CPSC 559 Design Document

A peer-to-peer file sharing system

David Nguyen, Inderpreet Dhillon, Gary Li, Michael Rediron, Kell Larson

INTRODUCTION

We use a peer-to-peer (P2P) system to let users download and upload files. Our application is intended for people who want to access files with speed by taking advantage of a P2P setup. The application that we made is intended for technical users as running the program will be done from the command line. It also requires a few additional setups that includes port forwarding to be able to run our application. Knowledge of networking and a bit of java code is strongly recommended to ensure that the program is executing smoothly and properly.

SYSTEM ARCHITECTURE

Using a P2P network can be very useful in a distributed system. As opposed to your typical client-server architecture where all client requests are handled by one server, P2P allows every computer to act both as a client and a server (sometimes referred to as a node or peer). We use this type of network to download and upload files, something like bittorrent. Our system is similar to the Gnutella protocol but with a few modifications. The network is unstructured meaning peers can join by connecting to a random subset of nodes already in the network. This makes it nice and simple for our project and implementation but there are disadvantages with it. Since there is no correlation between the peer and the content, you have to search the entire network to find a file (sometimes referred to as flooding the network). This is no problem if the content is very popular as you will most likely get a query hit very quickly, but with a not so popular file you are never guaranteed whether to find it or not. As mentioned already, we have a few modifications and that is the introduction of a global tracker. The global tracker is a dedicated server that we assume will always be running and reachable. The purpose of the global tracker is to give a requesting peer a random subset of nodes in the network so that a file query can be done. Other than the global tracker, our system is completely decentralized and users will connect appropriately to get their files without the need for a main server.

TYPES OF PROCESSES

Seeder - A seeder is someone who already has the complete file and can give the content to a requesting peer.

Leecher - A leecher is someone who does not have the complete file and is downloading the file content from a seeder.

Global Tracker - The global tracker keeps track of all existing nodes that are alive in the network and can help peers to join and connect to the network by giving them a random subset of these existing nodes.

File Tracker - A file tracker is someone who has the complete file and will have additional information for that file which includes a list of all peers that also own that file, the file name, and the current leader for that file.

Leader File Tracker - A leader file tracker is the 'main' tracker for that particular file and will be responsible to give a requesting peer the file size, file hash and the peer list for the file. The first person to upload the file will be the leader tracker for it.

What can a peer do

Join the network - A peer can join the network by getting a random subset of node addresses given by the global tracker and then connecting to their ips.

Downloading a file - A peer can download a file by querying their subset of nodes list to see if they own the file. Eventually either the leader tracker is found or the file does not exist. If the leader tracker is found, the peer will be given the peer list for that file which it can then connect to and start downloading the file.

Seed a file - A peer can seed the file once they have the complete file. While the peer is alive there will be requesting peers that will ask for some of the file content which it will then give to them.

Upload a file - A peer can upload a file by using the command line interface we built. It requires the peer to enter the path to the file being uploaded. Uploading a file essentially creates a file tracker for the file and the peer uploading will be the leader.

Leaving the network - A peer can leave the network by using the command line interface we built. The global tracker and leader file tracker will know if the peer leaves and will update their peer list accordingly.

There are a few assumptions we have made for this application. The first is that we assume nodes cannot delete a file while they are alive and running. The second assumption is that file names are unique. Lastly, due to time we were not able to build a search list for file names which means we assume that peers will know the file name they want to search for.

REPLICATION

Purpose of replication in our system

We use replication to ensure that downloads are fast, crashes are minimal in impact and that files are available. Downloads are sped up the more other nodes/peers have the file. With more nodes that have the file, the peer can split the file into more chunks and download them in parallel from each peer with the file thus increasing the download speed. Searching can also become faster because as more peers have the file, a requesting peer will have a higher chance of getting a query hit. Replication also serves a purpose for fault tolerance because for each peer that has the file, they all act as a backup in case some nodes decide to leave or crash. Finally, replication keeps the file 'alive'. Since our application has no database or process that stores the file persistently, this means that as long as there is at least one peer with the file, someone else is still able to download that file. However if the last peer with the file leaves or crashes, the file is gone forever until it is uploaded again thus replication can reduce the likelihood of a file being lost forever.

The advantage of replication is very powerful because as you can see the more the better. However a downside to this is that the benefits only occur for the files that are being replicated. Since no peer is forced to replicate a certain file unless they choose to download it, this means that some files will be better off than others.

Type of replication in our system

We have passive replication in our system. When replicating we are concerned with replicating the file trackers and the peer list. It is important to mention that by replicating a tracker it also implies that the file content is replicated as well.

In a typical scenario, a peer will query for the file until the leader tracker for the file has been found. Once found, the peer will connect to the leader tracker and ask for the file size, file hash and the peerlist for the file. With this information the peer can begin chunking up the file into parts and connect to the available peers in the peer list to start downloading. Once the peer has finished downloading the file it will then verify the hash of the downloaded file with the hash given originally by the lead tracker. If it matches then the peer will send a message to the lead tracker saying that it can now be a seeder for this file. The lead tracker will update its peer list to include the new seeder and also tell every other peer in the list to update their list as well. In the meantime, the new seeder will also be creating a tracker for this file.

As you can see the tracker is being replicated when the new seeder creates one after finishing the download. The peer list is also being replicated through the update command from the leader as well to all its followers.

Replication of processes

The replicated processes are the trackers. As explained above, the new seeder is now a tracker and is able to give query hits whenever another peer out there is requesting this file.

Replication of data

The replicated data are the peer list and the file content itself. The peer list is being replicated and updated each time the leader gets a new seeder. The file data and content is being replicated as the download occurs and eventually all of the data will be downloaded.

Communication

The lead tracker will be the one communicating between the nodes. Everything is done through the lead tracker for any file that is being downloaded. The requesting peer will get the file information from the lead tracker. The lead tracker is also responsible for letting the followers know when to update their peer list when a new seeder has joined.

FAULT TOLERANCE

How will our system detect failures

Our system will detect failures in three ways: heartbeats, timeouts, and hashes. Heartbeats are essentially messages that should be echoed back to the sender. Periodically each lead tracker in the system will send these messages to its followers to verify that they are still responsive and reachable. A node that doesn't reply to a heartbeat will be considered to be offline/crashed, which will cause it to be removed from any trackers that are keeping track of it. Timeouts in our system will be used to detect omission. To detect omission faults we will use a timeout after sending a message, if the timeout expires before a response then that server may not be sending or transmissions are being lost. In our system we do not have any timing faults since we are not synchronous. Finally, our system uses hashes along with concurrent agreement to detect byzantine faults. In our system a file is considered complete when all parts are retrieved and verified. Hashes are used to verify if a file is correct, by calculating the local hash and matching it with the known hash. A known hash is one that is agreed upon to be the correct one by all peers with that file. This agreement happens between a leader tracker and its followers, all will calculate a hash locally and one that shows up the most will be counted as the correct one.

How will our system deal with different faults

Crashes:

The global tracker will send out heartbeats to nodes, and if a node were to crash (by not replying to the heartbeat 5 times) it will remove it from the global tracker's peer list. This is to ensure that the crashed node will not be part of the random subset of node list that is given out by the global tracker.

If a leader tracker were to crash, a leader election would start. This election will be a bully algorithm. It will assign random numbers to the ips of the nodes with the same file, look for whoever has the highest number and make them the leader.

If a follower tracker were to crash. The leader will detect it through the heartbeat as every leader will be heart beating their followers for a particular tracker file. Once the leader finds out which node has crashed, it will update it's peer list to remove this node and tell everyone else to do the same as well. This will ensure that the peer list will always have existing and alive nodes to download files from.

Omission: If a server fails to respond to the message from the client, it will try to resend but after a certain amount of times it will skip the server and look for the next server available.

Byzantine: Our system does not deal with byzantine faults directly but it does inevitably resolve them. If a peer were to randomly modify a file, then this file has now been corrupted because any new peer downloading the content from this node will not have the final correct hash for the file. If we had more time we would implement a way to find out which node was the one responsible for the corruption and remove them from the peerlist. However, for now our system does not stop the problem, but it does prevent the problem from spreading. That is, preventing any other nodes from creating trackers and seeding this corrupted file because the hashes will not match.

How will our system recover from failures

A leader election will occur if it is a leader tracker. This will then update everyone else's peer list of the new leader. If the file has only one person and disconnects, the system will not be able to recover the file. If the server fails to respond to a leecher, via heartbeats, it will skip to the next server and will continue until a reply occurs.

Masking errors

The user will have no error messages sent, other than when a file hash downloaded is not correct. If the file hash does not match, this means that the file has now been corrupted. This file is no longer downloadable. If we had more time we would implement a way to find out which peer modified the file that made it corrupt and remove them from the peerlist.

CONSISTENCY

Type of consistency

Our system follows the eventual consistency model. We say this because for a set of nodes in a tracker group, either downloading or seeding, they will eventually have the same data for a given file. Further, all nodes downloading a particular file will eventually have an identical copy of that file given enough time and nodes for seeding.

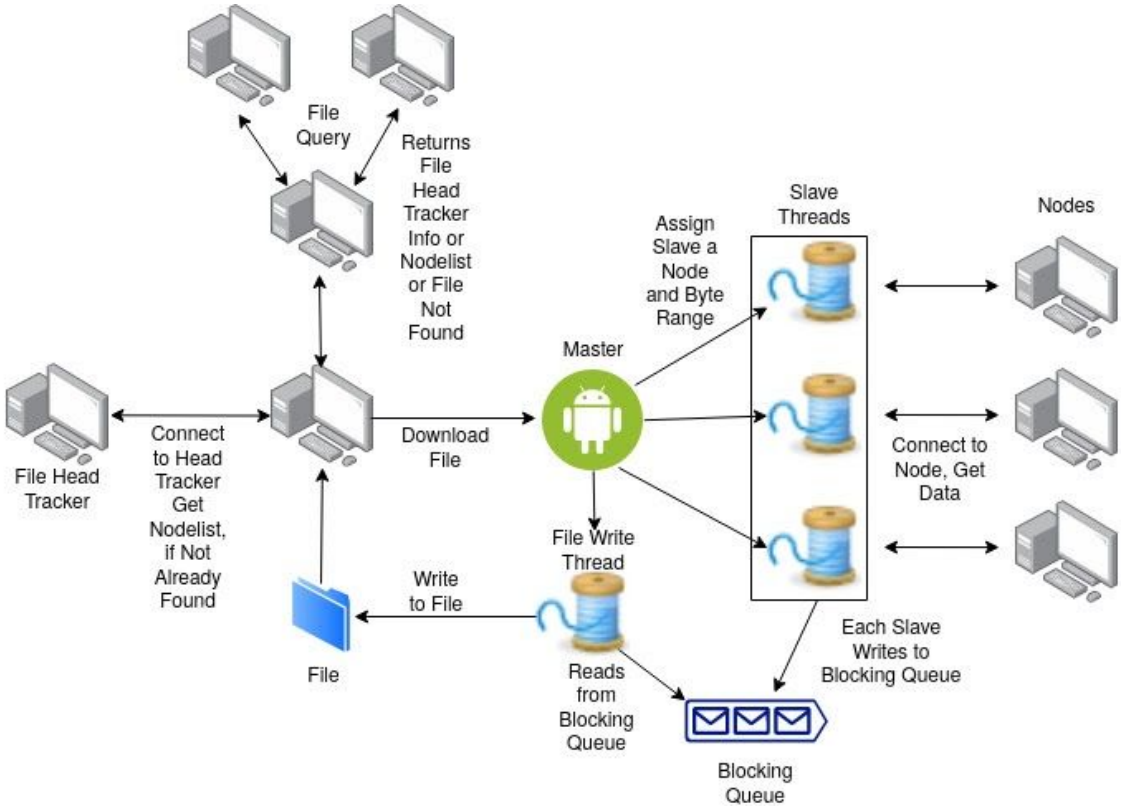
Consistency for files

This consistency is achieved through how files are transferred between nodes in the system. A requester node will ask some subset of the trackers of a file for some part of the file. This file part is sent along with a hash of that part, ensuring that the data has not been corrupted in transit. After all of the parts of a file have been retrieved the hash of that file is compared to the one given by the lead tracker. If the hashes match then the files are identical and consistency has been achieved for that file.

Consistency for trackers and peerlist

Once the leader tells it's followers to update to include the new peer that just finished the download, it is not necessary for all the followers to have this information available right away. It is only important that they eventually update their list. Even if the follower does not update their list, every peer requesting for a file will be redirected to the leader anyways and the leader will always have the most up to date list. Having a leader per file is an advantage in keeping our system consistent because any updates only come from one source, therefore we avoid situations where any follower can tell others to update which may lead to inconsistent peerlist per tracker at any given time.

IN-DEPTH OF OUR SYSTEM



This is a diagram of our network and how a file is leeched. For the center node, symbolized as a computer, to leech a file, it first queries its peers for a file which are the nodes above it in this image. If any peer is a tracker for that file, it returns the connection information for the lead tracker, otherwise it returns an error for file not found. Assuming the lead tracker information is returned, we then connect to the lead tracker which is shown on the left hand side in the image. We get the file size, file hash, and node list from the leader. Then the node locally creates a master who splits the download to slave threads. Each slave thread attempts to download a certain part of the file by requesting it from one node from the list. The slave then saves the data received to a blocking queue. Meanwhile, the master reads the data from the blocking queue and assembles the file locally. Once the file is entirely downloaded, the master hashes the file and sends the results to the lead tracker for verification.

COMMAND HANDLER

Each packet we send starts with a 4-byte command number, followed by a 4-byte number representing the length of the remaining data encapsulated in the UDP packet. We use this standard format so that we can develop a parsing algorithm that every packet sent and received can go through. The command number allows each peer to know what action to take and the length ensures the entire packet has arrived. We're using UDP since any commands sent can always be resent without issue and UDP is more efficient than TCP for transmitting large amounts of data.

Commands with variable length

Bytes {0..3}	Bytes {4..7}	Bytes {8..n}
Command Num	Msg Length	Message data

Commands with fixed length

Bytes {0..3}	Bytes {4..n}
Command Num	Message data

Complete message table with abbreviated descriptions and formats:

Name	Purpose	Cmd	Len	Data Structure (bytes)
Heartbeat	To check if a peer is still alive	00	0	N/A
Heartbeat reply	A reply to the heartbeat command	01	0	N/A
Request seeder	Request a seeder that can seed a specific file	05	n	Filename
Return seeder	Give a requester a seeder for a specific file	06	n	File size, hash, IP addresses of seeders
Request File	Request a file	10	n	Byte start index (4 bytes), Byte end index (4 bytes), String containing filename (variable bytes)

Send file chunk	Send a certain file chunk, after a request has been received	11	n	Packet sequence number (4 bytes), md5 hash of data (16 bytes), data (n bytes)
Resend file chunk	Peer requests packets to be resent	12	n	Byte start index (4 bytes), Byte end index (4 bytes), Packet sequence number (4 bytes), String containing filename (variable bytes)
Ready to seed	Peer sends to tracker it can seed a certain file to other peers	20	n	Filename
New leader found	New leader chosen through election	23	n	Filename, IP
Call election	Start election	24	n	Filename
Add peer	Leader says to add a peer	25	n	Filename, IP
Delete peer	Leader says to delete a peer	26	n	Filename, IP

WALKTHROUGH

Code

https://github.com/LighTec/CPSC559_Bittorrent

Initial setup

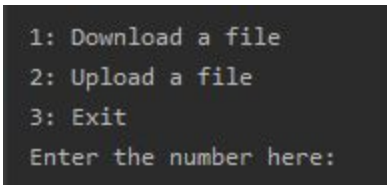
If you are running the system across computers on different networks then you will need to port forward. For the GlobalTracker TCP port 1962 has to be forwarded and for the clients UDP ports 6150 to 6100 need to be forwarded.

In NodeList you will need to replace “75.156.158.110” on line 19 with the public IP of the computer that is hosting the GlobalTracker server, if you are running the GlobalTracker locally then change it to “localhost” instead.

After these changes you will need to build all the files and then start GlobalTracker. After GlobalTracker is running you can start some Node instances to use.

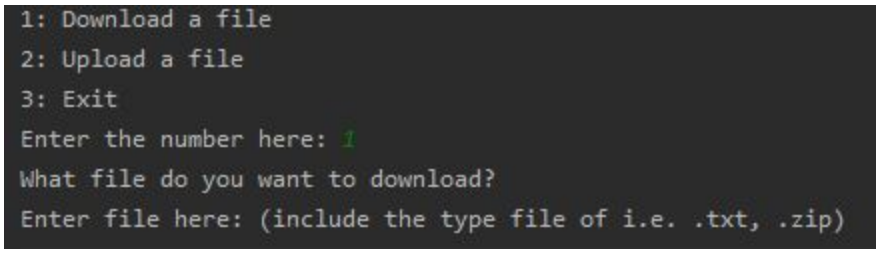
Running the program

Main menu

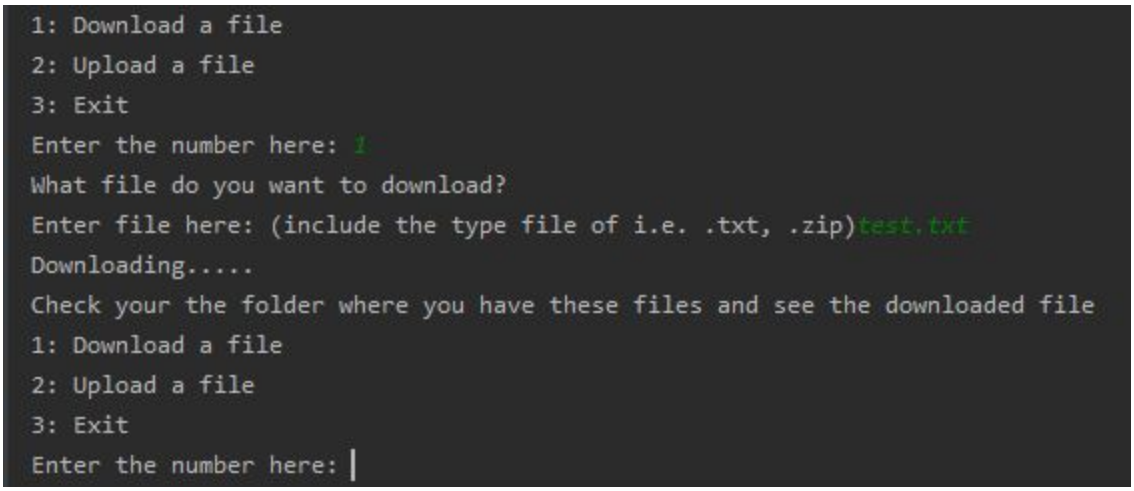


After starting the java file, the first thing you will see is this menu and you will be given 3 options to choose from. “Download a file” will let you download a file. “Upload a file” lets a file of yours be downloaded from others. “Exit” is to close the application. To access these options you put in the number.

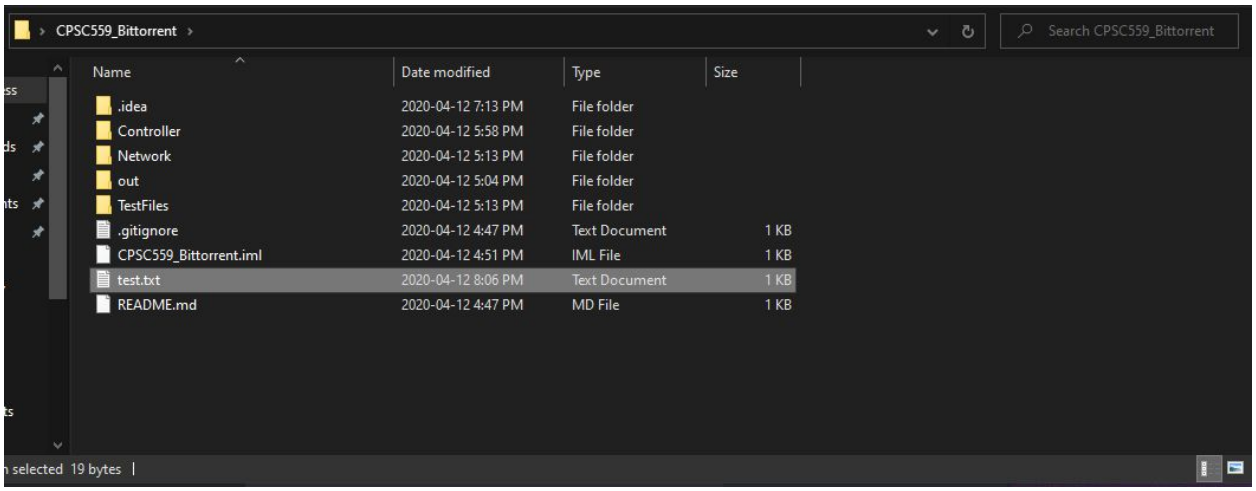
Download menu



If you were to put in 1 into the command line, you will go to the “download a file” menu and you can download a file. If there are no seeders for that file, you will get a file not found.



After you input a file name, it will redirect you back to the main menu and you can put in another number.



After it is done downloading, it will go to the root folder of where you have all the files.

Uploading menu

```
1: Download a file
2: Upload a file
3: Exit
Enter the number here: 2
What file do you want to upload?
Enter file here including the directory: (for example: .\TestFiles\alphabet.txt)
```

If you put in 2, it will take you to the “upload a file” menu. This lets a file from your computer be able to be downloaded from others.

```
1: Download a file
2: Upload a file
3: Exit
Enter the number here: 2
What file do you want to upload?
Enter file here including the directory: (for example: .\TestFiles\alphabet.txt)C:\Users\Gary\Desktop\CPSC559_Bittorrent\text.txt
Your file can now be seeded by others
1: Download a file
2: Upload a file
3: Exit
Enter the number here:
```

After you put in the location of the file and the name, other people can now download the file. Then it will redirect you to the main menu.

DEMO

What didn't work

Leader Election - **Fixed**

User adding more than one file - **Fixed**

User seeding two different files to a user - **Fixed**

User seeding two different files to two different users - **Fixed**

User downloading a file size greater than 65028 bytes - **Not Fixed**

Having a command line interface - **Fixed**

CONCLUSION

There were many challenges throughout the course of this project. Our intention at the start was to implement everything on our own so that we could get a good understanding of how everything worked without relying on pre-built things. Unfortunately, doing this meant lots of time debugging since we were doing it like C, most things were done ourselves. Most of the time it felt we were doing a computer networks project rather than a distributed system project. Our group fully understood the proof of concept we designed and how we wanted to do it but when it came to implementing we had a lot of networking issues. Every group member was able to work on their individual parts and test them on a loopback host but as a final product, we wanted the application to actually be P2P meaning every computer is exactly one node. This led to port forwarding issues as some of our routers were difficult to configure properly. If we had more time we could have worked more on the UI and other features that were not fully implemented. Overall our group enjoyed doing this project and learned essential concepts of distributed systems and how crucial they are. Being users ourselves in the everyday world, we did not realize how many things are hidden from us and resolved automatically in the background without us knowing anything.