Spring 2022 CSE 422
Project 3: Bittorrent Lite

April 2nd, 2022

---

1 A Word of Warning

Make sure you read through the submission instructions thoroughly before you submit your project. Failure to comply with the submission instructions will result in a zero for this project.

2 Project Due Date

We will be handling the due dates a little bit differently for this project. Instead of offering extra credit for adding additional functionality, we will offer extra credit points for submitting the project early. If you submit your project by **April 21st at 11:59 pm** we will give you +10 points to your score. If you submit your project by **April 26th at 11:59 pm** we'll give you +5 points, and the final due date is **April 29th at 11:59pm**. There will be no extensions given past this final due date, as it is the last day of the semester. Extensive submission instructions are at the end of this document.

This project is worth 25% of your final grade, so it will be very important to start early and ask questions if you are confused. Additionally, the extra credit is worth a lot of points towards your overall score considering the project itself is worth such a large percentage of the course grade.

3 Project Overview

In this project you will built Bittorent Lite, and simplified clone of the Bittorent protocol. Bittorent Lite will run on top of TCP, so you will not need to implement reliable data transfer yourself (you're welcome!). Bittorent Lite will provide a way for processes to download a file that is stored amongst many hosts. This is called 'peer-to-peer file sharing,' and it greatly differs from most file sharing applications you might be familiar with. In a more typical application, a central server will have the entirety of a file you wish to download, however, in Bittorent Lite,

pieces, or chunks, of our desired file will be spread amongst peers in the 'peer-to-peer' network. You will have to request various chunks of the file from the peers who currently have the chunk, and reassemble the file locally once you have all of the chunks. While this is happening, your own process must be able to respond to requests from other peers asking for chunks that you currently own.

A good tutorial on TCP sockets can be found here: https://beej.us/guide/bgnet/html/index.html. We will release some starter code in the next few days, you can use the provided starter code if you like, but you are not required to use it. If you do use our starter code, I recommend that you spend some time learning how TCP sockets work.

You can use the same Mininet VM we used for Projects 1 and 2. For this project you should test on Mininet frequently to make sure that your code works, as how different platforms handle things can be quite different.

4 Bittorent Lite Specifications

Bittorent Lite will have two major components, the peers, and the tracker server. The tracker server knows the identities of all peers in the network, and also provides torrent files to peers who want to download a file. The torrent files for Bittorent Lite will contain a list of the peers in the network as well as a list of the chunks of the file, including the hash for each chunk. Each chunk of a file has a hash value to ensure that any peer has not modified that chunk of a file before sending. This is one way that Bittorent attempts to reduce the threat of malware in peer-to-peer file sharing. If any peer tries to inject malware into a file, it will be caught by a changed hash value, and all other peers will know to discard that chunk of the file. If the file becomes corrupted during transit, then TCP's error detection will handle retransmissions, and therefore the hash value used in Bittorent Lite is only used to detect malicious peers in the network, and not packet corruption that might occur in transit.

Peers in the network will first connect with the tracker server and receive the torrent file. Once a peer has the torrent file, they will ask each peer in the network which chunks of the file they own, and after this they will proceed to request different chunks of their desired file from the other peers in the network. Once they have all the chunks of the file they will finish reassembling the file. Because peers in Bittorent Lite are not selfish, our peers will remain running in case another peer still needs a chunk that is owned by a particular peer that has already finished downloading the entire file.

There are provided PacketHeader.h and crc32.h files included with this project. The PacketHeader.h file contains a struct that defines the various packet headers in Bittorent Lite. The crc32.h file will be used to create 'hash codes' for the various chunks of the file. Using crc is not typical for Bittorent, which would usually use something like a SHA-1 hash. However, for simplicity's sake, and because everyone should be familiar with the crc32 computation from Project 2, we will use the crc32 as our chunk hashes.

4.1 Chunk Size

In Bittorent Lite, files are split into chunks of 512KB, or 512,000 bytes. While this is larger than a single Ethernet frame, we will let the network protocol stack handle packet fragmentation and reassembly, so you can send an entire chunk in one TCP message.

4.2 Packet Structure

All packets sent in the network will have the same structure. They will contain a packet header as specified in PacketHeader.h, followed by the data relevant to the type of request being sent. This means that a single packet will always contain 8 bytes of packet header, and if there is anything in the payload, that data will directly follow the packet header.

5 Tracker Server

The tracker server in Bittorent Lite will create torrent files, and distribute the torrent file to any peer who wants to download a file. In general, each time the tracker server is run it will create only one torrent file, as specified in the tracker invocation. This means that any individual run of Bittorent Lite only has to worry about one file that can be downloaded.

Once the tracker server starts, it will create the torrent file, then wait for peers to connect, and then send the torrent file to each peer who connects. The tracker server has to be able to handle multiple requests from peers simultaneously, which means that you will need to build the tracker server with some sort of concurrency, most likely through the use of threads. While the tracker server is creating the torrent file it does not need to handle requests, it will only start accepting requests once the torrent file has been completed.

To create the torrent file, the tracker server will first read from the peers-list.txt file, the path to this file is sent in by user input. The peers-list.txt file contains the IP addresses of all of the peers in the network, each on its own line. A sample peers-list.txt file can be found on D2L. The peers-list.txt file will have to be manually created, then the path to the file will be specified on invocation of the tracker server. The filepath to the file to be downloaded is also specified by user input. The tracker server will chunk the file, and compute the crc32 code for each chunk of the file. It will then create the torrent file that includes all of the peers in the network, the chunk IDs and associated crc32 code for each chunk. A torrent file will be a .txt file with the following format, and an example can be found on D2L. The torrent file starts with a single integer that specifies the number of peers, n, in the network. Then, on each of the n lines afterwards is the IP address of a peer in the network, in dotted decimal format. Then there is another integer on its own line that specifies the number of chunks, m. The following m lines contain the chunk ID and crc32 hash code for the chunk, separated by a space.

```
2
10.0.0.1
10.0.0.2
5
0 2623628000
```

1 3362059802
2 420599673
3 801491073
4 1228312268

Note that the chunk IDs start at 0, and increment by 1 after each 512KB chunk.

6 Running the Tracker Server

The tracker server should be invoked as follows:

./tracker <peers-list> <input-file> <torrent-file> <log>

Peers-list is the path to the peers-list.txt file that specifies all the hosts in the network, input-file is the path to the file that will be shared amongst the network and must be chunked by the tracker, torrent-file is a path to where the tracker server will write the torrent file it creates, and log is the path to the log file the tracker will create.

The input file that will be chunked and downloaded by the peers can be any binary text, image, or video.

The tracker server will always listen for peer connections on the same port, so it does not need to be specified by user input. In Bittorent, the tracker server traditionally listens on port 6969, so that's what we will use as well. By the way, that is completely legitimate, take a look at what Wireshark has to say on the matter: https://wiki.wireshark.org/BitTorrent#protocol-dependencies.

When responding to a peer that requests the torrent file, the tracker will send back a packet with a header with the type value = 1. This specifies that they are sending the torrent file back to whomever requested it. The tracker will also specify the length of the torrent.txt file in the packet header's length field. You can assume that the torrent file is always less than 512KB.

You can assume that all inputs to the tracker are valid.

7 Bittorent Lite Peers

The peers in Bittorent Lite are processes that wish to download a given file from the peer-to-peer network. When a peer starts up, it will request the torrent.txt file from the tracker server. The IP address of the tracker server will be specified by user input, and the port number of the tracker server will always be 6969. Once the peer has the torrent file it will parse it to determine the other peers in the network, as well as the chunk IDs and chunk hashes of the file it wishes to download. Once the peer has parsed the torrent file it must request various chunks from other peers, as well as respond to requests for chunks from other peers in the network. This must be able to be done simultaneously, which will require some form of concurrency, again, most likely through the use of threads.

7.1 Requests and Responses to Other Peers

There are two types of requests that can be made to peers in the network. The first is a request to a peer to determine which chunks a specific peer owns. Once a peer sends requests to all other peers in the network to determine who owns what chunks, it can request the actual chunks from whomever owns them. The particular chunks that a peer owns are contained in the owned-chunks.txt file. This text file must be created manually for each peer and is specified by user input when the peer is started. The owned-chunks.txt file contains the chunk IDs of all chunks that are owned by a peer on start-up, each chunk ID on its own line in the file. An example owned-chunks.txt file can be found on D2L. The owned-chunks.txt file should not ever be modified by the peer. Whenever a peer responds to a request for what chunks it owns, it will always send the data specified in the owned-chunks.txt file, even if they have gained more chunks since start-up. The packet structure of both the request for owned chunks as well as response for owned chunks can be seen in a following section of this specification.

The second type of request that can be made to other peers in the network is a get chunk request. This request asks a specific peer to send a chunk that it owns. Peers will request chunks from other peers based on a rarest-first algorithm. For example, if only one peer owns chunk 2 of a file, and two peers own chunk 1 of a file, and 3 peers own chunk 0 of the file, you will first request chunk 2 from the owner of that chunk, then chunk 1 from one of the two owners of the chunk, and then finally chunk 0 from one of the 3 owners of that chunk. Ties can be broken any way you prefer. When receiving a chunk from another peer, you must check the crc32 hash code against the torrent file. If they do not match, you should discard the packet and send the request again.

Requests to other peers be done sequentially, meaning that you do not need to send out all of your requests at once in individual threads. However, you must be able to send out requests to other peers while also accepting requests from other peers simultaneously. For example, if you are going to use threads for concurrency, all of your requests to other peers can occur in one thread, while other threads will be accepting requests from other peers at the same time.

When accepting requests from other peers, each request must be able to be handled simultaneously. For example, if you are using threads, you will accept a request, and spawn a new thread to handle the request, then immediately start waiting for another request to come in while the previous request is being served. Each of these threads to handle requests from other peers, as well as the thread to send out requests must be able to run concurrently.

7.2 Structure of Different Packet Types

There are six different types of packets that can be sent in Bittorent Lite:

- Packet type 0 is a request from a peer to the tracker server for the torrent file. This packet will be only a packet header with type 0, and no payload. Whenever there is no payload for a packet, you can make the length field 0.
- The second packet type is type 1. A type 1 packet is a response from the tracker server to the peer who requested the torrent file. The length field of a type 1 packet is set to the

file size of the torrent.txt file. The data payload of this type 1 packet is the file data for the torrent.txt file.

- A type 2 packet is a request to determine what chunks another peer has. This packet only contains a packet header with the type value set to 2, and a length of 0.
- A type 3 packet is a response to a type 2 packet. A type 3 packet contains a payload that is a list of chunk IDs owned by a peer. Each chunk ID is an unsigned integer, and therefore four bytes long. The length field of a type 3 packet is equal to the number of chunks owned by a peer multiplied by 4. There should be no padding or spacing put between the chunk IDs in the data payload.
- A type 4 packet is a request from one peer to another for a specific chunk. The data payload of a type 4 packet is the crc32 hash code associated with the chunk that is being requested. This means that the length field of a type 4 packet is 4, because the crc32 has code should be an unsigned integer, which is always 4 bytes long.
- A type 5 packet is a response to a get chunk request. In a type 5 packet the data payload is the file data corresponding to the chunk requested by another peer.

8 Running Bittorent Lite Peers

The peer should be invoked as follows:

./peer <my-ip> <tracker-ip> <input-file> <owned-chunks> <output-file> <log>

My-ip is the ip address of the peer being run. Each peer needs to know its own IP address for Bittorent Lite. Tracker-ip is the IP address of the tracker server. Remember that the tracker is always listening on port 6969. Input-file is the path to the file that we want to download. Owned-chunks is the path to the owned-chunks.txt file. Output-file is the filename of where the downloaded file should be written. Log is the path to the log file.

A point of note, is that each peer will have access to the full master file that it is trying to download. Which means that each peer has access to the full file. However, for our purposes each peer is only allowed to read the chunks that it owns from the file.

Each TCP connection should be used for exactly one request and response. Therefore, if I want to request two chunks from the same peer, I must do so over two different TCP connections. As specified by Wireshark, our Bittorent Lite peers will always be listening to requests on port 6881. This means that each peer will bind and listen for incoming requests on port 6881, while also making requests to other peers from another port. This means that whenever a peer wants to make a request to another peer, it can assume that it can reach the peer at port 6881.

When terminating a TCP connection, whichever peer initiated the request will perform an orderly shutdown of the TCP connection after it successfully receives the response to its request. This means that the peer who responded to an initial request will end its connection (and its thread) once it knows that the other peer has shut down properly.

You can assume that all inputs to the peer are valid. Additionally, you can assume that every chunk of a file is owned by at least one of the peers, and the entire file can be recreated properly. You can assume that the tracker server and each peer all have the same input file they want to download, and you do not have to support multiple files at the same time. You can assume that every peer specified in the torrent as well as the tracker server will be running. That means that if you make a request to connect that fails, you should continue to try and reconnect until you succeed.

9 Logging

Each peer, as well as the tracker server should create a log of its activity. After sending or receiving each packet, it should append the following line to the log.

For type 0, 1, 2, and 3 packets:

<ip-address> <packet-type> <packet-length>

Where ip-address is the IP address of the peer or tracker that you are currently connected to.

For type 4, and 5 packets:

<ip-address> <packet-type> <packet-length> <crc32-hash>

Where crc32-hash is the hash code of the requested chunk.

10 Hints

For file reading and writing it is recommended that you use ifstream and ofstream. You can handle concurrency however you prefer, but the std::thread library is a good way to achieve this. You have to manually create a few files, make sure that you make the files valid, for example, that every chunk is owned by at least one peer. It might be helpful to run the tracker server first to see the torrent.txt file to know exactly what chunks exist in a file before you make the owned-chunks.txt files. You will need to perform some synchronization, at least for logging on peers, and potentially for other things depending on how you implement Bittorent Lite. I recommend the mutex class to help with synchronization, however, there are many options that can handle synchronization.

11 Project Writeup

On D2L, in addition to this project specification there is a project writeup file. This is a barebones template of the project writeup that you will have to submit.

You must put significant time and effort into this writeup, please check the syllabus for how

your writeup grade can impact your final project grade.

In general we are expecting paragraph answers (at least 3-4 sentences) for each question. Failure to go into enough detail will result in a low score. For questions that ask you to describe your code and the choices you made, we are expecting that you will write about the different decisions you made, why you made them, and what they accomplish. You do not need to walk us line-by-line through your code, but anyone reading your project writeup should be able to understand your code and how it works.

The Project 2 writeup should have a filename as follows: <netid>_project3_writeup.pdf

12 Submission Instructions

For project 3 you will be uploading your files directly to handin, one by one. You must submit your C/C++ source code files, a makefile, your project writeup, the PacketHeader.h and crc32.h files, and your peer and tracker binary executables that are generated after running the 'make' command. Your makefile must support both 'make' and 'make clean' commands and must generate both the peer and tracker executable files with a single 'make' command. If your code does not compile, or does not compile with a 'make' command, then it will not be graded.