BitTorrent Protocol

Team 5: Valeria Jovel, Ethan Lau, Harrison Kim, Yermek Mukashev

University of Maryland, College Park

CMSC417 - Computer Networks - Fall 2023

List of supported features

- **UDP Tracking Support**
- **Rarest first piece downloading**
- **Optimistic Unchoking algorithm**
- **Seeding support**

<u>cmd/client</u>
main.go :

- The program starts with a comment describing how to use it, stating that it should be executed as ./client [torrent file] [output path].
- This program is a basic BitTorrent client that takes a torrent file and an output path as command-line arguments, attempts to create a BitTorrent client, and logs any errors encountered during the process. The actual functionality of handling BitTorrent operations is implemented in the core package, specifically in the NewClient function.

<u>internal/core</u>
client.go

- Client.go contains the main logic for the program. The “NewClient” function is essentially the real main function that initiates the parsing of the torrent file, the initial interaction with the tracker, and the queuing of pieces to download.

hash.go

- This file contains a helper function to get the SHA1 sum of a byte array.

http.go

- This file contains a helper function to make an HTTP GET request using TCP.

<u>internal/data:</u>

file_handling.go

- File_handling.go contains routine functions that are called as goroutines for each peer (sender/receiver routines). These routines handle message receiving and sending loops for each peer. Communication between the two routines and the main loop is done via channels. It also contains functions to help find the top four peers to unchoke as well as a peer to optimistically unchoke.

file_metadata.go

- File_metadata.go contains a FileMetadata struct that represents a file to be downloaded. It contains relevant file information parsed from torrent file such as the length, path, name, etc. It also contains a function to parse a map[string]interface{} (the result from the bencoded torrent file) into a file struct.

peer.go

- This program defines a Peer struct and associated methods for handling communication in a BitTorrent client. Key functionalities include establishing connections, handling handshake, parsing various message types, and sending messages. The Peer struct represents a connection to another peer in the BitTorrent network and includes details such as peer ID, IP address, port, and connection status. The code implements the BitTorrent protocol's message types, including Choke, Unchoke, Interested, Uninterested, Have, Bitfield, Request, Piece, and Cancel.
- Additionally, the code defines methods for packing and sending messages, checking piece availability, and finding reachable peers based on their bitfields. The error handling and message parsing mechanisms are designed to ensure the proper functioning of the BitTorrent communication protocol.
- Overall, the code provides essential building blocks for handling peer-to-peer communication within a BitTorrent client, adhering to the protocol specifications.

piece.go

- Each piece contains information such as its status, hash value, index, starting position, length, associated data block, the number of peers possessing the piece, and its rarity factor. Additionally, the package includes a sorting function, SortTorrentPieces, which utilizes the Go standard library's sort package to sort an array of TorrentPiece instances based on their rarity in ascending order. This sorting function enhances the efficiency of managing torrent pieces, allowing for prioritization based on rarity, a crucial factor in optimizing the distribution of pieces among peers during the BitTorrent protocol implementation.

torrent_metadata.go

- Torrent_metadata.go contains a TorrentMetadata struct that represents a torrent file and its field as in the bencode dictionary. Its fields include things such as the announce addresses, info_hash, pieces, etc. It contains a function to parse a torrent file into a TorrentMetadata struct as well as a helper function to parse the “info” section of the torrent file.

tracker_information.go

- Tracker_information.go contains a TrackerInformation struct that stores information received from the tracker. This struct is also where the peers are stored. This file also has a function to take a decoded bencode tracker response and turn it into a TrackerInformation struct.

Design and implementation choices that you made

1. First of all, we should mention the language choice. We chose Go because of the restriction on third party libraries. Go comes with very nice asynchronous capabilities straight out of the box with goroutines and channels, so this made Go a very attractive choice due to the project being network related. Asynchronous support significantly helps in a networking environment where there is a lot of waiting around for connection initializations, writes, reads, etc.
2. There are a lot of moving parts within the BitTorrent protocol, so we decided to split it into logical components. For example, all the torrent parsing code would be one section and all the tracker interaction in another. By making the components loosely coupled, we reduce the need to wait on the completion of other parts. Instead, we can develop independently and then glue together later.
3. Another design choice we made was having peers act sort of like actors in an actor model. The main benefit for this is that we can have the state mostly remain local without needing to share state behind mutexes or locks. We instead communicate with peers through channels. This reduces complexity and allows faster development.

Problems that you encountered (and if/how you addressed them)

1. One problem that we encountered was the actual implementation of the peers. While we had a general idea of how the peer was going to be able to do, and what each peer needed to do, we did not know how to implement it. Eventually, we decided on having two goroutines spawned for each peer connection. One would be for reading from the connection, the other would be for downloading or sending. While some messages could be handled in one-shot, such as “interested” messages, messages such as “have”, “piece”, and “request” either need more logic to process them or need to be forwarded somewhere else. By having two goroutines, we can send these special messages to the corresponding goroutine to be processed.
2. Dealing with failed downloads proved to be another challenge. In our implementation, we have a global queue of tasks in which downloaders pull from. If we fail a download, it would be bad if the downloader put the piece at the back of the queue, especially for rare pieces, so we have a priority queue instead. If a download fails, the downloader will put the task into the priority queue so that the

Known bugs or issues in your implementation

1. If we were to ask a peer to download something, and the peer does not respond indefinitely, we will have a big problem
2. We continue to ask peers for 4 byte ints (length of message) even when they may not be sending messages
3. Our downloaders do not get refreshed as they should
4. No seeding after we are finished downloading

Contributions made by each group member

- Valeria Jovel
  - Setting up the peer.go functionality, constants, and functions for the peer wire protocol, to initialize peers, connect to peers, and establish the handshake between peers.
  - Contributed to the parsing and handling of messages sent between peers, and processing “bitfields” with each other peer.
- Ethan Lau
  - Did tracker interaction
  - Helper with BitTorrent logic
  - This guy is the goat ~ Harrison
- Harrison Kim
  - Did most of file handling/downloading stuff
- Yermek
  - Worked on send/processing of messages