Final Project Report

Chase Kanipe, Alexander Wu, Manuel Iribarren, Noah Bennison

Supported Features

Core Features

As required by the project specification, we implemented the following core features as described in the BitTorent specification.

1. Communication with HTTP trackers (with support for compact format)
2. Download a file from other instances of our client
3. Download a file from official BitTorrent clients

Extra Credit

In addition to the core features specified by the bittorrent spec, we also implemented several additional features mentioned in the project specification.

1. Support for the UDP-tracker protocol
2. Support for optimistic unchoking
3. Support for the rarest-first strategy
4. Support for endgame mode
5. Support for an optional BitTyrant mode

Design and Implementation

Preliminaries

To implement our client, we chose to use the C++ programming language. We chose it because it would enable our program to run efficiently, while allowing us to create more convenient abstractions than would be allowed by C, for example. To build our program we used the g++ compiler, along with makefiles to simplify the build process. Upon building, our program will compile into a single executable named “client”.

Program Inputs

Since our program is headless, it takes inputs as command line arguments. When invoking the program, one must specify at least two arguments: the first is a torrent file, and the second is either an output file or a file to seed. Based on whether the file passed to this second argument exists, the program will either attempt to download the specified file or enter seeding mode immediately.

Running the program with the wrong number of arguments will produce the following usage information.

```
$ ./client
Usage: ./client <torrent file> <output path / file to seed> <upload limit
in kB/sec> (optional)
```

An example invocation is shown here.

```
$ ./client deps/odyssey.torrent output.txt
```

The program may optionally take a third argument, which is an upload limit in kB/sec. This is used in our implementation of BitTyrant, and will enable this mode when passed to the program.

Program Outputs

Program outputs are sent through stdout. It will output basic information about the torrent file, connected peers, and download progress. Some example output can be seen below.

```
chase@chase-dell:~/school/final_project$ ./client test/large/udp.torrent
/tmp/output
Got torrent file:
  announce: udp://localhost:8000
  comment:
  name: bitwig
  pieces count: 905
  pieces length: 262144
  file length: 236988696
Bound to port 1025

Sending UDP tracker request
Received 1 peers

Listening for messages
Optimistic unchoking
Received bitfield from 127.0.0.1:1024

Connecting to peers...
Sending interest message
Received unchoke from 127.0.0.1:1024
Downloaded 5110/14465 blocks
```

```
[##################-----------------------------] 35%
```

Libraries

To encode and decode the torrent files, we used the open source bencode.hpp library, which can be found here: https://github.com/jimporter/bencode.hpp. We chose it because it supports C++17 and is header only, which enabled us to easily integrate it with the rest of our code base. We also used the SHA-1 library provided for assignment 4, as suggested in the specifications, in order to generate hashes for download validation and tracker requests. These are the only libraries used in our program.

Program Structure

Parsing

The main function of our program can be found in clients.cpp. In this main function, it parses the arguments entered by the user to obtain the hash file, determine whether it should enter download or seeding mode, and determine whether to enable BitTyrant mode. The torrent file path is then passed to our TorrentFile class, which does the parsing using the library and provides a simple interface for obtaining the arguments. If the parsing succeeds, the main function will invoke the tracker request, and then pass the resulting list of peers to either the download or seeding loop.

Tracker Announcement

The tracker code can be found in tracker.h. We implemented both the HTTP and UDP versions of this protocol. Which is called is determined by the prefix of the announcement URL obtained from the tracker file. The HTTP implementation supports both normal and compact format.

Downloading

The main file downloading loop can be found in download.h. To increase code modularity, we have separate functions to download all pieces, some specific piece, and some specific block. It will also confirm if the download of a specific block or piece has succeeded, and will update the progress bar when one succeeds. This file also contains our implementation of the rarest-first strategy, and endgame mode, both of which are executed by default.

The code in this class triggers various actions using the Peer class, which contains the socket and connection information for each peer, as well as functions to trigger connections, downloads, receives, etc. Most of these functions are invoked asynchronously (that is, invoking them starts a new thread where they are executed), to prevent blocking. New instances of the Peer class are created to handle information and

execution related to a peer upon receiving a peer list from the tracker, as well as when a new peer initiates a connection with our listening port.

Listener

The listener for incoming connections is implemented in the listener.h class. This loop runs on its own thread for as long as the program is active. When it receives a new connection it accepts the connection, creates a new peer object, handles the handshake, and adds the peer to the master list if the handshake succeeds.

Uploading

Uploading is handled mostly by the peer class, in various functions that handle requests received from the corresponding peer. Requests are collected by a separate thread, running code defined in peer.h, which unchokes up to a set number of interested peers at random. These requests are organized by the peer who sent them, and served by the same thread that collected them when the client is ready. Peer.h also contains most of the code for the BitTyrant implementation, which is enabled if the appropriate argument is passed in to the main function, and which replaces the random unchoking with a strategic algorithm that takes into account the specified maximum upload rate and attempts to maximize the number of reciprocating peers as economically as possible within the upload limit.

Testing

To ensure our implementation was functional we used both other instances of our client and 3rd party ones. Testing with remote files is difficult because most real world trackers we encountered either used HTTPS (which we aren't supporting), had no seeders, or were used for downloading files illegally. Thus, we conducted most of our testing using a tracker we set up locally (found here https://github.com/webtorrent/bittorrent-tracker), as well as torrent files we created ourselves that used localhost in the tracker address. We tested files as small as a few kilobytes, and as large as a gigabyte. Most of the related files can be found in the test directory.

Problems Encountered

Challenging Bugs

We encountered many obstacles when implementing our client. While there were too many to discuss fully here, below is a selection of the more interesting or challenging ones.

1. The bencoding library we initially chose crashed when attempting to parse some .torrent files. We had to replace it with the library currently in the repository, which proved more robust.
2. The client was receiving unrecognized message codes from its peers. The issue was that large packets, such as piece packets, were being fragmented by TCP and our

read call was non-blocking, so those portions which had not arrived by that time were not read until the next attempt to read, which presupposed that a valid BitTorrent header was at the front of the message. We were able to adjust the reading code to block until the full message had arrived, which resolved the issue.

3. Some of our downloads/uploads were resulting in incomplete outputs. The problem was that if hash verification failed for a piece, no attempt was made to redownload. We addressed this by adding a clause that would reset the piece's status to "unreceived" if the hash verification failed, causing the code to continue requesting the piece.
4. At one point our download was functioning fully except the final piece of a torrent was always corrupt. The problem was that the final piece wasn't necessarily the same size as the others, but we were treating it as if it were, and thus producing bad hashes. We fixed this by adjusting the buffer size for the last piece, based on the total size of the file.
5. At first our rarest-first implementation was not working as expected. The issue was that the calculations took place before knowing what pieces peers had. After identifying the issue, the code that calculates the rarest pieces was moved after the code that establishes connections and receives the pieces in their possession.
6. The program generated segmentation faults in some trial runs; we determined that this was the result of thread-unsafe handling of the data structures in which we tracked peers and received requests. Using C++'s mutex objects, we were able to wrap the functions that accessed these data structures so that no collisions took place, resolving the issue.

Testing Difficulties

Creating adequate tests for our client was difficult for a variety of reasons. We started by attempting to use open trackers and torrent files. However, this wasn't very useful to our group because it's hard to find well seeded torrents that don't use HTTPS (like debian), or are for distributing content illegally. Since no one in our group could port forward, we couldn't seed the file ourselves. This also limits the usefulness of the poole tracker. Thus, we resigned to mostly local testing with our own tracker, and testing tracker interaction against open source ones and poole.

Known Issues

Unsupported Features

Below is a list of notable unsupported features.

1. HTTPS tracker communication
2. Multi-file torrents
3. Re-announcing to tracker

Bugs

When connecting two of our clients to a local seeding rTorrent client, if we wait to start seeding until after both clients are connected (so the downloading starts exactly simultaneously), only one of the downloads will succeed. We are unsure if this is due to our client or due to how rTorrent tracks peers. There may be other bugs but our tests didn't reveal any.

Member Contributions

Chase Kanipe

1. Makefile and initial argument parsing
2. Bencode library integration and torrent file parsing with a new TorrentFile class
3. UdpStream class (a layer of abstraction for communicating over UDP)
4. Added timeouts to connect() calls in peer and other places
5. Code to start a new thread, listen for incoming requests, and add new peers to the peer list should the handshake succeed
6. Asynchronous dispatching of peer connections and handshakes
7. Sending interest messages
8. Made downloading methods in peer class multi-threaded
9. Updated peer class downloading code to be per-block as opposed to per-piece
10. Initial send_bitfields and send_have functions in peer (Alexander fixed up later)
11. Main loop for download mode and seeding mode (Alexander made edits)
12. Final logging code
13. UDP tracker request
14. Optimistic unchoking
15. End game mode
16. Report contributions
17. Various bug fixes and testing throughout

Alexander Wu

1. Set up the Peer class and skeletons for basic functions
2. Implemented handshake functions
3. Implemented storing Peer bitfields
4. Implemented sending requests to peers, first pass at download functionality (Later expanded and fixed by Chase)
5. Implemented requests handler
6. SHA-1 library integration
7. Handled timeouts and sending keep-alive messages
8. Code to send bitfields and handle incoming handshakes
9. First pass at upload functionality (Later expanded and fixed by Chase)
10. Rarest-first algorithm
11. BitTyrant mode

12. Ensured thread-safety with mutexes
13. Various bug fixes
14. Report contributions
15. Testing throughout

Manuel

1. HTTP tracker request
2. HTTP tracker request bug fixes
3. Testing of our client against bttrack

Noah

1. Bind a socket for peer listener
2. Initial report draft
3. Report revisions
4. Spelling fixes
5. Testing of our client against itself and rTorrent

Contributions Log

	Chase	**Alexander**	**Manuel**	**Noah**
Mon 29th	Makefile Argument parsing			
Tue 30th	Torrent file parsing Integrate bencode lib UdpStream class	Peer class		
Wed 1st	UDP tracker request Peer connect timeout	P2P connection		
Thu 2nd	Async peer connect	Peer handshake Track peer bitfields		
Fri 3rd	Listener skeleton			
Sat 4th	Async dispatch Send interest msg	Partial download		Bind to socket
Sun 5th	Redid bencoding Redid listener.h code Receive client conn	Timeouts Keepalives		
Mon 6th	Started upload	Fix download		

Tue 7th	Download per-block Send bitfield msg Send have msg			
Wed 8th	Generate info hash	Fixup send bitfield		
Thu 9th	Local tracker Upload fixes File seeding	Upload code Handle incoming handshakes	HTTP tracker request	
Fri 10th	Fixed download bugs Optimistic unchoking	Fixed download Fixed upload bugs Rarest first	HTTP tracker request	Report draft
Sat 11th	Redid logging Optimized download Fixed tracker bugs	BitTyrant mode Testing Bug fixes	Fixed tracker bug	Report revisions
Sun 12th	Testing HTTP req fix Report overhaul	Testing BitTyrant fixes Rarest-first fixes		Testing
Mon 13th	Report additions Randomize peer id Other bug fixes	Report additions Added mutexes	Report additions	Report additions Extensive testing