Lab 5: BitTorrent Client Implementation

Due: Nov. 30th at 11:59 PM
Milestone: Nov. 19th during Lab

Overview

In this lab, you and your lab parterner will develop a basic BitTorrent client that can, at minimal, exchange a file between multiple peers, and at best, function as a full feature BitTorrent Client. The programming in this assignment will be in C requiring standard sockets, and thus you **will not** need to have root access. You should be able to develop your code on any CS lab machine or your personal machine, but all code will be tested on the CS lab.

Deliverable

Your submission should minimally include the following programs and files:

- `REDME`
- `Makefile`
- `bencode.c|h`
- `bt_lib.c|h`
- `bt_setup.c|h`
- `bt_client`
- `client_trace.[n].log`
- `sample_torrent.torrent`

Your `README` file should contain a short header containing your name, username, and the assignment title. The `README` should additionally contain a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. **Your `README` should also contain a list of all submitted files and code and the functionality implemented in different code files.** This is a large assignment, and this will greatly aid in grading.

As always, if there are any short answer questions in this lab write-up, you should provide well marked answers in the `README`, as well as indicate that you've completed any of the extra credit (so that I don't forget to grade it).

Milestones

The entire assignment will be submitted and graded, but your group will also hold a milestone meeting to receive feedback on your current implementation. A poor showing in your milestone meeting can adversely affect your grade. More info on milestone meetings will be discussed in lab.

Submission Instructions

To submit this lab you will use `handin43`. You can retrieve relevant files via `update43`.

1 BitTorrent Protocol

BitTorrent is a peer-to-peer file sharing protocol develop by Bram Cohen. It was developed in response to centralized file sharing services, like Napster, where centralized infrastructure was necessary to enable downloaders to connect with uploaders. Additionally, centralized services did not enable for multiple file uploaders to send data to multiple file downloaders.

BitTorrent is designed to address both of these shortcomings by enabling file downloaders and file hosters to interact directly in a peer-to-peer fashion; that is, BitTorrent participants interact using a peer-to-peer protocol where each participant acts as both a server and client. Perhaps BitTorrent must important feature is that it enables a large number of downloaders to collaborate in downloading a file from a much smaller number of file uploaders. BitTorrent also has built in mechanisms to ensure fairness which encourages participants to share previously downloaded data as uploaders, and if they don't, a downloader is punished by limiting download speeds.

In this lab, you will implement a core subset of the BitTorrent protocol by implementing a BitTorrent client. Below, I outline the relevant portion of the BitTorrent protocol, for more detailed descriptions, you should read Bram Cohen original specification here: `http://www.bittorrent.org/beps/bep_0003.html`. Unless otherwise stated in this document, you are expected to implement your client to meet the specification outlined by Cohen.

1.1 BitTorrent Basics

There are three distinct parts of the BitTorrent protocol:

- **Metainfo File**: The *metainfo file* (or a *torrent file*) is a description of the file[s] to be downloaded as well as information to contact the tracker server. A metainfo file is generated by the *file originator* and uploaded to a web server where file downloaders can retireve the metainfo file and use it to learn about the tracker server.

- **Tracker Server**: The tracker server monitors the number of clients downloading/uploading files. When a file downloader contacts the tracker server, a list of other clients ip and ports are returned. Tracker servers have become rather advanced, and some even actively participate in the BitTorrent client procedure to better monitor and ensure fairness. (*You are not required to implement the tracker server for BitTorrent in this lab.*)

- **Clients**: A BitTorrent client is the application run by downloaders/uploaders that exchanges *pieces* of a file. The division of the file into pieces is described in the metainfo file, and the protocol for exchanging file pieces is called the *client protocol* or *peer protocol*. Clients learn about other BitTorrent clients via the `tracker server`, as well as accepting new request from peer clients directly in a peer-to-peer fashion.

  There are three distinctions of BitTorrent clients: A client that is only uploading a file is described as a *seeder*; a client that is only downloading a file is described as a *leecher*; and a client that are connected to each other, either downloading and/or uploading, are described as *peers*. All seeders, leechers, and peers actively downlaoding/uploading are described as the *swarm*.

The BitTorrent workflow is as follows. A file originator who wants to start sharing a file generates a matainfo file which contains details about the tracker server, the size and partitioning of the file, and other relevant details. The metainfo file is uploaded to a generic web server where others who wish to download the file can retrieve the metainfo.

With the metainfo in hand, a BitTorrent client contacts the tracker server described in the metainfo file and retrieves a list of other clients participanting in the swarm. The client then contacts the other clients, establishes a connection, and starts the client protocol by requesting and downloading pieces of the file *in random order* until the entire file is downloaded. As the client collects pieces, it starts making these available to others in the swarm by announcing that it has the piece. In this way, together, all clients can aid each other in downloading the file faster and from dispersed sources.

In this lab, your client ***must*** be able to parse and understand metainfo files and participate in the client protocol. However, ***you are not required*** to implement features of the tracker nor have your client contact a tracker to retrieve a list of peer clients. Instead, your BitTorrent client will take a pre-selected list of peers as a command line argument.

1.2 Metainfo Files and bencoding

Metadata in BitTorrent is encoded in a special format call *bencoding*. This assignment is not about parsing bencoded data, so I have provided you with a library that can parse bencoded data (see `bencode.c|h`); however, you will still need to traverse the parse tree to retrieve relevant info. I did not write this library, and you will need to consult the header file for details. For more details on bencoding, you can read the entire bencoding specification in Cohen's BitTorrent specification.

Most relevant to this discussion is that bencoding allows for standard data types, like integers, strings, dictionaries and lists, and a metainfo file is just a bencoded dictionary with the following keys and values:

- `announce` : The URL of the torrent tracker
- `info` : A dictionary with the following keys
  - `name` : The suggested name for saving the file
  - `piece length` : The size, in bytes, of a piece for this file. This is always a power of 2.
  - `length` : The length of the file to be downloaded
  - `pieces` : A string of length in multiples of 20 bytes. Each 20 bytes subpart represents the SHA1 hash of the piece of the file at that index.

There are other formats for downlaoding multiple-files, but ***you are not required to handle multi-file torrents*** in this assignment. If you are interested, I refer you to the BitTorrent specification. Most importantly, you should think carefully about the `info` portion of the metainfo file. Particularly, the `pieces` value which not only describes the data expectation of the file (i.e., its hash), but also indicates the index of the pieces. The client protocol is all about exchanging pieces; specifically, the client protocol is concerned with the requesting, announcing, and downloading of pieces.

1.3 Tracker Protocol

Since you are not required to implement or communicate with a tracker, I refer you to the BitTorrent specification for details on communicating with the tracker servers.

1.4 Client Protocol

The client protocol describes the process by which peers contact each other, learn which pieces others have, and request, download and upload pieces per these requests. Peers establish a TCP connection between each other, and these connections are symmetric, used for both uploading and downloading of pieces between peer clients.

Connection State For each connection to a peer, a client maintains two bits of state. A connection can either be *choked* or *unchoked*; that is, a client is willing to upload to an unchoked connection but not to a choked connection. The process of choking a connection is what encourages fairness in BitTorrent as well as limits the amount of uploading a client is willing to do for a given downloader. The other bit of state is based on *interest*, which indicates the client is interested in some piece that the peer has. ***A transfer can move forward when the connection isn't choked and one side is interested; initially, connections are set to chocked and not interested.***

Connection Establishment and Handshakes Before any data transfer, two peers must first establish a BitTorrent connection via a *handshake*. The handshake protocol is as follows and is symmetric, that is, both sides send the same information.

1. 20 bytes: The first byte is the ascii character 19 (decimal), i.e., the number 19 encoded in an unsigned byte, and the remaining bytes is the string value "BitTorrent Protocol".

2. 8 bytes: Reserved bytes all set to zero that may be used in extensions of the BitTorrent protocol.

3. 20 byte: `SHA1` has of the bencoded `info` value in the metafile, also known as the `info_hash`. Both sides should send the same value, if not, then they are not interested in participating in the same swarm.

4. 20 byte: The peer ID used to identify this client. ***We will use shorter peer id's but you should still place them within a 20 byte value***. If the peer ID doesn't match the expected ID, the connection is serve red.

Messages Once the handshake is complete, further communication is conducted via an exchanges of *messges*. BitTorrent messages can be described in the C structures presented in Figure 1, which I have provided for you in the skeleton code. Below, using Figure 1 as a guide, I describe the components and usage of the key messages.

Starting with the `bt_msg_t` type, a message starts with a `length` which indicates the length of the mesage, and following, is a one byte field, the `bt_type`, which describes the type of the message. There are different types of messages, described in the `payload union`. Below, we describe each of the types and there type identifiers:

- 0 - `choke`: A message indicating to peer that they are now choked. This message has 1 as its length value.

- 1 - `unchoke`: A message indicating to peer that they are now unchoked. This message has 1 as its length value.

```
typedef struct {
  char * bitfield; //bitfield where each bit represents a piece that
                   //the peer has or doesn't have
  size_t size;//size of the bitfiled
} bt_bitfield_t;

typedef struct{
  int index; //which piece index
  int begin; //offset within piece
  int length; //amount wanted, within a power of two
} bt_request_t;

typedef struct{
  int index; //which piece index
  int begin; //offset within piece
  char piece[0]; //pointer to start of the data for a piece
} bt_piece_t;


typedef struct bt_msg{
  int length; //prefix length, size of remaining message
              //0 length message is a keep-alive message
  unsigned char bt_type;//type of bt_mesage

  //payload can be any of these
  union {
    bt_bitfield_t bitfiled;//send a bitfield
    int have; //index of piece just completed
    bt_piece_t piece; //a piece message
    bt_request_t request; //requxest messge
    bt_request_t cancel; //cancel message, same type as request
    char data[0];//pointer to start of payload, just for convenience
  }payload;

} bt_msg_t;
```

Figure 1: The basic data structures for BitTorrent's message protocol.

- 2 - `interested`: A message indicating to peer that they are interested in some piece that at the peer. This message has 1 as its length value.
- 3 - `not interested`: A message indicating to peer that they are not interested in *any* pieces at the peer. This message has 1 as its length value, the length of `bt_type`.
- 4 - `have`: A message sent once to all peers once the client has downloaded a piece completed and verified it using the hash in the metainfo file. The value is just an integer indicating the piece index.
- 5 - `bitfield`: A message *only* sent after the handshake is complete to tell peers which pieces the client has available. The first byte in the bitfield represents pieces indexed 0-7, the second byte reperesents pieces index 8-15, and so forth. Extra bits are set to 0.
- 6 - `request`: A message sent to a peer indicating that they would like to download a given piece. A request message contains an `index`, `begin`, and `length` field (see structure above). The `index` indicates which piece is being requested. The `begin` field is the byte offset within the piece, and the `length` is the amount of data requested, in bytes. Generally, the `length` is a power of 2 and must

not exceed 2^{17}. Most implementation use 2^{15} as the length. The subset of the piece request is referred to as the *piece block*.

- 7 - `piece`: A message that indicates a data transfer of a piece (or a block of a piece). It has two fields: `index`, indicating which piece this associated with; and, `begin`, indicating the byte offset within the piece. The finale value, `piece` is a pointer pointing to the start of the raw file data.
- 8 - `cancel`: A cancel messsage is used to cancel a request for a piece (or block of a piece). A cancel message has the same data layout as a request, and it is generally used in the "End Game" where the file download is nearly complete. At the end of he download, the final blocks and pieces have a tendency to take a long time, so a client sends a request for the block or piece to every peer, and once it is downloaded, it must cancel the request using a cancel message.

Additionally, clients check the live-ness of their open connections by sending keep-alive messages. These messages have 0 set as the message length and are ignored by receivers with respect to taking actions. Keep-alive messages are sent once every two minutes, but if you are already exchanging pieces with a client, you can consider the client alive. ***You must implement a strategy for handling client failures***.

2 Provided Code

I have provided a good portion of skeleton code for you to work from.

- `bencode.[c|h]` : This is C library written by Mike Frysinger and edited by myself. It will parse a bencoded string into a tree like structure which you can then traverse to learn relevant info. See `load_be_node()` function for more details.
- `bt_setup.[c|h]` : This is a C library written by me that provides functionality for parsing command line arguments and peer strings.
- `bt_lib.[c|h]` : This is the core BitTorrent client library and header file. I have outlined a number of functions that you will need to write to complete this lab, with descriptions of their intended actions. Additionally, this file contains relevant structures for messages and arguments to your BitTorrent client.
- `bt_client.c` : This is the main portion of the client.
- `sample_torrent.torrent` : A sample torrent file to use in your development.
- `download_file.file` : The file that will be downloaded and uploaded by your client.

3 Non-BitTorrent Functionality

In addition to the core BitTorrent functionality, there are other standard software engineering challenges that you will encounter. Below, I outline some of the challenges and provide some advice for addressing them.

Program Command Line Arguments: I have built in basic command line parsing procedures into your client. Below is the usage:

```
bt-client [OPTIONS] file.torrent
  -h                     Print this help screen
  -b ip                  Bind to this ip for incoming connections, ports
                         are selected automatically
  -s save_file           Save the torrent in directory save_dir (dflt: .)
  -l log_file            Save logs to log_filw (dflt: bt-client.log)
  -p ip:port             Instead of contacing the tracker for a peer list,
                         use this peer instead, ip:port (ip or hostname)
                         (include multiple -p for more than 1 peer)
  -I id                  Set the node identifier to id, a hex string
                         (dflt: computed based on ip and port)
  -v                     verbose, print additional verbose info
```

Program Output: You should manage your program output such that information printed to the screen is useful but not overwhelming. Recall that your client will have a verbose flag, and you should reserve overly detailed output for verbose settings; however, there is a standard set of output you must provide. ***Your client must, at minimum, output the download status of the torrent, the number of connected peers, the amount of data uploaded, and the amount of data downloaded.*** Below is some sample output, note that you don't have to delete previous output.

```
File: foobar.mp3 Progress: 12.8% Peers: 5 Downloaded: 12 KB Uploaded: 15 KB
File: foobar.mp3 Progress: 13.1% Peers: 5 Downloaded: 14 KB Uploaded: 15 KB
```

Additionally, you should not print continually to the terminal. This will greatly slow down your program. Instead, you should print at reasonable intervals, for example, once a piece completes.

Multiple Socket Connections: Your client must be able to handle multiple connections seamlessly. That is, a client will have open sockets, and it needs to be able to read and send on all of them. Note that some operations are blocking, such as reading from a socket when there is no data to be read. Refer to the manuals for more information about how to use non-blocking procedures with sockets, but **I strongly recommend that you use the `select()` and `poll()` which both check the status of a file descriptor (like a socket) to see if it is ready for reading/writing/etc**.

File I/O: The core feature of BitTorrent is the ability to exchange sub-parts of the file in any order. This means you need to be careful about managing your file input and output to ensure that you are writing to the right part of the file based on the piece currently being downloaded. I refer you to Lab 4 for detailed information about file seeking and file streams.

Hashing: BitTorrent makes use of SHA1 hashes to check data integrity. The hash, or digest, is a 20 byte string, and is easily computed using the OpenSSL library implementation of SHA1. In Figure 2, I provide some sample code for computing the SHA1 digest of data read from a file.

State Information: Another key feature of BitTorrent is the ability to restart downloads in progress. This feature is enabled by maintaining state information about which pieces are currently downloaded and verified. The files bitfield is the state information that enables restarting features. It's important to keep this information fresh and also cache it out (to a file).

```
char data[BUF_SIZE]; //some size
char digest[SHA_DIGEST_LENGTH];//20 bytes
int fd = open(some_file); //some file
int bytes_read, i;

bytes_read = read(fd,data,BUF_SIZE);//read data
SHA1(data, bytes_read, digest); //compute hash

//print the hash
for(i=0;i<SHA_DIGEST_LENGTH;i++){
  printf("%02x",digest[i]);
}
printf("\n");
```

Figure 2: Sample code for computing the hash of data read from a file.

Logging Information: ***Your development must provide reasonable logging output.*** Events that must be logged are peer handshakes, peer messages, and file download status. You may also log other events, but describe these logs in your README. All log messages should contain: a *timestamp* in seconds/miliseconds since the start of the client; an *identifier* (in all caps) indicating the type of the log message, e.g., a peer handshake; and, *parameters* indicating other aspects of the logged action, e.g., which peer the handshake occurred with. For example:

```
[102.12] HANDSHAKE INIT peer:192.168.1.2 port:22 id:0xdeadbeaf
[103.34] HANDSHAKE SUCCESS peer:192.168.1.2 port:22 id:0xdeadbeaf
[105.23] MESSAGE REQUEST FROM peer_id:0xdeadbeaf piece:10 offest:0 length:256
[107.25] MESSAGE PIECE TO peer_id: 0xdeadbeaf piece: 10 offest: 0 length:256
( . . . )
```

4 Development Advice

This is a large and complex assignment, and you will need to carefully plan your development with your lab partner. Below, I outline some development hints that should help with your planning and testing.

- **Mini-Goals**: Although there are many moving parts to this assignment, there are a lot of independent portions. Think small first, and then think big. Get one thing working, and then move on to the next. Here are some mini-goals you should look towards: (1) Parsing of bencoded files; (2) Handshake protocol; (3) Managing multiple clients; (4) Reading/Writing piece blocsk from files; and etc.
- **Skeleton Code is Not Cannon**: There is no requirement that you must use every part of the skeleton, use the same data structures, nor use the function outlines. This development is yours, so make it yours. Note that in many situations you will find it advantageous to define your own functions, alter or adapt provide function outlines, or complete scrap portions of the provided code base. There will be other times where the skeleton code can be easily adapted or changed, and I do encourage you to maintain as much as you can because it makes it easier for me to help debug your clients and grade the lab.
- **Divide and Conquer**: Although I believe that there is much to gain from pair programming, but if you and your partner spend all your time programming together you will not complete this assignment. Instead, once you divide the lab into distinct goals, you should focus on accomplishing these tasks individual and then later work together to merge developments. This mimics real world development.

- **Consider Error Conditions**: When testing your program, you should be considering error conditions from the get go. Ask yourself: What can go wrong? For example, how would your program respond if a client crashed? What happens if the reading/writing of a file is out of sync and pieces or corrupted? Can you recover or must the client quit?
- **Debug Early and Often**: In addition to using standard debugging tools, like `gdb` and `valgrind`, you should also build in your own debugging facilities via the log file and the verbose output flag.

5 Grading and Extra Credit

Grading Breakdown Below is general grading breakdown that will be used for this assignment. Note that other issues will be considered, such as programming style and logic errors.

- (B/B+) **1-Seeder and 1-Leecher**: Build a client that can implement the standard client protocol such that a single seeder can transfer a file to single leacher.
- (B+/A-) **1-Seeder and 1-Leecher with Restarts**: Build a client that can implement the standard client protocol such that a single seed can transfer a file to a single leecher, and the leecher can restart from a partial download.
- (A-) **1-seeder and N-Leechers**: Build a client that can support single seeder and multiple leechers, but the leechers are not sharing with each other.
- (A-/A) **1-seeder and N-Leechers with Restarts**: Above, but clients can fail and restart with partial torrent downloads.
- (A) **1-seeder and N-peers**: Build a client that can support a single seeder and multiple peers who also exchange with each other.
- (A/A+) **N-Peers with Restarts**: To receive a full credit your client should be able to maintain connection to the swarm as other clients com and go, plus be able to restart a torrent that may have been incomplete.

Extra Credit The following functionality is for extra credit. I strongly suggest that you do not attempt the extra credit until you have complete the core portion of the assignment.

- (5pt) **Choking Protocol**: Build a client that can do the above, and also implement the choking algorithm described in Cohen's specification.
- (5pt) **Tracker Protocol**: Add in features for communicating with a tracker server. Note that you do not need to implement a tracker.
- (10pt) **Interact with other Group**: Build a client that can do the above, but can also interact with another groups client. This must be demonstrated live!
- (25pt) **Real Working**: Build a client that can work with real world bittorrent clients. (HARD!)