CS 262 Final Project Write Up - BitTorrent

Andrew Zhang and Maxwell Zen

April 2025

1 Introduction

This file serves as the final write-up for our project. It contains the specifications of our project, what we were able to accomplish, and our findings. This file also contains all of the documentation of our code, as we discuss all of the functions of each file in the Implementation section. Our engineering notebook is in the GitHub repo under `notebook.txt`.

Our project focuses on the design and implementation of a simplified peer-to-peer (P2P) file sharing system inspired by the architecture and design of the BitTorrent protocol. The main distinction of this type of distributed system in a file-sharing context is that users can download files by leveraging the bandwidth of other users, as opposed to have the entire load on a single server. This has a number of benefits, including much higher download speeds and increased fault tolerance.

2 P2P and BitTorrent

Rather than relying on a centralized server to store and serve files to clients, a P2P system distributes the responsibility of hosting and exchanging data across a network of clients, known as *peers*. Peers act as both server and client: they upload and download different parts of the desired file from a network of other peers. Peers that are downloading from other peers are known as *leechers*, while peers uploading to others in the network are known as *seeders*. Under this model, there is no need for a central server, thereby bypassing the download bottlenecks that would be created be a single server handling many clients. Additionally, the existence of a single server creates a single point of failure for the entire network, but P2P networks can easily handle any one peer crashing, as the rest of the peers are still able keep sharing with each other.

BitTorrent is one of the most popular and wide-spread use-cases of P2P networks. Under the BitTorrent protocol, a key innovation is to split a given file into many small pieces that peers can obtain and start propagating it amongst

themselves, massively speeding up the process of sharing a large file in comparison to, say, uploading the entire file to one peer at at time.

A central tracker is present to provide peers with information about other peers in the network, such as how to connect to them, what pieces they have, etc.

3 Implementation

3.1 Overview

In our implementation, a single file is represented as a series of binary files, which are distributed across a set of participating peers. We have a script `generator.py` that we used to create such test files; by default, the script generates 10 binary files of size 100 MB each (`piece_0.bin, piece_1.bin,...`), which we used to test, run, and experiment with our code.

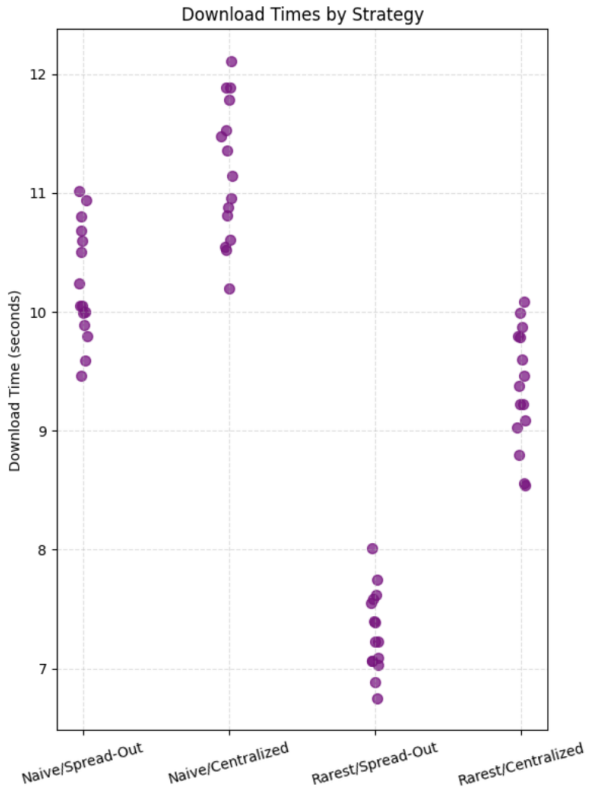
These peers cooperate by exchanging the pieces they possess in order to reconstruct the complete file without any single peer having to provide the full content from the start. Each peer, when launched, scans its local directory to determine which file pieces it already has. It then contacts a central tracker, registering its identity and the pieces it currently holds. The tracker, implemented as a lightweight server, keeps track of all active peers, the pieces they own, and responds to new peers by sending them a list of already known peers and the pieces they have. We represent downloads/uploads between peers as sending the binary files as b-encoded files (an encoding used by BitTorrent) over the wire, where the uploading peer would send over these bits and the downloading peer would write these bits into a new binary file.

Each peer operates in two concurrent roles: as a server and as a client, each in different threads. As a server, the peer listens for incoming connections from other peers and responds to handshake and piece requests. As a client, the peer periodically contacts the tracker to update its view of the network, decides which piece to download next, and then requests that piece from a peer that has it.

In our project, we used two primary strategies for piece selection. We first implemented a naive sequential strategy, where peers download missing pieces in the order of their index, i.e., piece 0, piece 1, piece 2, and so on. While simple, this approach often leads to inefficiencies and bottlenecks when multiple peers simultaneously attempt to download the same few pieces from the same few peers that possess this piece. To address this, we also implemented a second strategy that is present in the BitTorrent architecture: rarest-first piece selection. In rarest-first, peers analyze the current network state and determine which file pieces are least available across all peers. The peer then prioritizes downloading the rarest piece it does not yet have. If multiple pieces are tied as the rarest, one is chosen at random to help distribute requests and avoid

collisions. Similarly, a random peer that possesses this rarest piece is selected to query to also reduce collisions. This strategy improves piece diversity and download parallelism, resulting in faster and more balanced data propagation.

In addition to piece selection algorithms, performance metrics were added to measure the total time required to download a complete file under various conditions. We conducted tests comparing the naive and rarest-first strategies in a centralized scenario (one seeder starts with all data) and spread-out (pieces distributed uniformly among peers, with one copy of each piece in the network to begin with). The graph below indicates a significant performance boost when using the rarest-first algorithm:



We now go over documentation of the functions in each of the following files:

- **tracker.py** – central tracker that manages peer coordination.
- **peer.py** – individual peers that participate in piece sharing and downloading.
- **utils.py** – helper functions for communication and encoding/decoding data.

3.2 tracker.py

Key Data Structures

- `peers`: A dictionary mapping `(ip, port)` to a set of piece indices available at that peer.

Functions

handle_peer(conn, addr) Handles a new peer connection. Supports:

- `announce`: Registers a peer and their pieces.
- `has_piece`: Updates known state that a peer has acquired a new piece.

serve(listen_sock) Main server loop to accept incoming connections from peers.

main() Initializes the tracker and listens on a user-defined host and port.

3.3 peer.py

Global Variables

- `own_pieces`: Set of pieces this peer has locally.
- `known_peers`: List of known peers and what pieces they own.
- `info_hash`: SHA-1 hash identifying the file being distributed.

Functions

handle_peer(conn, addr) Responds to handshake and piece requests. If the requested piece is available, it is sent to the requester.

serve_peers(listen_sock) Listens for incoming peer-to-peer connections and spawns threads to serve them.

connect_tracker(tracker_addr, listen_port) Sends an `announce` to the tracker and updates the list of known peers and their pieces.

notify_tracker_of_piece(tracker_addr, listen_port, piece_index) Notifies the tracker that this peer has newly obtained a piece.

download_loop(tracker_addr, listen_port) Attempts to download pieces until all are acquired. Implements rarest-first piece selection, selecting a random peer among those that have the target piece.

main() Initializes the peer server, connects to the tracker, and starts downloading pieces in a background thread.

3.4 utils.py

Functions

bencode(obj) Encodes a Python object (int, bytes, list, dict) using bencoding.

bdecode(data: bytes) Decodes bencoded data into a Python object.

send(sock, data: bytes) Sends data over a socket with a 4-byte length prefix.

recv_helper(sock, length: int) Ensures exactly `length` bytes are read from the socket.

recv_all(sock) Receives a 4-byte-prefixed message from the socket. Returns the full message or None.

3.5 Example Commands

```
If a folder named peer1 has been created:
    python generator.py
    python tracker.py 127.0.0.1 8000
When inside directory of peer1, run the following:
    python ../peer.py 127.0.0.1 5001 127.0.0.1 8000
```

4 Limitations

4.1 Free-Riders

One important question that still remains is how to ensure that peers will stay behind to seed files for others, or if they will simply leave once they have downloaded their desired file. This is a very real free-rider problem in the real world of torrenting files, and there are a couple ways to tackle this. The main way in the BitTorrent architecture is to implement a mechanism in which peers only upload to other peers they deem worthy—i.e., those who are willing to upload back to them. For peers that are greedy and only leech without seeding in return, they will be *choked*, as in they will not be uploaded to anymore by this peer. This becomes a game theoretic question about how much upload bandwidth should be considered enough to return the favor, and, as all studies of human behavior go, it's kind of unclear. However, one popular strategy is to constantly "optimistically unblock" a random peer, so even if a peer has been choked in the past or isn't being uploaded to currently, there is a possibility that they are actually generous in bandwidth and it is worth it to be optimistic. (This ends up being quite similar to the "best" solutions to similar problems in game theory, where the best strategy is often to lead with kindness, but be firm in reciprocity, such as in the game of a repeated Prisoner's Dilemma.)

4.2 Fault Tolerance

Another important limitation of our current implementation of the BitTorrent architecture is that, due to our centralized tracker, we have reintroduced a single point of failure into our P2P system (given that there is a central tracker, this is more a hybrid P2P system). We discuss the solution to this issue in the following section.

5 Kademlia: Distributed Hash Table

BitTorrent leverages Kademlia's DHT so that each client can discover peers without ever querying a single, central tracker. Instead, torrent info-hashes (the SHA-1 of the torrent's metainfo) serve as DHT keys, and clients perform FIND_NODE/FIND_VALUE lookups to locate peers announcing that hash. Because every node participates in storing and returning contact lists in k-buckets, the network remains resilient to individual failures or censorship. This decentralized lookup means anyone can join or leave without disrupting peer discovery, and there's no single point of trust or attack.

5.1 Identifier Space and XOR Metric

Kademlia assigns every node and every data key a uniformly distributed 160-bit identifier (ID). Node IDs are typically generated by hashing some unique value (e.g., the node's IP address and port or a public key) with SHA-1; keys are hashed in the same way. The distance between two IDs x and y is defined as:

$$\mathrm{dist}(x, y) = x \oplus y,$$

interpreted as an integer. This XOR metric is symmetric, satisfies the triangle inequality, and distributes distances uniformly over the ID space. Because IDs are random, the network approximates a balanced binary tree over the 2^{160} possible IDs, enabling lookups in $O(\log N)$ steps.

5.2 k-Buckets and Routing Tables

Each node maintains a routing table composed of up to 160 k-buckets, one for each possible prefix length from 0 to 159. A k-bucket at index i holds peers whose IDs share the first i bits with the local node but differ in the $(i+1)^{\text{th}}$ bit, and can store up to K contacts (commonly $K = 20$). Buckets are ordered by recency: new contacts append to the tail, and when a bucket is full, the least-recently-seen contact at the head is pinged. If it responds, it moves to the tail and the newcomer is discarded; otherwise, the dead contact is evicted and the new one is added. This eviction policy favors stable peers and keeps routing state fresh.

5.3 Core RPCs

Kademlia defines four lightweight Remote Procedure Calls—PING/PONG, STORE, FIND_NODE, and FIND_VALUE—typically sent over UDP. PING/PONG is a simple liveness check in which nodes periodically ping contacts (especially when evicting) and expect a PONG reply within a timeout. STORE instructs a peer to store a key–value pair locally under the hash of the key. FIND_NODE requests the K contacts in the receiver's routing table closest to a given target ID. FIND_VALUE returns the value if the receiver has the requested key locally; otherwise, it returns the K closest contacts as in FIND_NODE. Every RPC carries the sender's node ID so that recipients can immediately update their routing tables with fresh contact information.

5.4 Iterative Lookup and Concurrency

Kademlia locates a node or retrieves a value through an iterative lookup. It begins by selecting the α closest contacts to the target ID (commonly $\alpha = 3$) and sending FIND_NODE or FIND_VALUE requests to them in parallel. Each

reply yields either the desired value—ending the search—or up to K contacts closer to the target, which are added to the shortlist and inserted into the routing table. The node then repeats this process by querying the next set of up to α unqueried contacts, continuing until no respondent returns contacts closer than those already known. Because each round typically halves the XOR distance to the target, lookups converge in $O(\log N / \log \alpha)$ network round trips.

5.5 Join / Bootstrap Procedure

When a new node with an empty routing table joins the network, it must bootstrap by contacting known peers. The node first generates its own 160-bit ID and sends FIND_NODE messages targeting that ID to well-known bootstrap nodes, inserting any responsive contacts into its k-buckets. It then performs an iterative lookup on its own ID to discover and insert the K closest existing nodes. After this process, the node either has a populated routing table and can fully participate in the network, or, if no bootstrap node responds, it starts alone with an empty table or may retry later.

5.6 Storing and Retrieving Data

To store a key–value pair, the node hashes the key to a 160-bit keyID, performs an iterative lookup for that keyID to find the K closest nodes, and sends STORE(key, value) to each of those nodes, optionally storing locally if it is among the closest. Retrieval works similarly: the node hashes the key to keyID and initiates a FIND_VALUE lookup; if any node along the path has the value, it returns it immediately, otherwise the lookup yields the K closest contacts that can be queried again or cached for future attempts.

5.7 Maintenance and Dynamics

Nodes keep the network healthy by refreshing buckets that have seen no activity through lookups for random IDs within each bucket's range, replicating popular values periodically by reissuing STORE requests, and detecting failed contacts via PING timeouts—evicting unresponsive peers in favor of live ones. Redundant storage across multiple nodes (K replicas) and parallel lookups ensure robustness against churn.

5.8 Results

Unfortunately, we were unable to get all parts to work, but we have a draft of this DHT protocol that can be seen in `DHT.py`. In particular, we discovered that it's actually quite hard to get the nodes in the network to find each other, and so every node would just assume it was the first node in the network and start

off its own hash table. With more time, we would be able to refactor the code to ensure proper finding of other nodes in the network. However, since we are coming up on the deadline for this project, we decided to ditch this attempt. This isn't too disappointing, since the trackerless version of BitTorrent using a distributed hash table was only introduced later on and so this was never a "required" part of the project for us to complete.

6 Concluding Thoughts and Future Work

We wanted to try a project related to BitTorrent since both of us have found the architecture interesting and found relevance for it in other classes we have taken. Implementing our simplified model of it gave us much a deeper appreciation of how the protocol actually works.

A major takeaway from the project was the importance of concurrency in distributed systems. Peers had to simultaneously serve incoming requests while making outgoing requests for missing pieces, which we accomplished via multithreading. Though we could have also simply set the peer command loop to cycle between uploading and downloading, we valued the concurrency, especially given that this is something that peers would likely be doing in an actual P2P filesharing network. Another important takeaway was the impact of algorithm design on system efficiency. Simply altering the piece selection logic led to significant improvements in performance.

Finally, the project highlighted the complexity of real-world file sharing systems like BitTorrent. Although the implementation was intentionally simplified, replicating just a subset of BitTorrent's functionality was a nontrivial task. This is especially relevant regarding our final efforts with Kademlia's DHT.

There are many more complex additions we could make to better improve our codebase. Another key part of the BitTorrent architecture that we never implemented was the BitTorrent-style "choke" and "unchoke" logic, which would address the free-rider problem detailed above. Other potential future work includes a graphical user interface, network visualizations that show which peers are connected and which pieces they have, and how pieces flow through the network in real-time. Finally, the system could be extended to support multi-file torrents, metadata distribution via `.torrent` files, and even the simulation of peer churn (peers joining and leaving) to examine the resilience of the network.