BitTorrent Decomposition Document

Ryan El Kochta
relkocht

Andrei Kotliarov
akotlia1

November 30, 2022

Contents

1 Contributions

Andrei and Ryan sat down and wrote this report together, and both understand all of the components listed and their responsibilities. It wouldn't really be fair to say that one person worked more or less on the report; the whole thing was done collaboratively.

2 Components

2.1 Argument Parsing

Our BitTorrent client will be a command-line program. The user will configure the client using arguments. Currently we plan to implement the following arguments:

`-t, --torrent`:	torrent file to download
`-c, --max-connections`:	maximum number of peer connections, defaults to 10
`-p, --port`:	listening port, defaults to random
`-t, --tracker-type`:	force tracker type. `http` or `udp`
`-d, --directory`:	output directory, defaults to current directory
`-b, --bittyrant`:	enable BitTyrant mode

Argument parsing will be done using the `clap` crate. The `Args` struct will contain fields for every option, and its method `Args::get()` will lazily parse the arguments and return a reference to them. We expect to use this interface throughout the code.

Andrei and Ryan will aim to have this portion of the code done by December 5th.

2.2 Multithreaded Architecture

Our client will be architected as follows: the main thread will actually maintain the download state (which blocks we've received, all of our peers, etc). The main thread will spawn one thread for each connection with another peer. Each of these threads will be able to perform *blocking* I/O, and send/receive instructions with the main thread using channels (`std::sync::mpsc` for multi-producer, single-consumer queues).

The main thread will be able to destroy threads, disconnecting peers. It can also create threads, creating peer connections. The tracker thread will remain static throughout the lifetime of the program; if it ever exits, this is considered an error and the program will panic. There will also be a similar thread dedicated to accepting new connections and passing them to the main thread using the channel. In addition, there will be a thread dedicated to handling periodic timers (like when to update choking/unchoking, tracker interval, etc).

We plan to work on this together, having the above done by the Dec. 5th check-in. However, we are considering another slightly-tweaked architecture, where each peer thread polls on the fd it's responsible for and an eventfd. When we send a request to the thread that needs immediate attention, we can send an event to the eventfd, waking poll immediately. In this case the I/O would actually be non-blocking.

2.3 File I/O

The core of our file I/O interface will be the `Piece` struct. A `Piece` will have a range of bytes in the output file that the piece represents, along with a SHA-1 hasher. We will also have a `DownloadFile` struct, which will maintain a list of pieces, a bitmap specifying which pieces have been fully received, and the correct SHA-1 hash of the piece.

The main loop, upon receiving a block from a peer, will call `Piece::process_block()`, which will add it to the appropriate buffer and incrementally compute the SHA-1 hash of the piece. `DownloadFile::piece_bitset()`

will also be exposed so that the later game-theoretic components can decide which pieces to request according to their own logic. Other components will also use `Piece::remaining_range()` to decide which pieces to request from other peers.

Once `Piece::process_block()` fills in the last block, it will compute its SHA-1 hash and compare it.

For convenience, outside components can call `DownloadFile::is_complete()` to see if we have the entire file.

```
impl Piece {
    fn process_block(&mut self, buf: &[u8], offset: usize);
    fn remaining_range(&self) -> Range<usize>;
}

impl DownloadFile {
    fn piece(&self, i: usize) -> &Piece;
    fn piece_mut(&mut self, i: usize) -> &mut Piece;

    fn piece_bitset(&self) -> &BitSlice<usize, Msb0>;
    fn is_complete(&self) -> bool;
}
```

Ryan will finish the file I/O component by December 5th.

2.4 Message Encoding & Decoding

We plan to message encoding/decoding using the `serde` and `bendy` Rust crates. We will need to manually implement the peer protocol payload parsing (although the id and length will be handled and stripped by the peer threads). The API will look something like this:

```
const SHA1_LENGTH: usize = 20;

struct Torrent {
    announce: String,
    name: String,
    length: usize,
    piece_length: usize,
    piece_hashes: Vec<[u8; SHA1_LENGTH]>,
    info_hash: [u8; SHA1_LENGTH],
}

impl Torrent {
    fn from_bytes(bytes: &[u8]) -> Self;
}

impl TrackerResponse {
    fn from_bytes(bytes: &[u8]) -> Self;
}

// peer responses, corresponding to each peer request below
```

Andrei will finish this component by Dec 5th.

2.5 HTTP/1.1 Client

Our HTTP/1.1 client wlil be very simple, supporting only one operation: sending a GET request and receiving a response. The API is fairly self-documenting:

```
struct HttpResponse {
    status: u32,
    content: Vec<u8>,
    headers: HashMap<String, String>,
}

fn http_get(url: String, parameters: HashMap<String, String>)
    -> Result<HttpResponse, HttpError>;
```

Since we're using threads, it's okay for `http_get()` to block.

Andrei will do this component by Dec. 5th.

2.6 Tracker Interaction

```
enum TrackerRequestEvent {
    Started,
    Completed,
    Stopped,
}

struct TrackerRequest {
    info_hash: &[u8],
    peer_id: String,
    ip: Option<IpAddr>,
    port: u16,
    uploaded: usize,
    downloaded: usize,
    left: usize,
    event: Option<TrackerRequestEvent>,
}

enum TrackerResponse {
    Success(u32, Vec<Peer>),
    Error(String),
}

impl TrackerRequest {
    fn send(&self, dest: impl ToSockAddrs) -> TrackerResponse;
}
```

The main thread will periodically construct TrackerRequests and send them to the tracker thread over the channel. The tracker thread will then send it over the network, block until it receives a response, and then send the TrackerResponse over the channel to the main thread. The main thread will use the peer list returned in the TrackerResponse to update its peer list, connecting to/disconnecting from peers (and creating/destroying the appropriate threads) to maintain the maximum connection limit.

See below for the `Peer` type.

Ryan will have this component done by Dec. 5th.

2.7 Peer Protocol

```
struct Peer {
    // basic info
    ip: IpAddr,
    port: u16,

    // game theory
    am_choking: bool,
    am_interested: bool,
    peer_choking: bool,
    peer_interesting: bool,
    bytes_sent: u64,
    bytes_received: u64,
    start_time: std::time::Instant,
    bitmap: BitVec<usize, Msb0>,
    when_joined: std::time::Instant,
}

enum PeerRequest {
    Choke,
    Unchoke,
    Interested,
    NotInterested,
    Have(u32),
    Request(u32, u32, u32),
    Cancel(u32, u32, u32),
    PeerInfo, // not actually sent over the net
}
```

At the start of the program, the main thread will wait for the tracker thread to receive a set of peers. It will then create threads, passing them a Peer struct over the channel. After this, it can send PeerRequests to each thread, after which the thread will send the appropriate request over the network. The exception to this is the `PeerInfo` request, which will cause the receiving thread to send the main thread statistics which it can use for game theory decisions (see below). The PeerRequests will have an associated PeerResponse sent over the channel.

This is a very large component that is core to the program, so Andrei and Ryan will both work on this together. Our aim is to have this done by Dec. 5th (although to have it fully working by then is probably optimistic).

2.8 Game Theory

The first part of the game theory component is the part that will decide which peers to choke and which peers to unchoke based on their actual performance. The next part will do optimistic unchoking, so that all peers are eventually discovered. Finally, part of the game theory component will be dedicated to picking which pieces we should request from which peers.

This is a little bit less concrete at this point than the other interfaces, so we don't know exactly what the functions will look like yet. It is likely that we will have traits such as `ChokeStrategy`, `PieceStrategy`, and so on that can be implemented by all of the different strategies that we choose to implement, allowing them to be switched out easily. Those implementing these traits will have functions like `should_choke()`, `which_piece()`, etc.

For example, Top-4 can be implemented as a `ChokeStrategy`, and rarest-first can be implemented as a `PieceStrategy`.

An important note is that all of this will be performed by the main thread. It will coordinate obtaining information about all of the peers and sending it to the game theory component, then instructing all of the peer threads to send requests appropriately.

Like the peer protocol, this is a deeply interconnected with the others component, so Andrei and Ryan will do this together. We won't aim to have all of this done by Dec. 5th; instead, let's say Dec. 10th.

3 Testing

3.1 General

We will do most of our testing using the Rust/Cargo built-in testing suite, including a `mod test` in each file and using the `#[cfg(test)]` and `#[test]` macros. This will be used for unit tests.

3.2 Argument Parsing

This doesn't need to be thoroughly tested because almost all of the logic here is delegated to the `clap` crate.

3.3 Multithreaded Architecture

This will mainly be tested in practice. Some possible unit tests could spin up threads and test our channel communication strategy.

3.4 File I/O

We will write unit tests, passing fake blocks and constructing fake pieces, ensuring that each function returns the correct result (for example, the requested ranges and bitmaps are correct). Also, we should try this for very large files to ensure that any buffering is efficient. We could even make a mock file (using something that implements Seek and Write) instead of a real file, so that the tests are more reliable.

3.5 Message Encoding & Decoding

We will construct a number of bencode test cases, and write unit tests that make sure our encoding and decoding construct the correct serialization and deserialization of various data.

3.6 HTTP/1.1 Client + Tracker Interaction

We will write some unit tests that send requests to various public trackers, parse the responses, and verifies that the resulting data structures are correct. This will test both the HTTP/1.1 client and the tracker request/response parsing components.

3.7 Peer Protocol + Game Theory

As with the threading, this is very difficult to write proper unit tests for. Instead, we will inspect manually and in practice; for example, we will inspect transcripts of our peer protocol using a tool like Wireshark to ensure that the correct data is being sent over the wire. We can also measure performance of the various game theory strategies that we implement (using a common large torrent file) to compare them (definitely legal Linux ISOs only). We will also compare to commonly available torrent clients such as Transmission and Deluge.

A Libraries Used

We currently plan to use the following libraries:

1. `clap`
2. `libc`
3. `bitvec`
4. `serde`
5. `bendy`
6. `sha1`