file locally by using the method multiFetch (algorithm 10. This way the file download is spread out equally among all peers known to own the file. Once the maxIndex is achieved (all file chunks have been requested) the loop is exited and the method is complete.

Algorithm - 9 onFetch (filergui.py)

Input - maxIndex, files, fname
Output - None

```
1: i = 0
2: while (i <= maxIndex do
3:    for (peer in files[fname][1]) do
4:       host, port = peer.split(":")
5:       t = threading.Thread( target = self.multiFetch, args = [ host, port,
         FILEGET, "%s,%d"
6:       t.start()
7:       i + = 1
8:    end for
9: end while
```

The multiFetch function is multithreaded by onFetch (algorithm 9) on line 5. This is the main Fetch function that writes the received file chunks to memory. Line 1 one of this method makes the request of the specified file chunk which returns the response with the file chunk in variable resp. First it makes sure that the response is a file chunk and if so will open the local file that the user is building. The seek method is used to move the file to the correct position for where the file chunk is meant to go. The chunk size is set to 1,000,000. The file chunk size can vary and in the context of this program does not matter much. That said, on a larger scale programmers choose specific file chunk sizes to best suit their P2P file sharing network's needs. After the desired location is found the chunk is written and saved. The request made on line 1 of this function takes us back to btfiler.py and specifically, handlefileget (algorithm 11)

Algorithm - 10 multiFetch (filergui.py)

Input - self, host, port, msgtype, msgdata, fname, index
Output - None

```
1: resp = self.btpeer.connectandsend( host, port, msgtype, msgdata)
2: if (len(resp) and resp[0][0] == 'FILE') then
3:    fd = open( str(self.btpeer.serverport) + "/" + fname, 'r+b' )
4:    fd.seek(index * 1000000)
5:    fd.write( resp[0][1] )
6:    fd.close()
7: end if
```

Algorithm 11 handles all FGET message types like the one sent in algorithm 10. The goal of the method is to retrieve the file chunk that is requested and send it to the peer that requested it. The input data contains the requested file name and the start index of the file chunk. This means that this user will be sending this particular chunk of file back to the requestee. On line 2 the start index is multiplied by 1,000,000 because this is the specified file chunk size. The method then checks to see if the file exists within self.files. If it does, the method proceeeds as usual but if not, an ERRO response is sent to the file requestee and the method exits. If the file does exist locally it is opened in read-byte mode because this is how files are stored in this project. The seek method is then called on the file to move the current position of the file to the start of the chunk using the sIndex variable. The specified amount of bytes is read to fit the chunk size and is sent back to the requestee of the file.

Algorithm - 11 handlefileget (btfiler.py)

Input - self, peerconn, data
Output - None

```
1: fname, sIndex = data.split(",")
2: sIndex = int(sIndex) * 1000000
3: if (fname not in self.files) then
4:    peerconn.senddata(ERROR, 'File not found')
5:    return
6: end if
7: filepath = str(peerconn.id).split(":")[1] + "/" + fname
8: fd = open(filepath, 'rb')
9: fd.seek(sIndex)
10: str(os.path.getsize(filepath))
11: filedata = b""
12: filedata = fd.read(1000000)
13: fd.close()
14: return
15: peerconn.senddata(SENDFILE, filedata)
```

Another very important method in btfiler.py is updateFileList. This method (al-

gorithm 12) is more concerned with UI than with the backend. It shows the user what files they have added to their client locally and which they are able to download. Every time a change is made to the list of files locally occurs this method is called to update the user. The visual format of files is defined as follows, "fname:fpeerid1:fpeerid2:..." where fname is the file's name, and fpeerid1, 2, ... are the peers which own the file. A peer's id is added to a file name when that peer returns back a successful search result.

Algorithm - 12 updateFileList (btfiler.py)

Input - self
Output - None

```
1:  for (f in self.files) do
2:      p = ""
3:      if (len(self.files[f]) == 1 or os.path.exists(str(self.btpeer.serverport) + "/" +
        f):) then
4:          p = '(local)'
5:      else
6:          for (x in self.files[f][1]) do
7:              p + = x[x.index(":") +1:] + ","
8:          end for
9:          p = p[:-1]
10:     end if
11:     self.fileList.insert( END, "%s:s%" % (f,p))
12: end for
```

3.5 UI

The UI was created to be very simple yet powerful. There are 6 buttons in total which can be seen in figure 3.2. The add button (above the search button) will add a file to your available file list if the file name inside of the text box to the left matches that of a file inside of the folder named your client's ID. This is demonstrated in figure 3.4. The folder is created in the same location as the btfiler.py python file. Below the add button is the search button. This button allows you to search for any files on your peer's available file list. After you successfully search for a file, it is added to your available file list but with the port number of each peer which has it in their available file list. This is demonstrated in figure 3.5. The fetch button below the available file list will download the selected file from the peers listed next to the file on the available file list. This is demonstrated in figure 3.6. On the right, there is the remove button which removes the selected peer from your peer list, thus disconnected from them. Refresh will update your Peer list to accurately reflect the peers available on the backend. Lastly, the rebuild button will rebuild a client's peer list based on peer typed in the text box which is equivalent to that peer being the peer-ip:port argument when launching the program.

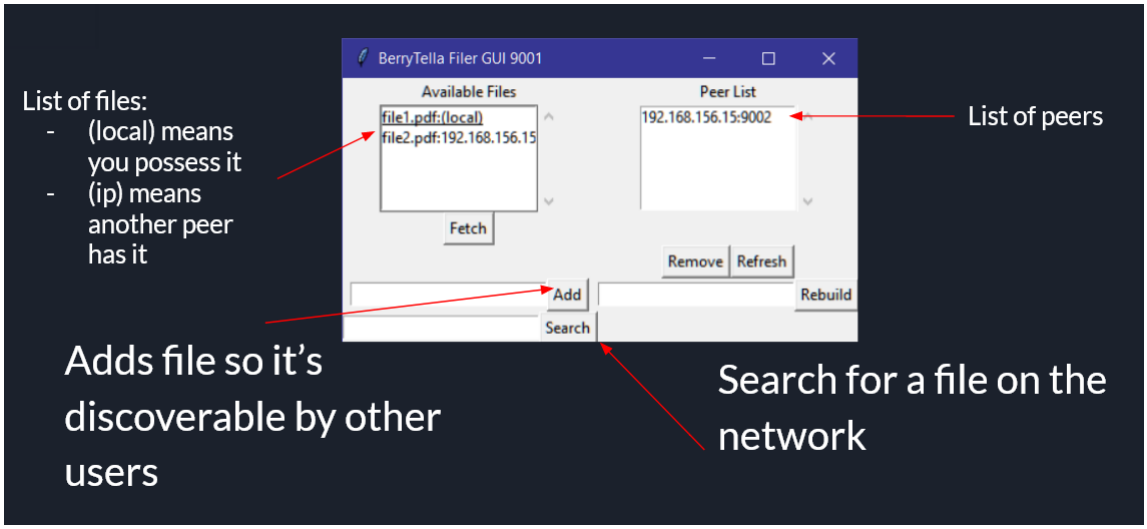

Figure 3.2: Diagram of UI

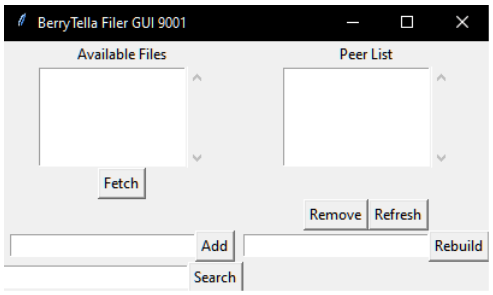

Figure 3.3: The plain GUI without connecting to anyone

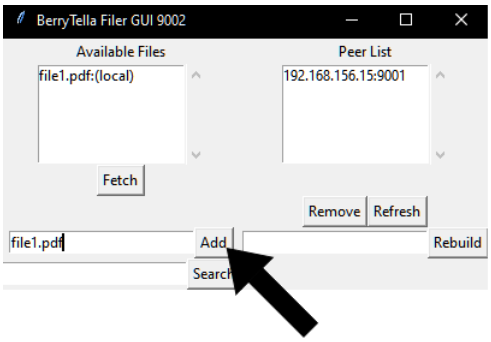

Figure 3.4: Pressing Add will add your local file to your list of available files

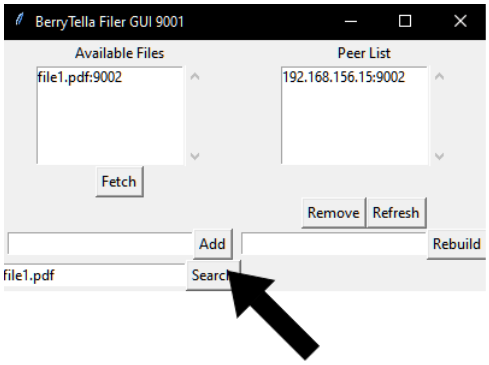

Figure 3.5: Pressing search will ask nearby peers if they have "file1.pdf"

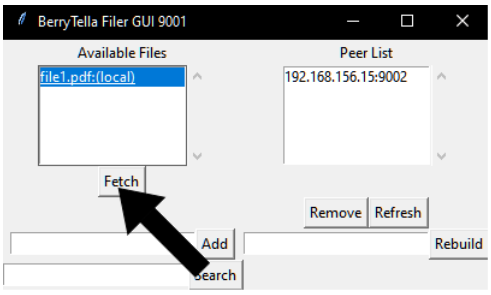

Figure 3.6: Pressing Fetch will download the file from peers listed next to the file name

Chapter 4

Experimental Results

4.1 Experimental Design

When beginning this project, there were many different ideas as to how to evaluate this project's success. To begin with, a straightforward approach of evaluation was to conduct experiments delving into the efficiency of a file sharing network utilizing the abilities of a client-server network versus a peer-to-peer network. Originally, the goal was to conduct a real world test using different computers and IP's to transfer files between users. As it turned out, this was just not viable. The most difficult aspect of creating a P2P network is testing because in order to effectively test one, you must have tens of hundreds of users. Comparing this to client-serer networks which only need a server and one single client to be tested, the hurdle to attain hundreds of users is unrealistic.

To reconcile this issue, all testing was done locally. This means the IP address for all peers was localhost and the only part of a peer's ID that differed was their port number. When this is done, files are transferred locally instead of over a network so file transfer is just inhibited by the the user's drive speed. This means the total bandwidth does not go up even when adding more peers to the network because the bandwidth is determined by the drive, not the network speed. In order to solve this problem peers were forced to sleep for a certain amount of time after transferring a file chunk to simulate a user's limited network bandwidth. There was also another factor that controlled a user's transfer and download speed, the chunk size. For all of the tests, a chunk size of 1 MB was used. This can be seen in the Method of Approach section in algorithms 10 and 11. Many different chunk sizes were tested all the way up to 5 MB. As soon as the 4 to 5MB mark was approached the transferred files became corrupted so a chunk size of 1 MB was settled upon. If the chunk size is kept consistent and each peer is put to sleep for the correct amount of time (to simulate download speed), the only variable that needs to be manipulated is the file size that is downloaded. An assortment of file sizes were chosen to best show where potential bottlenecks may be present. The file types used do not affect network performance so different videos with assorted sizes were used because manipulating file sizes of a video is very simple. After every file transfer while testing there must always be a validity test to make sure the file is still intact and and usable.

Table 4.1: Percent Difference between Expected and Actual Results

Number of Peers	10 MB	100 MB	1 GB	5 GB
1	0.94	10.34	103.50	485.43
2	0.55	5.32	54.30	274.43
4	0.27	2.74	29.83	142.92
6	0.19	2.01	20.34	98.49
8	0.16	1.58	16.39	75.33

4.2 Evaluation

Above in table 4.1 are the results from testing. As you can see, there were 4 different files that were used for testing. These were video files each chosen to be exactly the specified file size in the table. To verify each video files validity after transfer, they were opened to make sure they were not corrupted. On top of this, the files bytecode was compared to see if there was any difference between the original file and the transferred one. For all the results above, there were no discrepancies found between the original and the transferred version.

On the left hand side of the table you can see the number of peers present during transfer. This should be interpreted as the row with 1 peer being a simulated client-server network and the rest of the rows being P2P networks with different numbers of peers. The results are measured in seconds to download. The timer began as soon as the fetch button was pressed and the timer ended when the last file chunk was saved in the transferred file. Each result is the average of results from 3 trials. At a glance, it appears that the download speed is roughly 10 MB/sec for one single peer. This is intended as the thread.sleep was timed in such a way to simulate a download rate of 10 MB/s for each peer. If we look to the next trial with 2 peers, the download time is approximately halved. This makes sense, since every peer's simulated download speed should be around the same. The same halving can be seen to the next row, where there are 4 peers instead of 2. Each row after that reduces the time by around 1/3rd and then 1/4th. This trend is easy to see in the graphs further below labeled figure 4.1, 4.2, 4.3, and 4.3. The relationship between download time and number of peers can be described by equation 4.1 where eET is expected elapsed time and AvgDS is the average download speed of each peer. This relation ship can be seen more easily on the next few pages where these results are graphed on plots. The first few peers show a drastic decline in the trend but each extra peer shows a slightly less effective drop. The bonds of each graph are designed to fit the file size in a proportionate manor so the graphs look astoundingly similar.

$$eET = (FileSize/AvgDS)/numPeers \tag{4.1}$$

Since the download speed of each peer is simulated to be 10 MB/sec and every other variable is known, the eET can be calculated for each of these trials and can be compared to the actual results. A percent difference calculation comparing the expected elapsed time and the experimental elapsed time was conducted and table 4.2 shows the results.

Table 4.2: Percent Difference between Expected and Actual Results

Number of Peers	10 MB	100 MB	1 GB	5 GB
1	-6.38	3.29	3.38	-3.00
2	9.09	6.02	7.92	8.90
4	7.41	8.76	16.19	12.54
6	12.28	17.08	18.06	15.39
8	21.88	20.89	23.73	17.03

This table uses equation 4.2.

$$PercentDifference = (ActualResult/ExpectedResult)/ActualResult * 100 \quad (4.2)$$

Some of the percentages in table 4.2 are negative because the actual result were below the expected result meaning that the average peer download rate was actually faster than the set 10 MB/s. The reason for this could simply be the overhead for each peer for those trials was lower than expected so the thread sleep method to force a 10 MB/s download rate was not enough to keep it to 10 MB/s. Since that there are only 2 trials this occurred in it can be assumed that this possible error does not pose a risk to the efficacy of these results. A significant trend can be seen as the amount of peers increases the percent difference also increases in in all four of the test cases. This can most likely be attributed to a higher overhead as the more peers there is also means more sockets need to be monitored as well as incoming requests and responses. It can be reasonably assumed that with a more optimized and well thought out P2P network that this issue of overhead becoming higher the more peers a user connects to can be largely minimized.

Another trend that can be seen is as the downloaded file size increases, the increase of the percent difference decreases when compared to smaller file sizes. This is most likely because because of the law or large numbers. The law of large number states that as a sample size increases, the percent error will decrease. So even though all these results are averages of 3 trials, increasing the file size shows a decrease in the amount of error which makes sense.

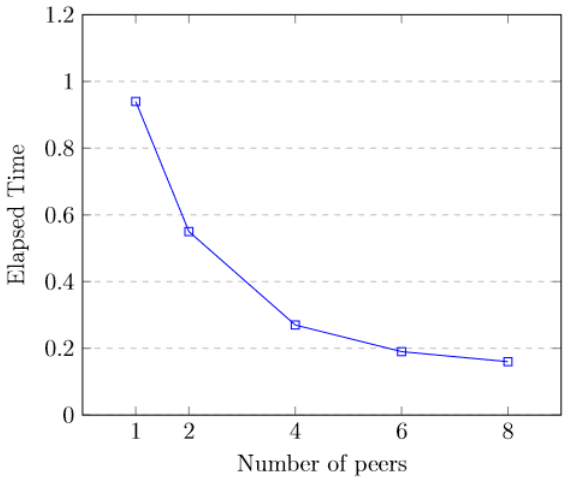

Figure 4.1: Number of peers vs. Elapsed Time for 10 MB file

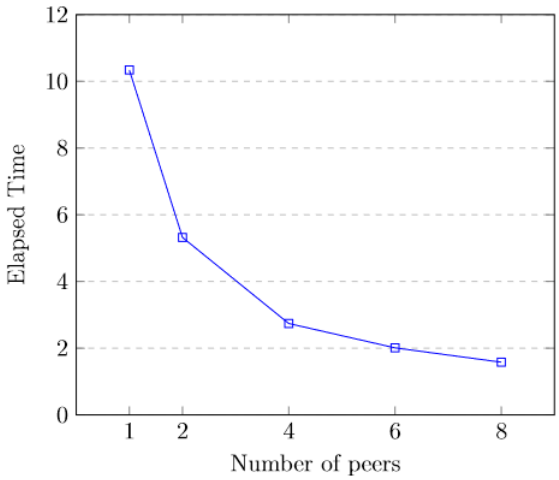

Figure 4.2: Number of peers vs. Elapsed Time for 100 MB file

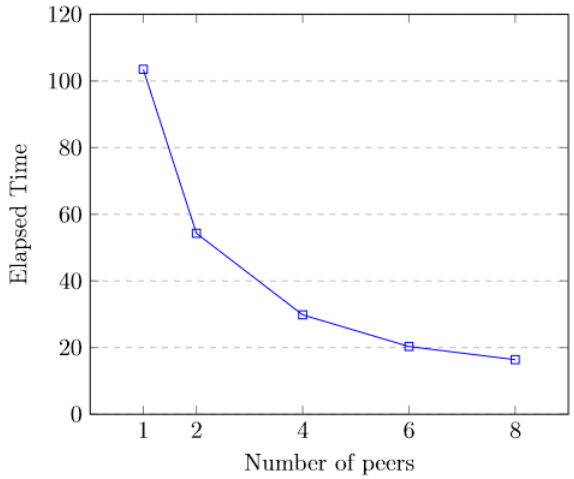

Figure 4.3: Number of peers vs. Elapsed Time for 1 GB file

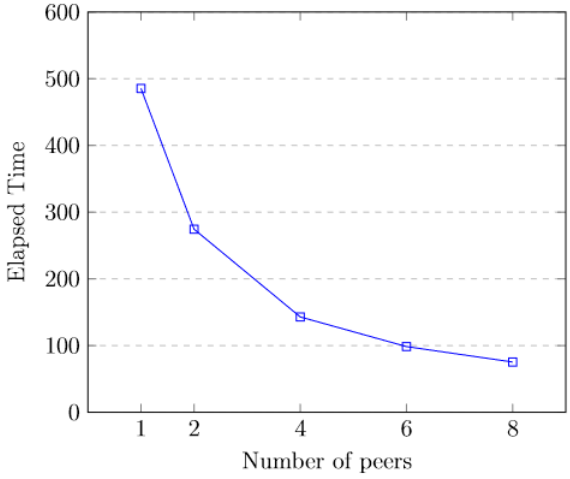

Figure 4.4: Number of peers vs. Elapsed Time for 5 GB file

4.3 Threats to Validity

The validity of results is extremely important for the efficacy of research in general. In the case of this project, the validity is unfortunately lower than expected. This can be attributed to a few things such as raw computer performance and overhead processes affecting the result. In an ideal environment, the processes to test the P2P network would be the only ones present to minimize this threat to validity but this is not realistic. The machine used to complete these tests was running Windows 10 as well as a web-browser to record the results of each trial. In addition to this each time a trial was completed the efficacy of each file downloaded had to be verified so the file explorer had to open as well. Background processes of windows had a high probability of creating small disturbances in performance thus raising some download times and reducing others if not present. The raw compute power of the machine also

plays a major role since the test was done completely locally. The raw compute power is mostly affected by drive speed. This particular machine was running on a solid state drive which can be much faster than a hard drive when dealing with transfer speeds. Not only this but the size and current free space on the drive can also have a drastic impact on real world performance.

If these results were attempted to be replicated, these are some issues to look out for. The results would primarily change depending on the raw performance attainable by the machine used.

Chapter 5

Discussion and Future Work

5.1 Summary of Results

To look back, one of the large goals of this project was to create a lightweight P2P file sharing network using python. I believe this goal was met. This network is capable of transferring large files between peers at speeds directly proportional to the number of peers on the network that have the file. The next large goal was to test this network in the real world on a large scale. Unfortunately this was not possible. The sheer scale and difficulty of testing P2P networks is significant hurdle to overcome while they are in development. Compared to testing a client-server file sharing network where you only need one peer to test file downloads a P2P network is orders of magnitude more difficult.

This hurdle may be one of the reasons that P2P networks are not as popular as traditional client-server networks. Testing is integral to deploying a network on a large scale and the increased difficulty to test and more importantly get users to bolster the networks performance may have been too great for some of the early P2P networks to attain a place among modern society. On top of the testing difficulty, P2P networks rely on their users to sustain a network. Sustain in this case means supplying files and bolstering download speeds. Without users, many of the benefits a P2P network boasts are not possible.

Another goal of this project was to implement some advanced features that Bittorrent utilizes such as verifying file hashes, network topographies like the Kademlia architecture, and P2P incentives to promote file sharing. These advanced algorithms did not get implemented. Many of these feature rely on many users to able to test them on the network. For example, a network topography for a P2P network dictates which peers should connect to a specific user in such a way that every user is 4 or 5 jumps away from any other user. In order to test a feature like this, lots of users have to be on the network.

Despite failing to meet some of these goals, there is lots of data that this P2P network has created. This data points to many different significant results. One of them being that when creating a P2P network one must be sure to reduce the overhead when increasing the amount of peers. This is needed desperately if the P2P network is going to scale especially past the initial testing phase. Another result that is important for future P2P networks is being aware of the difficulty in testing. Being aware of this,

future efforts to create P2P networks will be prepared to create simulations in order to successfully test complex algorithms that apply to P2P networks. These are both very valuable results for the future of P2P technology.

5.2 Future Work

The future work to step off this project is actually very bountiful. To start, some of the more complicated algorithms could be implemented before a full scale test is conducted with many users on the network. This is a very lightweight Bittorrent adjacent P2P platform so it could be used by a company as a quick and efficient file transferring tool. There are many expensive file transferring solutions and this P2P platform only needs python to function. While it would not be as simple as copy and paste, bringing this application to the mobile platform would make it widely more accessible and therefor easier to test since there are more users that would be willing to participate. Further research into one of the largest P2P file sharing networks called Bittorrent should be done to sufficiently gather data on how to successfully test and deploy a P2P technology such as this one.

5.3 Conclusion

Overall, I believe this project was a success. Much was learned from both the goals met and the goals not met. The original motivation for this project was to determine if P2P networks can replace client-server networks in many parts of the Internet. It has been thoroughly demonstrated as a proof of concept that P2P networks possess both the robustness and scalability required to achieve such a feat in regard to file sharing. File uploading and downloading is a large part of what makes the Internet the Internet. A summary of the current work in the field has been done which also supports this conclusion. Bittorrent was presented as a large scale, global file sharing network that this lightweight framework created specifically for this project attempts to imitate. Bittorrent is proof that this P2P network has the ability to be tested sufficiently and successfully so the future work related to this network is viable.

Bibliography

[1] L. Marshall, "The effects of piracy upon the music industry: A case study of bootlegging," *Media, Culture & Society*, vol. 26, no. 2, pp. 163–181, 2004.

[2] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, Berkeley, CA, USA, 2003.

[3] F. R. Santos, W. L. da Costa Cordeiro, L. P. Gaspary, and M. P. Barcellos, "Funnel: Choking polluters in bittorrent file sharing communities," *IEEE Transactions on Network and Service Management*, vol. 8, no. 4, pp. 310–321, 2011.

[4] J. A. Johnsen, L. E. Karlsen, and S. S. Birkeland, "Peer-to-peer networking with bittorrent," *Department of Telematics, NTNU*, 2005.

[5] T. Locher, P. Moore, S. Schmid, and R. Wattenhofer, "Free riding in bittorrent is cheap," 2006.

[6] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do incentives build robustness in bittorrent," in *Proc. of NSDI*, vol. 7, p. 4, 2007.

[7] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu, "Influences on cooperation in bittorrent communities," in *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pp. 111–115, 2005.

[8] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee, "Bittorrent is an auction: analyzing and improving bittorrent's incentives," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pp. 243–254, 2008.

[9] A. Legout, N. Liogkas, E. Kohler, and L. Zhang, "Clustering and sharing incentives in bittorrent systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, pp. 301–312, 2007.

[10] W. Liu, D. Peng, C. Lin, Z. Chen, and J. Song, "Enhancing tit-for-tat for incentive in bittorrent networks," *Peer-to-peer networking and applications*, vol. 3, no. 1, pp. 27–35, 2010.

[11] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang, "Exploiting bittorrent for fun (but not profit)," in *Proc. of IPTPS*, 2006.

[12] A. Grunthal, "Efficient indexing of the bittorrent distributed hash table," *arXiv preprint arXiv:1009.3681*, 2010.

[13] B. Wong, A. Slivkins, and E. G. Sirer, "Approximate matching for peer-to-peer overlays with cubit," tech. rep., 2008.

[14] S. R. Wilkinson and J. S. Almeida, "Qmachine: commodity supercomputing in web browsers," *BMC bioinformatics*, vol. 15, no. 1, pp. 1–12, 2014.

[15] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@ home: Lessons from eight years of volunteer distributed computing," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–8, IEEE, 2009.