CS3530 : COMPUTER NETWORKS

FINAL PROJECT

ZenTorrent

a P2P based bittorrent system

Yagnesh CS21BTECH11003	Sai Vamsi CS21BTECH11038	Asli Nitej CS21BTECH11011
Srinith CS21BTECH11015	Pranav Varma CS21BTECH11044	Anudeep CS21BTECH11043

Contents

1 Read Me :

- Please install these python packages and run on linux

```
pip install -r requirements.txt
```

- To download a torrent, run the following

```
python main.py nameoffile.torrent
```

- Uncomment lines 63,64 in main.py to see peer availability per piece like this while downloading

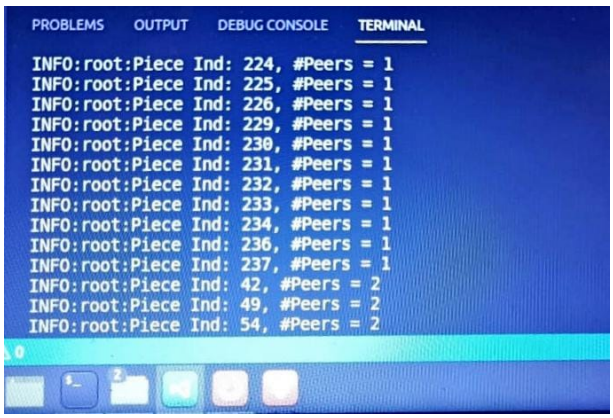

Figure 1: peer availability

- Without them the progress of download will be

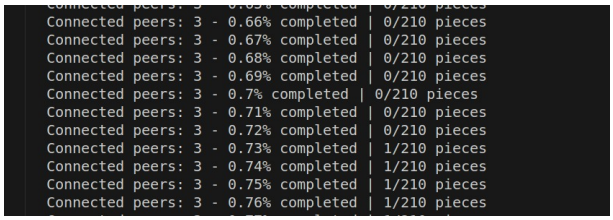

Figure 2: file downloading

2 Learning's

- We now understand the BitTorrent protocol quite well. We've grasped the fundamentals, such as peers, seeds, trackers, and leechers, and how they collaborate to enable fast and efficient downloading of large files.
- A tracker server keeps a list of IPs that either have the file or are in the process of downloading it, just like us. When someone wants to share a file, they upload its details to a tracker server. In return, the tracker server provides a torrent file. This torrent file includes information about the file, such as its name, size, directory paths, a list of tracker servers with information about the file, and a key for unique identification on the tracker server.

- Extracting it and sending a request to the trackers in the file gives us the list of ip's. Now we can send requests to each of these ip's and get parts of file from them.
- How we choose those files and peers has an impact the speed and correctness of our download.
- As this process is dynamic, we can't always guarantee that the entire file is available for download at any given time. When we initiate a download, we start by downloading the rarest piece. This strategy increases the likelihood of the complete file being available on the internet.
- If we have large number of peers with the file this will **not be effective**, but when we have low peers and seeds this works effectively.
- Since we cannot completely implement the bittorrent client application, as we are unfamiliar with the handling of multiple files(file management) , handling of multiple sockets , decoding of torrent file etc. we used code from github and tried to understand it so that we can modify it and implement rarest-piece selection algorithm
- While understanding the code we came across file-management (when we are downloading), different types of bittorent messages and how to parse them.
- There's a specific type of message used to request a peer or seed to send the bitfield, indicating the availability of particular pieces. This allows us to iterate over each piece and select the one with the fewest available peers.
- This torrent does not include uploading file we have downloaded to other peers, so there is high chance someone might choke us.

3 Difficulties :

- Initially we thought of implementing a working bittorrent with rarest-piece selection and choking-unchoking algorithm. but we could only implement rarest-piece selection algorithm.
- We couldn't find much about bit-torrent protocol and usage. We have tried many git repositories but were struck with dependency and version deprecation issues. Also we could not find much on bittorrent simulation.
- So, We implemented the rarest piece selection algorithm on an existing bitorrent client repository.
- We initially thought of just sorting the pieces based on availability but the same piece might be requested to download again, if we do this. So, we had to move in the order of availability.

4 Introduction :

This report will has all the detailed analysis of BitTorrent and its working along with requried files, its attributes and methods.

5 Overview :

5.1 Download Client: a peer looking for a file

- The user or client would begin the application and inputs the name of the file he wants and the process begins.
- Firstly, the app looks for the matching ".torrent" file in a set directory based on the given file name.
- When the .torrent file is located, it is analysed so that we can extract metadata, including tracker information and file structure of the file that the user is searching for.

– Once we have the tracker information, we connect with it and receive a list of peers who are sharing the same file.
– From this list, it selects some peers by default (8 connections are required to start download) to connect with by sending a 'message,' establishing individual peer-to-peer connections.

After the establishment of the connection, the app starts downloading the file in parts known as pieces. It can simultaneously fetch multiple pieces by connecting to several peers willing to share. Our torrent uses the $rarest\_piece\_first$ algorithm to decide the order in which pieces should be downloaded first.

When the download is finished, the app puts together and verifies all the pieces. Depending on the verification outcome, appropriate actions are taken. Once the download and file verification are complete, the application switches to seeding mode, uploading file pieces to other peers.

6 What we Achieved:

– This the evidence for working of rarest-piece selection algorithm from `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4197948&tag=1`

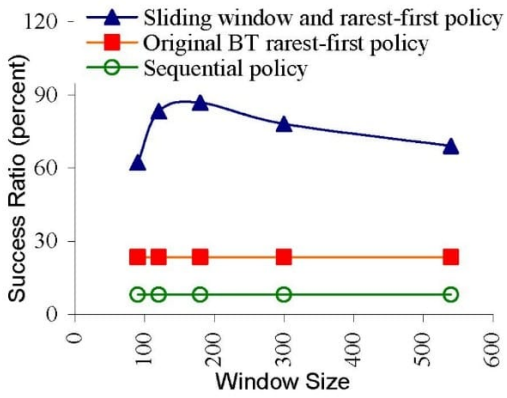

TABLE 1. Success ratio for different chunk selection policies for video consumption rate 4 Mbps and a playback delay of 60 s.

Chunk Selection Policy	*Success Ratio (percent)*
Rarest-first policy (original BT)	23.6%
Sequential policy	8.1%
Sliding window and rarest-first policy	83.3%

– We an see that the rarest first algorithm works better than sequential download policy.
– We were able to pick piece which is rarest among the swarm and make it download first from external tracker System.
– We were able to display the piece's that are rare and list them finally use them first in downloading process

Out Entire Work and effort was huge occupied by learning and details of Implementation of P2P BitTorrent system, which itself is a huge subject to learn study and understand.
we've provided our Understanding of the project down below that includes in depth analysis and use case for working of the program.

7 Deeper Understanding of the Components:

7.1 List of all Files:

- tracker.py
- SAMPLE.torrent
- torrent.py
- pieces_manager.py
- piece.py
- message.py
- peer.py
- peers_manager.py
- rarest_piece.py
- block.py
- main.py

7.2 Working of these components:

7.2.1 main.py:

This component serves as the entry point of our torrent application. It defines a class 'Run' which is responsible for the overall functionality of the client.

- **percentage_completed:** Tracks the download progress.
- **last_log_line:** Stores the last log message for comparison and update purposes.
- Handles command-line input to get the torrent file.
- **self.torrent:** Instance of Torrent from torrent.py, loaded with the torrent file.
- **self.tracker:** Instance of Tracker from tracker.py, initialized with the Torrent instance.
- **self.pieces_manager:** Manages the pieces of the torrent, from pieces_manager.py.
- **self.peers_manager:** Manages the peers in the network, from peers_manager.py.
- the peers manager and logs the start of both the PeersManager and PiecesManager.

  The main loop of the application operates continuously until all pieces are completed. Within this loop, the application fetches peers from the tracker, incorporating them into the peers manager. Continuously, the program requests and downloads pieces from these peers, coordinating the interaction by invoking methods from the peers_manager, pieces_manager, and message modules. Throughout this process, the application provides a visual representation of the download progression. The loop concludes when the entire download is complete, leading to the application's

7.2.2 tracker.py:

This component is used to fetch the list of peers that have a particular file. This file plays a key role in connecting the torrent client to the network by obtaining and managing peer information from trackers.

- **torrent**: Reference to a Torrent object (defined in torrent.py).
- **threads_lis**t: Stores threads, though not utilized in the visible code.

- Handles command-line input to get the torrent file.
- **connected_peers**: Dictionary of peers successfully connected.
- **dict_sock_addr**: Dictionary of socket addresses from the tracker.

  The get_peers_from_trackers function iterates through the tracker URLs listed in the torrent. It dynamically determines whether to call http_scraper or udp_scrapper based on the URL scheme and proceeds to establish connections with the identified peers.

  The try_peer_connect function is responsible for attempting connections to peers, utilizing the Peer class from peer.py. Successful connections are recorded in the connected_peers collection.

  The http_scraper function focuses on scraping HTTP trackers for peers. It decodes responses and populates the dict_sock_addr dictionary with the gathered information.

  Lastly, the send_message function plays a role in facilitating communication with the tracker. It is responsible for sending messages and receiving corresponding responses, with help of methods from PeersManager to ensure coordination between the application and the tracker.

7.2.3 torrent.py:

This file is crucial for understanding and managing the data in a torrent file, such as file names, sizes, piece hashes, and tracker URLs. It serves as the foundation for how the torrent client interacts with the torrent network.

- **torrent_file:** Dictionary to store the parsed torrent file contents.
- **total_length:** Total size of the files to be downloaded.
- **piece_length:** Size of each piece in the torrent.
- **pieces:** Binary string containing the hash of each piece.
- **info_hash:** SHA1 hash of the torrent's info dictionary.
- **peer_id:** Unique ID for the peer in the network.
- **announce_list:** List of tracker URLs.
- **file_names:** List of file names and their lengths in the torrent.
- **number_of_pieces:** Total number of pieces in the torrent.

Method: load_from_path

The load_from_path method begins by opening and reading the torrent file from a specified path, utilizing the `bdecode` functionality. It proceeds to set class attributes based on the contents of the torrent file, generating a unique peer ID through the `generate_peer_id` method. The method then retrieves tracker information with `get_trackers`, initializes file information using `init_files`, calculates the number of pieces, and incorporates assertions to ensure the validity of the torrent file.

Method: get_trackers

The `get_trackers` method focuses on retrieving the list of trackers from the torrent file.

Method: `generate_peer_id`

The `generate_peer_id` method generates a unique peer ID by creating a SHA1 hash of the current time.

7.2.4 pieces_manager.py:

The pieces_manager.py file is responsible for managing the pieces of a torrent file during the download process. It involves handling the data of each piece, tracking their completion, and organizing the blocks within each piece

- **torrent:** Reference to a Torrent object (from torrent.py).
- **number_of_pieces:** Total number of pieces in the torrent.
- **bitfield:** A `bitstring.BitArray` representing the completion status of each piece.
- **pieces:** List of Piece objects representing individual pieces.
- **files:** List of file data related to the pieces.
- **complete_pieces:** Number of completely downloaded pieces.

Method: `update_bitfield`

The `update_bitfield` method serves the purpose of marking a piece as completed in the bitfield.

Method: `receive_block_piece`

The `receive_block_piece` method is responsible for processing a received block of a piece. It updates the corresponding `Piece` object with received data and checks if the piece is complete, subsequently updating the `complete_pieces` count.

Method: `get_block`

The `get_block` method is designed to retrieve a specific block from a piece if it is available.

Method: `all_pieces_completed`

The `all_pieces_completed` method checks whether all pieces have been completely downloaded.

Method: `_generate_pieces`

The `_generate_pieces` method performs the task of generating `Piece` objects for each piece in the torrent. It calculates the length and hash of each piece.

Method: `_load_files`

The `_load_files` method plays a crucial role in preparing file data for each piece. It associates file information with pieces during the process.

7.2.5 piece.py:

This file manages individual pieces of a torrent file, it involves managing blocks within each piece and verifying the integrity of the data.

- **piece_index, piece_size, piece_hash:** Identifiers and hash for the piece.
- **is_full:** Indicates if the piece is fully downloaded.
- **files:** List of files associated with this piece.
- **raw_data:** Byte data of the piece.
- **number_of_blocks:** Calculated number of blocks in the piece.
- **blocks:** List of Block objects representing the blocks of the piece.

Method: update_block_status

The update_block_status method is responsible for updating the status of blocks that have been pending for too long.

Method: set_block

The set_block method sets the data for a specific block within the piece.

Method: get_block

The get_block method retrieves a specified block of data from the piece.

Method: get_empty_block

The get_empty_block method finds and returns the next empty block to be requested.

Method: are_all_blocks_full

The are_all_blocks_full method checks if all blocks in the piece are fully downloaded.

Method: set_to_full

The set_to_full method merges blocks and sets the piece as full if the data is valid. It also sends a message using pubsub to indicate the piece completion.

Method: _init_blocks

The _init_blocks method initializes the blocks for the piece.

Method: _write_piece_on_disk

The _write_piece_on_disk method writes the downloaded piece data to disk.

Method: _merge_blocks

The _merge_blocks method merges the blocks into a single contiguous piece.

Method: _valid_blocks

The _valid_blocks method validates the downloaded piece data against its hash.

7.2.6 message.py:

It handles the creation, parsing, and management of various messages used in peer-to-peer communication. It defines multiple classes for different types of messages exchanged between peers and trackers
Largely Handles all the requests and responses between seed and peer. and is Very Huge

7.2.7 peer.py:

It handles the connections, communication, state management, and message processing for each peer in the network.
The `Peer` class encompasses various attributes crucial for managing connection status, networking details, and state in peer-to-peer communication. These attributes include `last_call`, `has_handshaked`, `healthy`, `read_buffer`, `socket`, `ip`, `port`, `number_of_pieces`, `bit_field`, and `state`. Each of these plays a distinct role in facilitating and maintaining the connection and communication between peers in the network.

Method: __hash__

Returns a unique hash for the peer, useful for managing peer sets or dictionaries.

Method: connect

Attempts to establish a socket connection to the peer.

Method: send_to_peer

Sends a message to the peer.

Method: is_eligible

Determines if the peer is eligible for a new message request.

Method: has_piece

Checks if the peer has a specific piece.

Method: (am_choking, am_unchoking, is_choking, is_unchoked, is_interested, am_interested)

Manage and query the state of the peer regarding choking and interest.

handle_choke, handle_unchoke

handle_interested, handle_not_interested

handle_have, handle_bitfield, handle_request, handle_piece, handle_cancel, handle_port_request

Methods to handle various BitTorrent protocol messages.

_handle_handshake, _handle_keep_alive

Handle handshake and keep-alive messages.

Method: get_messages

Processes incoming messages from the peer.

7.2.8 peers_manager.py

It is responsible for managing the interactions with multiple peers in the torrent network. It handles connecting to peers, maintaining the peer list, and managing the communication with each peer. This file is a central component for managing peer-to-peer interactions, ensuring efficient communication and data exchange in the torrent network.

- **peers:** List of connected peers.
- **torrent:** Reference to the Torrent object (from torrent.py).
- **pieces_manager:** Reference to the PiecesManager object (from pieces_manager.py).
- **rarest_pieces:** Instance of RarestPieces (from rarest_piece.py) to manage piece availability.
- **pieces_by_peer:** Tracks pieces availability by peer.
- **is_active:** Indicates if the manager is actively managing peers.

.

Method: `get_random_peer_having_piece`

The `get_random_peer_having_piece` method returns a random peer that has the specified piece.

Method: `has_unchoked_peers` and `unchoked_peers_count`

The manager provides methods `has_unchoked_peers` and `unchoked_peers_count` for checking unchoked peers.

Method: `add_peers`

The `add_peers` method adds new peers to the manager.

Method: `remove_peer`

The `remove_peer` method removes a peer from the manager.

Method: `_process_new_message`

The `_process_new_message` method processes incoming messages from peers.

Method: `_do_handshake`

Performs a handshake with a new peer.

7.2.9 block.py

Blocks are the smaller segments of data that make up a torrent piece. Each piece is composed of multiple blocks, and this file defines how these blocks are managed.

- **FREE:** The block has not been requested yet.
- **PENDING:** The block has been requested but not received yet.
- **FULL:** The block has been successfully downloaded.

- **state:** A State enum value representing the current state of the block.
- **block_size:** The size of the block in bytes.
- **data:** The actual data of the block, stored as bytes.
- **last_seen:** A timestamp indicating the last time this block was interacted with.