CMSC417 BitTorrent Report

Andrei Kotliarov Ryan El Kochta

December 2022

1 Supported Features

1. Command Line Interface:
   (a) Specify torrent and other arguments (e.g. max connections, existing file, etc.) as arguments to executable
   (b) Uses Rust `log` crate for logging output
2. Tracker Interaction:
   (a) Support for HTTP/1.0 and HTTP/1.1 for tracker requests
   (b) Support for compact and dictionary modes of peer list
   (c) Periodic communication with tracker to keep peer list full
3. Downloading:
   (a) Speeds of up to 10-15 MB/s over Wi-Fi (measured on Debian disk torrent)
   (b) Configurable request queuing and max connections
   (c) Peer-ranking mechanism - high upload rate peers are kept when obtaining new peers from tracker
4. Seeding:
   (a) Other clients able to download from ours (tested with qBittorrent)
   (b) Our client is able to download from other instances of itself
5. File I/O
   (a) File subsystem manages the pieces/blocks of a file
   (b) Provides interface to obtain unfilled block ranges, which are used by the main thread to coordinate peer protocol interaction
   (c) Provides piece SHA1 verification on completion, verifying file piece integrity

2 Design Choices

Overall, we stuck with the general plan of our architecture in the decomposition document fairly well. The main change was in how the main thread operates and interacts with the sub-component threads; instead of each thread entirely handling its own isolated system (e.g. the peer thread handling pieces/requests), they are used as an abstraction to be able to remove nearly all blocking IO calls from the main thread, allowing for greater efficiency in handling the protocol.

The greatest change was to how our main loop was laid out in terms of file I/O. Originally the plan was to have each piece maintain a "cursor" that moved forward as we requested blocks. However, this solution was clearly untenable since it effectively made it so only one block of one size per piece could be requested at a time, so we quickly rethought it. Currently, the main loop is structured around a block request queue and a main receiving channel. As we obtain peers from periodic requests to the tracker queue, we are able to keep track of and see which peers have which pieces, and request what we need accordingly. These requests are added to the block request queue and also have timeouts registered in a timer thread. These requests are then sent off to some selected peers (currently random, but can easily be made to follow a game theory strategy), who are expected to return a response within some time. If the timeout in the timer thread runs out before this happens, the main thread is notified and closes its channel to the corresponding peer thread, which will then "evict" the peer and make room for a more cooperative peer to be acquired from the tracker later.

This structure allowed for creating a pipeline/queue of requests per peer, and makes sure that each connected peer is always busy at any given moment. However, there were some considerations that had to be made to make all our logic nonblocking. We considered using async Rust crates to be able to use green threads, however these were disallowed. Therefore, we ended up settling on Rust's threading and channels from the `crossbeam` crate to be able to easily pass messages between different threads. It's less optimal than green threads since we're almost entirely I/O bottlenecked in this protocol, but we decided that creating a proper event loop from scratch in Rust was going to be a lot more work than it was worth, and that channels could make a threading design fairly elegant.

To accomodate this block-based system, the filesystem component was redesigned to work with ranges of blocks. A `DownloadFile` (the user-facing part of the API) is able to provide a list of "unfilled" blocks for each individual piece, which can then be queued to a peer in whatever order is deemed necessary. Only when all of these blocks are filled is the piece SHA1 hash calculated to verify its integrity. This API was extensively tested (with probably the most extensive unit tests of our project) to ensure that we wouldn't see any hard to debug issues down the line. This paid off when we had to add more features to it in order to be able to support more features like seeding.

In terms of internal things like the message encoding/decoding (bencoding), we made very heavy use of the `bendy` and `serde` libraries. These two libraries

have an extremely ergonomic design, and made doing what would have been otherwise very mundane tasks as easy as just writing some annotations above each of the fields in whatever struct we wanted to (de)serialize.

3 Problems Encountered

One large problem that we dealt with was removing blocking calls from the main thread. While we initially decided to just split off the systems into individual threads, this quickly proved to be a fairly naive approach. The main issue was that these threads had to do blocking operations with I/O (peer interaction through a TCP socket), but also had to be able to take commands from the main thread through a channel. As far as we were able to tell, there was no system in place in the standard library/crossbeam to allow for polling on these two different types of interfaces at the same time.

This is where we decided on having the main thread keep track of most of the state, so that it wouldn't be forced to ask the potentially blocking system threads for information (and therefore take longer than expected to receive it). However, there was still a problem for the peer protocol - duplex communication was expected. We weren't able to just block on reading, since writing to the `TCPStream` on demand was also a necessity (and vice-versa). This necessitated the creation of another "receiver" thread that would act as the glue to block on the reading side of the `TCPStream`. From here, we decided on the crossbeam crate in order to make use of the `select` feature it offered on its channels (similar to the select seen with Go channels), which was able to multiplex between the reading and writing for an individual peer.

Another problem we had was the max size of a request. While Bobby did talk about this on Piazza, we had missed it, and used a 32k size since that is one of the sizes mentioned on the BitTorrent Spec Wiki. This resulted in some strange behavior where almost every peer would ignore us, except for the occasional peer that actually serviced our bad request (and made us think the issue was something else unrelated). Some more careful reading of the specification was eventually able to resolve this problem.

We also had a problem with regards to dealing with URLs. The infohash parameter of the tracker request requires each byte to be url-encoded. However, the Rust `url` crate took query parameters as `&str`, which is UTF-8 encoded as opposed to encoding each individual byte. This forced us to write our own routine to create this type of URL, which was an annoyance on top of what is otherwise a good interface.

We also encountered a problem with the class Flatland torrent where the tracker returned a peer list containing a peer with ip `le` (the bencode representation of an empty list). We were not able to determine what the cause of this was, but dealt with it by simply ignoring invalid entries in the `peers` list.

4 Known Issues

For large files, we have noticed that our download speeds intermittently get slower. This generally requires file sizes with sizes greater than at least 2GB to be very noticeable, although it can still be noticed with smaller files.

We can also see why something like Endgame would be useful to implement (ours doesn't), since the last few pieces take dramatically longer than the other peers to come in. This is because with our architecture we have a timeout that is attached to a request, and no other peers can be sent a request for that block while that timeout is active. This is fine when each peer has a full queue of requests, since the good peers can always make forward progress even if the bad peers are slow. However, a bad peer with a slow upload rate will end up waiting all the way until the timeout to "free" this block and allow other peer threads to request it, which leaves lots of time when no progress is being made.

5 Contributions

The only two components with a clear distinction on who worked on what were the Tracker + HTTP components (which Ryan worked on) and File IO component (which Andrei worked on). The other components were worked on by both of us, with both the design to the code done collaboratively. We were able to do this since most of the time when we worked on the project it was either in person or we at least were in contact with each other about any API/design changes that were being made.