First Deadline Report, Bit-Torrent

Fall 2024, CMSC417: Computer Networks

Members:

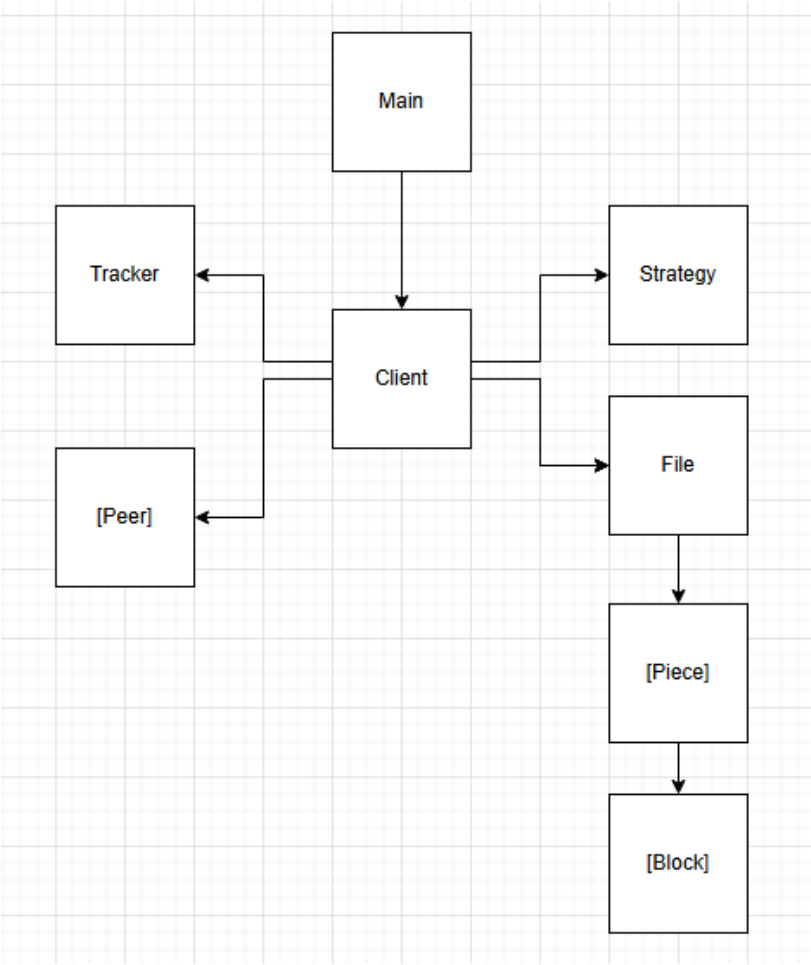
Kevin Goldberg
Steven Zhang
Eileen Yuan
Lee Forberg

Table of Contents

1. Decomposition and interfaces between components

1.1 High-level Overview

The Python project is split up into seven main areas with some additional helper files. The current architecture is as follows.



At the top, we have the main class which takes in the necessary information, validates it, initializes the client, and begins the download. The client class is the brains of the project as it is responsible for calling anything related to completing the file. The client file contains the main loop for our algorithm which can be seen in more detail in the client section. The client begins by talking to the tracker class which is responsible for handling all data and messages being sent and received from the tracker. The client will also initialize the file object at this time which is responsible for holding and handling any data associated with the file being downloaded. It will keep track of a temporary bitfield to send to peers and will remain in case of a stop and resume download. It will hold onto a list of pieces. Each piece class will hold the necessary information associated with the piece just as length, index, hash, and the necessary data structures for the blocks. Anytime a block is received it will be sent to the piece class to be saved in a data structure holding the block data with its offset and length. After this has been initiated it can join the swarm and upon doing so will receive its list of peers which will each be stored in a peer class. The peer class is responsible for tracking a single other peer and the connection between

the client and them. Any message being sent to that peer will be sent from here and any incoming message will be processed by the correct peer it came from. Lastly, the client initializes the Strategy class which is responsible for picking peers to unchoke and for finding the rarest piece to ask for. The main loop of the client class which will accept, send, and establish the proper epoch will continue until the download is either complete or stopped.

1.2 More and Specific Details

1.2.1 Client

The client class is the central controller of the program, calling the rest of the necessary classes and functions as necessary. It houses the main public API method download_torrent which features the main asynchronous event loop. It will maintain state for the following:

- Bytes downloaded
- Bytes uploaded
- List of peers
- ID
- Destination folder
- Information from the torrent file
- File

In the constructor, the client will extract the announced URL, info, and file information from the torrent file given to the program. It will use this information to initialize the File class. The main class will then invoke download_torrent which opens a socket to listen for incoming peer connections and contacts the tracker to join the swarm and receive addresses of peers. Control then enters the main loop that iterates until the file is fully downloaded. The layout of this critical section is as follows:

```
self.open_socket()
self.peers = self.tracker.join_swarm(self.file.bytes_left(), self.port)
self.strategy = Strategy()
self.epoch_start_time = datetime.now()
while not self.file.complete():
    self.accept_peers()
    self.receive_messages()
    completed_pieces = self.file.update_bitfield()
    self.send_haves(completed_pieces)
    self.send_requests()
    self.send_keepalives()
    if datetime.now() - self.epoch_start_time >= timedelta(seconds=EPOCH_DURATION_SECS):
        self.establish_new_epoch()
# Clean up resources
self.cleanup()
if self.file.complete():  # If file is complete, rename file to real name and delete bitfield.
    path = Path(self.destination)
    self.file.rename(path / self.filename)
    self.file.remove_bitfield_from_disk()
    return True
return False
```

Interface for Client:

open_socket()

Opens the socket and binds and listens

establish_new_epoch()

Resets start time and inform the peer class of a new epoch

execute_choke_transition()

Unchokes and chokes peers based on the top 4 strategy

accept_peers()

Accepts new peers until the maximum number of peers is reached

close_socket()

Closes the socket

add_peers()

Handles starting connections to all the peers in the peer list and also checking if connection completes.

send_haves()

sends a message to inform the other peer that we have the following pieces

send_keepalives()

Sends messages to peers to inform them that we have acquired new pieces.

send_requests()
Gets the correct peers and tells the peer class to request a particular block from tht peer

recive_messages()
Let the peer know that it has a message to receive from it

1.2.2 Strategy

The Strategy class is the brains behind the body of the application. The Client uses an instance of the Strategy class to export the decision making on which pieces to request from a peer, which peers to choke and unchoke, and which peers to join. By abstracting this part of the application, we are able to implement various strategies such as rarest-first or proportional share. This modularity also allows for easier benchmarking, testing and further experimentation.

Interface for Strategy:

choose_peers()
Given a list of peers, return a list of peers to unchoke. This is based on the top 4 strategies as well as an optimistic unchoke.

assign _pieces()
Given a list of peers (which contains their bitfields and whether they've unchoked us) and remaining pieces to download, returns a list of the rarest pieces to request from each.

send_bitfields()
Sends a message to inform the other peer that we have the following bitfield

send_haves()
Sends a message to inform the other peer that we have the following pieces

1.2.3 Tracker

The tracker's main purpose is to contact the tracker in order to join the swarm. It will hold onto the following information URL, timestamp for last contact, socket info, info hash, and if the request is in progress.

Interface for Tracker:

join_swarm()

Join swarm consists of many other functions but the main idea is to contact the tracker using HTTP, receive the response back, and process and extract the necessary information to initiate the download.

Send_tracker_request ()

Constructs and sends the tracker get request message. Called in join_swarm().

recv_tracker_response ()

Receive the tracker response, can be blocking or not blocking depending on args.

parse_tracker_response()

Parse the received data from the Tracker, decode the beconded dictionary, fill in the corresponding data structure fields.

send_scrape()

Send a scrape request to Tracker.

1.2.4 Peer

The peer class is responsible for tracking a single other peer and the connection between the client and them. It holds onto data specific to the peer and the connection, like:

- TCP connection state for peer
- Choking and interested states for client and peer
- Bitfield of blocks peer advertises
- Bytes received and sent (for top 4/propshare)
- Active outgoing/incoming requests
- Timestamps of last sent/received message
- Variables to contain state and fields to send handshake and bitmap at the start of connections

Interface for Peer:

Each send method will return a failure or success enum.

connect(self)

Connects using a nonblocking socket and returns a status enum

disconnect(self)

Closes socket and resets peer fields

send_handshake(self)

Send handshake message

receive_messages(self)

Assumes socket has data to read. Receives 1 entire bittorrent message and processes it accordingly. Current implementation blocks on recv until entire message is read. May need to update. Essentially every single receive method crammed into one.

send_keepalive(self)

Sends keepalive.

send_interested/not_interested(self)

Sends message and sets our state accordingly.

send_have(self, index)

Sends have message.

send_bitfield(self, bitfield)

Sends bitfield.

send_request(self, piece (piece object), offset, length)

Sends a request for the given params and adds it to outgoing requests.

send_piece(self, index, offset, data)

Sends a block of data if it was requested.

send_cancel(self, piece, offset, length)

If we have requested this block, send the cancel message and remove it from our list.

send_msg(self, msg (bytes type))

Helper method for send methods. Returns failure or success.

choke/unchoke(self)

If not already choking/unchoking, flip the boolean and send the message.

send_keepalive_if_needed(self)

Only send keepalive if no message has been sent for a while

establish_new_epoch(self)

Reset bytes received and sent for top 4/propshare calculation

disconnect_if_timeout(self)

If we haven't received a message in a while, disconnect.

1.2.5 File

The file class is initialized by the client and is responsible for handling any information that will be saved to the file. This included the list of pieces, the name of the file itself, and a temporary bitfield map to ensure that if we stop the download mid-way through we can pick off where we left off.

Interface for File:

get_bitfield()
    Returns the bitfield representing what pieces we have completed

update_bitfield()
    Check each of the pieces to see if any are complete if so will add them to the bitfield

get_download_percents()
    Gets the total download percentage of bytes downloaded to total bytes.

complete()
    Returns True if the file is complete, False otherwise.

bytes_left()
    Calculates how many bytes are left of verified data

rename()
    Renames the file being downloaded from .part to the correct extension

remove_bitfield_from_disk()
    Removes the bitfield we were using

1.2.6 Piece

The piece class is responsible for holding all necessary information associated with the particular piece. It will hold onto the index, length, and hash of the piece all of which was acquired from the torrent file. It will keep track of the following data with a dictionary of blocks where the key is the offset and the value is the block class which will hold onto the data. It also holds onto additional data such as completed, downloaded, and outgoing requests, and some additional data useful for performing the class functions. It bases the block lengths on the standard 16KB. It will hold onto half-completed pieces in memory and will write to the disk and free the memory after a piece has been completed and hashed.

Interface for Piece:

get_next_request()

Gets the next request to send out to a peer and makes this decision based on what it has sent already and the timestamp. It will return the offset and length to ask for. The lengths and offset will be based on the 16KB unless it's the last block

add_block()

Upon receiving the response from a peer it will add the block to the data structure we have and update the necessary fields. If it's finished it will assemble a full piece and check with the hash. If everything is equal it will write it to the disk at the correct location and set itself to complete

get_data_from_file()

Gets the data in bytes that a peer requested from the disk.

1.2.7 Block

The block class is responsible for holding onto a particular block that we had requested from a peer. The block class is initialized by a peer and will hold the data in bytes and the length. Everything else is done in the piece class.

Interface for Block:

get_length()

Gets the length of a block.

get_data()

Gets raw data in bytes.

2. Work divide and Progress on Project

2.1 Work Divide

Lee:

- Piece and block class code - done
- Help with file class code - done
- Test piece, block, and file - done
- Document due on 5th writeup and layout - done
- Helping to debug the merge/combination of everyone's work - in progress

Kevin:

- Implement group's object-oriented plan, define APIs and program structure - coded
- Client class - coded
- Strategy class - coded
- Started file class and bitfield persistence implementation - coded
- Rarest first strategy extra credit - coded
- Client/peer/strategy coordination - in-progress
- Prop share - designed
- Unit testing across various classes - in progress

Steven:

- Torrent experimentation & writeup - done
- Block/piece storage writeup - done
- Peer class and test_peer.py - done
- Merge peer class to main, bugfix - in progress

Eileen:

- Tracker class code - done
- UDP Tracker extra credit - in progress
- Testing with Wireshark - in progress
- Help debug code

2.2 Current progress and state

We are just about at a point where we can begin testing on actual torrent files. Each class has been finished or is 90% done and has been tested individually. Currently, we are working on bringing each section together which will require some debugging and trial and error. We hope to have the project functional in the next few days in order to further test and start with some extra credit/cleaning up the project.

3. Tentative testing/experiment scheme

We have already tested most of the class individually as we have a test folder with the following files:

- Test_client
- Test_file
- Test_file_piece_block
- Test_peer

- Test_tracker

We are currently in the process of testing all of the classes working together with a full process and plan to continue this over the next few days.

4. Extra credit plan/progress

4.1 Completed

- Rarest first - currently implemented in the strategy class as it finds the rarest piece to request

4.2 Tentative plans

- Propshare
- UDP tracker or HTTPS tracker
- Multifile

5. Appendix

Additional evidence of work on project planning and testing, but not part of report submission for a grade:

- **Group Google Doc**