A BitTorrent-like peer-to-peer based file sharing prototype system

Hao Zhou, University of Mississippi

***Abstract*—In recent years, BitTorrent has emerged as a very popular and scalable peer-to-peer file distribution mechanism. It has been successful at distributing large files quickly and efficiently without overwhelming the capacity of the origin server.[5] This is a project for CSCI 561 (computer networks) at University of Mississippi. In this project, a BitTorrent-like peer-to-peer based file sharing prototype system is implemented. This prototype system has the ability of downloading and uploading one file at the same time automatically.**

Index Terms – BitTorrent, Peer-to-peer, file-sharing

I. Introduction

Nowadays, with the rapid development of the internet, more and more people tend to use the internet, and the traditional client-server model is facing lots of challenges.

Peer-to-peer networking has emerged as a very popular and scalable way which solves the challenges of client-server model. Peer-to-peer computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.[2] Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.[3]

BitTorrent is an internet peer-to-peer file sharing protocol that works in a sort of decentralized fashion. Its uniqueness lies in the fact that as you download portions of your files from the the person who originally shared the file, you are also getting portions from fellow downloaders to maximize data exchange. BitTorrent is one of the most commonly used protocols for transferring very large files because it doesn't overload web servers that provide downloads—since everybody is both sending and receiving, it's much more efficient than everybody downloading from a single server. It has been successful at distributing large files quickly and efficiently without overwhelming the capacity of the origin server.[4]

II. Motivation

This project is a programming assignment from CSCI 561 – Computer Networks at University of Mississippi. In the course, students are required to choose one topic from a list that related computer networks. The most interesting topic to me is BitTorrent for several reasons. Firstly, as a computer science student, I wish to learn the relative new technologies in the world. Peer-to-peer is one of them. Secondly, I have learned some parts of BitTorrent in the course, this will make the assignment much easier than others. Lastly and most importantly, BitTorrent was not a new thing to me since I was using it to download files several years ago even though I don't know how it exactly works. Taking this assignment as an opportunity, I would like to learn more about implementing details of a BitTorrent-like file sharing system.

III. Macro Structure

The BitTorrent-like file sharing prototype system has four peers in total and only peer 1 has the file to be downloaded.

a. Peer 1 (file)
b. Peer 2
c. Peer 3
d. Peer 4

IV. Micro Structure

Each peer has one main thread, and under the main thread, server thread is also running until the main thread gets killed. The main thread also creates client threads to broadcast downloading requests, receive response from other peers and send requests to download file chunks.

A. *Client thread*

1. broadcast file information to peers
2. receive responses from peers
3. process responses and send downloading requests

B. *Server thread*

1. receive requests from peers
2. send responses or file to peers

V. Implementing Details

A. PROTOCOL

Like HTTP, the prototype system also has its own protocol defined in order to process requests and responses in a better way.

a. WHERE *filename*\r\n\r\n

This is used by client thread to broadcast file information.

b. HERE *filename_chunk_number chunk_size*\r\n\r\n

This is used by server thread to generate responses for requested file. *filename_chunk_number* is a chunk name after server thread finds the requested file successfully and split the file into five chunks. *chunk_size* is size of the chunk.

c. NO\r\n\r\n

This is used by server thread when no requested file found under peer directory.

d. GET *filename start end*\r\n\r\n

This is used by client thread. f*ilename* is the file the peer wants to download, *start* is where the peer wants to start to download, and *end* is where the peer wants to end downloading.

B. LOG SYSTEM

The prototype system has a log system. Every useful response will be stored in a log file temporally. After the log file is being processed by client thread, the log file will only store requested chunk names, chunk sizes, starts, ends, and addresses to download.

After all responses are being processed and requests generated in the log file, the client thread will automatically pick up unfinished downloading(if end is not equal to chunk size) and send requests based on the protocol defined.

If all chunks are downloaded successfully, the log system will automatically delete the stored information and concatenate all chunks into one file.

Both server and client sides have the log system.

C. SERVER THREAD DETAILS

In order to better serve as a server in a peer, the server thread is multi-threaded and server thread has several functionalities.

a. Split file

This function is used to split the requested file into five chunks. Each chunk will be 500,000 bytes and the fifth chunk will be whatever left. And each chunk could have a name like ‘split_0’, ‘split_1’, ‘split_2’, ‘split_3’ and ‘split_4’. So later, for client thread, it becomes much easier to concatenate them into one file.

This function will happen when there is a connection and all generated chunks will be removed after the connection is shut down.

b. Clean chunks

This is used to clean logs and chunks mentioned in split file.

c. Find requested file and generate responses

This is a function used by server thread when there is a request to find which peer has the file requested.

If the server thread can find the file, then the server thread will read file log and grab chunks’ information, then generate response to peer. For example, ‘HERE split_1 500000\r\n\r\n’.

If not, the server thread will drop the request message and just send ‘NO\r\n\r\n’ back to peer.

d. Send file

This function is used to send requested file to peers if request message is ‘GET’.

The function will first check if start is equal to end in the request message.

If start is equal to end in the request message, then the server thread will drop the message and send nothing back to peer.

If not, the server thread will open the file requested and drop the first start bytes, then send the rest bytes to peer.

D. CLIENT THREAD DETAILS

Client side of peer is controlled by the main thread. When a peer wants to download the file. First, client thread will broadcast the file information to all other peers to find out which peer has the file it needs. Then the client thread will process the received responses from peers as long as it is not ‘NO\r\n\r\n’, then after doing randomly selection, the responses for chunks will be stored in log file. Finally, the client thread will use multi-threads to download different chunks at the same time. There are several functions listed.

a. Broadcast

Broadcast is one important function in client side. The function will send ‘WHERE filename\r\n\r\n’ to all other peers in order to find out who has the file. Then the function will receive response messages from other peers, either ‘NO\r\n\r\n’ or ‘HERE file_chunk chunk_size\r\n\r\n’. The message will be processed and stored in log file.

b. Process log file

The log file now, stores all response messages from peers. Then the client thread will read all messages in the log file and kick out all ‘NO\r\n\r\n’ with only ‘HERE file_chunk chunk_size\r\n\r\n’ remaining.

c. Random selection

This algorithm will be responsible for selecting (chunk, peer’s address) pairs so that client thread could store the requests in form of chunk’s name, start, end, size, IP address in the log file.

Here is the psedocode:

```
Random-Selection:
  requestList =[]
    for chunk in set(responded_chunk_list):
      while TRUE:
         pick = random.choice(responded_chunk_list)
         if chunk in pick :
           requestList.append(pick)
           break
         end if
       end while
    end for
```

Algorithm explanation:

1. requestList will store final requests.

2. responded_chunk_list is a list of responses from peers, some of them have the same chunks so that the same chunk information will appear in the list. That is why it needs loop through a set of the list.

3. Random.choice is a function from library.

4. If a chunk is found in pick, it means the chunk has been selected then break the while loop and continue the for loop.

d. Retrieve requests from log file

With the random selection algorithm, the requests now are stored in log file. This function will pick up all requests and send requests based on (chunk, peer's address) pairs.

The function will send requests to a peer if and only if the chunk is not finished downloading because of either the client thread is being shut down or server side of the requested peer is not available at that time. In either situation, the client thread will keep a record of downloading status of chunks. So later on, the peer has the ability to continue downloading where the chunk ended downloading from the same requested peer or from another peer without re-downloading it from the very beginning.

e. Break-point re-connection

This is one key feature of the prototype system. The way of doing that is based on making use of the log system. Recall that any request that stores in the log file will be in form of chunk's name, start, end, size, peer's address. For example, *'split_1 0 500000 500000 127.0.0.1:8001'*. This is a request where the downloading is not started yet because of start is 0. When the downloading starts, the value of start will change until it hits size.

However, the process of download could be differ in different situations.

1. Server side down

In the situation, there is no way to make a connection with the server thread. In the example above, our client thread will not be able to connect to 127.0.0.1 at 8001. At the time, the log will change to *'split_1 **>=0** 500000 500000 127.0.0.1:8001'*. The client thread will detect the error then decide to re-broadcast only the chunk information to all other peers.

For example, 'WHERE split_1\r\n\r\n'. Then the client thread will follow the order of a,b,c,.. until it finishes downloading.

2. Client side down

In the situation, this means the peer left without finishing downloading. Now the log will change to *'split_1 **>=0** 500000 500000 127.0.0.1:8001'* and the log will be same until the peer is online again. If the peer is back, it will continue to download whatever left from 127.0.0.1 at 8001 unless 8001 is shut down, then go back to situation 1 until finished.

f. Update log

When the client thread is in this step, this would mean that the peer wants to leave no matter if all chunks have been downloaded. Then the function will clean up the log file.

First, it will check if there are any chunk that is already finished downloading based on the value of start and the value of size. If they are equal, then the function will delete that line. If not, then function will not touch that line.

This function will ensure that when peer joins the network again, they don't need to worry about chunks that have been downloaded.

VI. TESTING AND RESULTS

A. Requirements

Python 3.6+

Some required libraries

Ubuntu 18.04

B. How to run

1. This project has four peers. Each has one .py file as its application.
2. Go to peer 1's directory to open terminal and type: python p2p.py –ip 127.0.0.1 –port 8001. Do the same for peer 2-4.

C. Testing and Results

Main thread of peer 1 looks like:

```
hzhou3@Joe:~/Desktop/csci561/peer1$ python p2p.py –ip 127.0.0.1 --port 8001

Press 1 to know who has the file you need
Press -1 to exit
```

Now, peer 2 wants to download the file. Run the command and type 1 to download.

```
8001 split_2 0 500000 0
Thread 0 started to download split_2....
8001 split_1 0 500000 0
Thread 1 started to download split_1....
8001 split_3 0 500000 0
Getting split_1 from 127.0.0.1 at 8001
Getting split_2 from 127.0.0.1 at 8001
Thread 2 started to download split_3....
8001 split_0 0 500000 0
Thread 3 started to download split_0....
8001 split_4 0 350755 0
Thread 4 started to download split_4....
Total  500000  Bytes Got
Total  500000  Bytes Got
Getting split_3 from 127.0.0.1 at 8001
Getting split_0 from 127.0.0.1 at 8001
Getting split_4 from 127.0.0.1 at 8001
Total  350755  Bytes Got
Total  500000  Bytes Got
Total  500000  Bytes Got
```

Now, peer 2 has the file, too! All chunks downloaded from 8001. Note that the order of downloading is not fixed because of multi-threads.

Next, peer 3 wants to download, too!

```
8001 split_1 0 500000 0
Getting split_1 from 127.0.0.1 at 8001
Thread 0 started to download split_1....
8002 split_2 0 500000 0
Getting split_2 from 127.0.0.1 at 8002
Thread 1 started to download split_2....
8002 split_3 0 500000 0
Getting split_3 from 127.0.0.1 at 8002
Thread 2 started to download split_3....
8001 split_0 0 500000 0
Thread 3 started to download split_0....
8001 split_4 0 350755 0
Thread 4 started to download split_4....
Getting split_4 from 127.0.0.1 at 8001
Total  500000  Bytes Got
Total  500000  Bytes Got
Total  350755  Bytes Got
Total  500000  Bytes Got
Getting split_0 from 127.0.0.1 at 8001
Total  500000  Bytes Got
```

Note that chunk 2 and chunk 3 are downloaded from 8002 and other from 8001.

Now let us see how break-point re-connection works!
Suppose that Peer 2 left.
This is the log file of peer 4:

```
split_3 0 500000 500000 127.0.0.1:8002
split_1 0 500000 500000 127.0.0.1:8002
split_4 0 350755 0 127.0.0.1:8002
split_0 0 500000 500000 127.0.0.1:8003
split_2 0 500000 500000 127.0.0.1:8001
```

Now reconnect to 8002:

```
Getting split_4 from 127.0.0.1 at 8002
lost connection: 127.0.0.1 at 8002
Total  0  Bytes Got
```

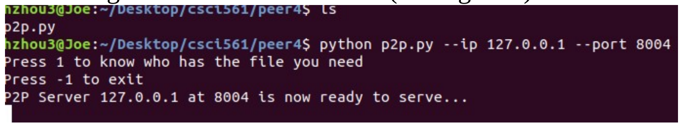
Connection cannot be made because peer 2 left!
Then the client will do as following:

```
127.0.0.1 at 8002 is not open, so the file split_4 cannot be downloaded from there.
Broadcasting again

8003 split_4 0 350755 0
Getting split_4 from 127.0.0.1 at 8003
```

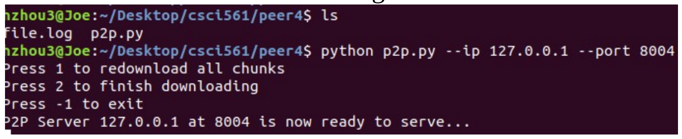
Finally, the chunk 4 is downloaded from peer 3 at 8003!

Main thread has 2 options:
1. Nothing downloaded from others(No log files)



```
hzhou3@Joe:~/Desktop/csci561/peer4$ ls
p2p.py
hzhou3@Joe:~/Desktop/csci561/peer4$ python p2p.py --ip 127.0.0.1 --port 8004
Press 1 to know who has the file you need
Press -1 to exit
P2P Server 127.0.0.1 at 8004 is now ready to serve...
```

2. There is unfinished downloading



```
hzhou3@Joe:~/Desktop/csci561/peer4$ ls
file.log  p2p.py
hzhou3@Joe:~/Desktop/csci561/peer4$ python p2p.py --ip 127.0.0.1 --port 8004
Press 1 to redownload all chunks
Press 2 to finish downloading
Press -1 to exit
P2P Server 127.0.0.1 at 8004 is now ready to serve...
```

VII. Future Work

Right now, the prototype system can only handle one file at a time, the future work will be extending the system to handle multiple files at the same time. And adding a real tracker to the system will be another task to do.

VIII. Conclusion

In the document, all details of a BitTorrent-like file sharing system are described. In the system, I mainly focus on how to achieve upload and download one file at the same time. Also I introduced a log system to solve break-point re-connection problem. This method worked well to solve the problem.

References

[1] Galuba, Wojciech; Girdzijauskas, Sarunas (2009), LIU, LING; ÖZSU, M. TAMER (eds.), *"Peer-to-Peer System"*, *Encyclopedia of Database Systems*, Springer US, pp. 2081–2082, doi:10.1007/978-0-387-39940-9_1230, ISBN, retrieved 2019-11-30.

[2] Rüdiger Schollmeier, “*A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications”*, Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE (2002).

[3] Bandara, H. M. N. D; A. P. Jayasumana (2012). "*Collaborative Applications over Peer-to-Peer Systems – Challenges and Solutions*".*Peer-to-Peer Networking and Applications*.**6**(3): 257–276.

[4] Yatritrivedi. “*BitTorrent for beginners: How To Get Started Downloading Torrents*.” April 4, 2018. https://www.howtogeek.com/howto/31846/bittorrent-for-beginners-how-get-started-downloading-torrents/

[5] Bharambe, A.R. Herley, Cormac. “*Analyzing and Improving BitTorrent Performance*”. Feb 2005. Microsoft Research.