itty bits

Nicole Glabinski, Everardo Rosales, Kimberli Zhong
6.824 Spring 2017 Final Project

Source code: https://github.com/everrosales/bittorrent

Overview

BitTorrent[1] has been the defacto solution for distributing large files over the internet as it addresses the increased burden on hosts. It is one of the largest peer to peer systems on the internet today. BitTorrent allows peers to share pieces of files amongst each other rather than relying on a single host to send them all parts of the file. BitTorrent is also resilient to client or network failures, cleanly resuming the download with other peers when they are available.

To distribute a file, the host simply needs to run a tracker and seed the file on one initial peer. The tracker is tasked with helping peers discover each other, and the peers connect directly to each other to download the pieces of the file they need.

Design

Metadata File

The metadata file, or *torrent* file, contains all the information needed to begin downloading the file it is associated with. It contains the URL of the tracker as well an an `info` dictionary consisting of SHA1 hashes of small chunks, or *pieces*, of the file, as well as the piece length and several other keys. The tracker provides addresses of peers to connect to and the hashes allow the client to verify each piece of the file it receives from peers.

Tracker

When a client wishes to download or seed a file, it sends a GET request to the tracker specified in the torrent file and receives a list of peers also downloading or seeding the file. The tracker is an HTTP server which keeps a list of clients which have contacted it recently. The GET requests contain a SHA1 hash of the info dictionary in the torrent file, a random peer id, and optionally an ip and port. The response is a bencoded dictionary containing a list of peer addresses and the heartbeat interval the client should use for contacting the tracker.

[1] http://www.bittorrent.org/beps/bep_0003.html

Client Protocol

When clients connect, they exchange a handshake as well as *bitfield* messages, which encode which pieces of the file the client currently has. From here, clients send out *request* messages asking for a part of the file. Request messages deal in *blocks*, which are a subunit of pieces. Blocks are usually 16 kiB while pieces are usually 32 kiB[2]. A request message specifies the piece number, byte offset within the piece, and number of bytes the client wants. Upon receiving one of these, a client can send back a *piece* message which contains the data for the block. Once the original client has received all the blocks for a piece, it sends out a *have* message to all peers to inform them that it has that piece.

Implementation

Client

The client has several main tasks: sending heartbeats to the tracker, checking which pieces of the file are needed and requesting them, and handling incoming and outgoing peer messages. Each time it receives a response from the tracker, it merges the new peer list with its current list. Then, it initializes connections with any new peers and sends heartbeats to existing peers.

The client iterates through all the file pieces in a random order for requests. For each piece, the client iterates through the peers it has connected to in a random order and requests from the first one it knows has this piece. These two sources of randomness aim to avoid overwhelming a single seeder with piece requests. Instead, each peer will eventually end up with a random subset of pieces and they will be able to share these among the swarm of peers.

After sending out requests for all the blocks in a piece, the client waits to receive piece messages. If a timeout passes without receiving the entire piece, the client returns the piece to the queue of pieces it needs to download. When piece messages do arrive, the client stores the block in memory and marks in volatile state that that block of the piece has arrived. Once all the blocks in a piece arrive, the piece is checked against the piece's hash stored in the torrent file. The piece is persisted to disk if it matches, or discarded otherwise. The information persisted includes the data of each piece as well as the piece bitmap that marks which pieces the client currently has.

The pieces are stored in memory as a list of blocks and persisted as encoded Go lists for convenience when starting the client from a paused download. The fact that the pieces are held in memory is convenient for servicing request messages but is problematic in the case of large files. An alternate solution would be to remove the piece from memory after persisting and read it from file I/O whenever the data needs to be sent to another peer.

---

[2] https://wiki.theory.org/BitTorrentSpecification

Tracker

The tracker is implemented as a standalone HTTP server that watches for requests with specific parameters. When the tracker is started up, it reads in a torrent file, calculates the hash of its `info` field, and handles client requests. Clients check in with the tracker in regular heartbeat intervals, updating the tracker on their progress so far and providing the torrent's infohash to verify that they are peering for the right data. The tracker stores a map of the peers it has seen in non-persistent storage and has a timeout so that if a client stops sending it heartbeats it's no longer considered an active peer.

The system is designed to work even in the face of tracker failure. Even if the tracker dies in the middle of a download, peers can still contact their known peers and communicate with them directly about pieces to exchange. However, in our current iteration, peers are unable to discover new peers without our centralized tracker.

Peering

Our current implementation of BitTorrent separates much of the networking logic into a few convenience methods that help limit its complexity. Upon initialization, each client opens a TCP port to listen for incoming connections from peers, and this port is reported to the tracker to broadcast to other peers. When a connection is established, it is handed off to a message handler function that loops until the connection is closed. It'll read a message from the TCP connection, and then enqueue a response if needed.

Outbound messages are enqueued to a Go channel of peer messages. These are then handled by a single Goroutine as they write out on the TCP connection. Other Goroutines simply need to enqueue the message to the Go channel through a convenience method and TCP connection dialing, message transmission and connection keep-alives are all handled for the caller.

Currently our implementation lacks a choking algorithm. This would greatly improve performance of the our client, but due to time constraints we were unable to fully complete and thoroughly test our implementations of peer choking.

Attempts to mitigate this pitfall included dropping stale packets in the message queue as well as preventing messages for no longer necessary pieces in the message queue from being sent out. While trying to drop stale packets we found that we ran into numerous issues while trying to delete and create new message queues. Dumping messages out of the queue by reading from the message queue and not sending the messages proved to be too slow in practice.

We did manage to implement a way to prevent duplicate messages such as requests for the same piece. To do this, we simply maintained a set of messages inside the queue and removed them from the set when they we taken from the queue. Any duplicate message that would be enqueued would instead be dropped.

Challenges

Just like with Raft, building a distributed system for BitTorrent came with a number of concurrency and timing intricacies. We tackled these challenges posed by parallelism with a similar approach, adding mutexes onto the client, tracker, and TCP peer data structures to protect state reads and updates from data races. Weeding out deadlock and race conditions was hard, but we managed to make our system generally reliable even in the failure conditions in our integration tests.

We also encountered difficulties with the specifics of the BitTorrent protocol, since SHA1 hash characters weren't always UTF-8 encoded, and transmitting those characters over HTTP required that they be escaped. Debugging piece hashing took a while, and in particular, the way Go handles string indexing made us take a few iterations to get what should have been a straightforward helper function that splits a string into pieces of length N correct.

Managing the peers' message queues is also a complicated task, and we did make some progress on preventing backlog and timeouts, but we still have much room to grow in this regard. Having our TCP logic as a layer below client application logic was ideal abstraction-wise, but we still have issues with leaky goroutines and peer connections in that lower layer causing cumulative tests to run slowly.

Testing

We wrote a unit test suite for each of the individual modules of the system: the client, the tracker, peer communication, torrent file utilities, and general debugging utilities. To easily verify our system's correctness and fault-tolerance during the development process, we used these unit tests as well as integration tests that automatically spun up trackers, seeders, and downloaders and checked whether files were successfully transferred between clients.

We also were able to successfully manually download files (including a 20 MB file) across different machines with multiple concurrent seeders and downloaders.

Future Work

A number of improvements would make our system more performant and scalable. For example, implementing an “endgame” strategy would improve our client's performance when downloading the last few pieces. This could be as simple as sending the same request for the last few pieces to multiple peers, followed by cancel messages once we receive the piece from any of our peers.

The current implementation also holds the entire file in memory as well as all of the metadata that we use to handle peering. This is fine for the smaller files that we used during testing but is impractical when sharing large files. Ideally, we would want to have a clever way of flushing and reading from disk to limit the amount of memory that we are using during the seeding process.