Project Report

-supervised by:

Dr. Ratan K. Ghosh

Dr. Subhajit Sidhanta

-submitted by:

Rohit Das (11910230), and

Shoumik Ghosal (11910300)

Indian Institute of Technology, Bhilai

CS525

March 2020

Contents

Declaration **3**

ABSTRACT **4**

CHAPTER 1: INTRODUCTION **5**
1.1 P2P File-Sharing: 5
1.1.1 What is P2P File-Sharing? 5
1.1.2 Types of P2P Architecture: 5
1.1.2.1 Centralized Directory: 5
1.1.2.2 Query Flooding: 6
1.1.2.3 Exploiting Heterogeneity: 6
1.2 Distributed Hash Tables (DHTs): 7

CHAPTER 2: LITERATURE SURVEY **8**
2.1 BitTorrent Protocol: 8
2.1.1 Bencoding: 8
2.1.2 Peer Protocol: 8
2.2 Kademlia and the XOR Metric: 9
2.2.1 System Description: 10
2.2.2 XOR Metric: 10

CHAPTER 3: METHODS DEVELOPED **12**
3.1 Bencode parsing: 12
3.2 Blocks and Pieces Management: 14

CHAPTER 4: RESULTS AND CONCLUSIONS **22**

CHAPTER 5: FUTURE SCOPE **23**

CHAPTER 6: REFERENCES **24**

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/-source our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Rohit Das
(11910230)

Shoumik Ghoshal
(11910300)

ABSTRACT

P2P file sharing is the process of sharing digital content like electronic books, multimedia (music, movies, video clips) through a direct connection between two peers or nodes over the P2P network. This is done with the help of a P2P client software which allows a computer to get connected to a P2P network. The peer which hosts the file is called a seed and the one which downloads it is called a leech. With the introduction of the BitTorrent[1] protocol, files can be easily shared between peers via seeders, without overloading a single file server. DHT, or distributed hash tables, are a unique way of keeping track of file-owners, or their neighbours. Kademlia[2] is a routing algorithm which optimizes this search for the nearest neighbour using DHTs.

Here we present a peer-to-peer file-sharing system using DHTs instead of trackers,i.e. servers where information of nodes, seeders, etc. are stored. Kademlia is used to optimize the search in DHTs for nodes having the full or partial file(s), and efficiently get the IP and Port of such nodes.

CHAPTER 1: INTRODUCTION

1.1 P2P File-Sharing:

1.1.1 What is P2P File-Sharing?

Peer-to-peer[3] file sharing is the distribution and sharing of digital media using peer-to-peer (P2P) networking technology. P2P file sharing allows users to access media files such as books, music, movies, and games using a P2P software program that searches for other connected computers on a P2P network to locate the desired content.

A peer-to-peer network allows computer hardware and software to communicate without the need for a server. Unlike client-server architecture, there is no central server for processing requests in a P2P architecture. The peers directly interact with one another without the requirement of a central server.

Now, when one peer makes a request, it is possible that multiple peers have the copy of that requested object. Now the problem is how to get the IP addresses of all those peers. This is decided by the underlying architecture supported by the P2P systems. By means of one of these methods, the client peer can get to know about all the peers which have the requested object/file and the file transfer takes place directly between these two peers.

1.1.2 Types of P2P Architecture:

There are roughly three such architectures:

- Centralized Directory
- Query Flooding
- Exploiting Heterogeneity

1.1.2.1 Centralized Directory: It maintains a huge central server to provide directory service. All the peers inform this central server of their IP address and the files they are making available for sharing. The server queries the peers at regular intervals to make sure if the peers are still connected or not. This server maintains a huge database regarding which file is present at which IP addresses.

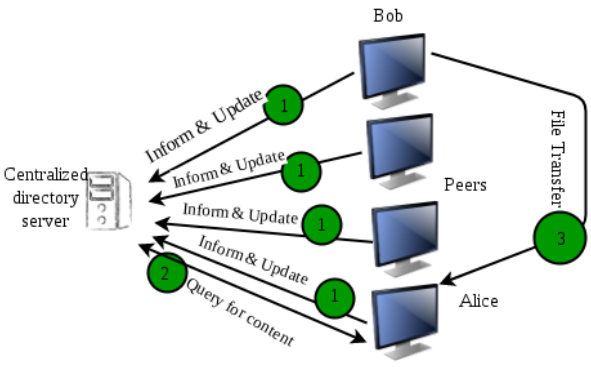

Figure 1: Centralized Directory

Now whenever a requesting peer comes in, it sends its query to the server. Since the server has all the information of its peers, so it returns the IP addresses

of all the peers having the requested file to the peer. Now the file transfer takes place between these two peers.

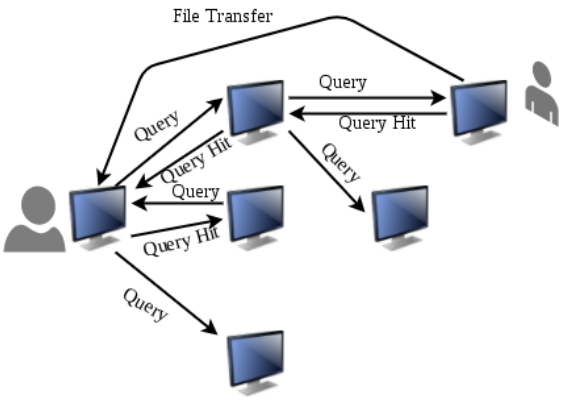

Figure 2: Query Flooding

1.1.2.2 Query Flooding: Unlike the centralized approach, this method makes use of distributed systems. In this, the peers are supposed to be connected into an overlay network. It means if a connection/path exists from one peer to other, it is a part of this overlay network. In this overlay network, peers are called as nodes and the connection between peers is called an edge between the nodes, thus resulting in a graph-like structure.

Now when one peer requests for some file, this request is sent to all its neighboring nodes i.e. to all nodes which are connected to this node. If those nodes dont have the required file, they pass on the query to their neighbors and so on. This is called as query flooding. When the peer with requested file is found (referred to as query hit), the query flooding stops and it sends back the file name and file size to the client, thus following the reverse path. If there are multiple query hits, the client selects from one of these peers.

1.1.2.3 Exploiting Heterogeneity: This P2P architecture makes use of both the above discussed systems. It resembles a distributed system like Gnutella because there is no central server for query processing. But it does not treat all its peers equally. The peers with higher bandwidth and network connectivity are at a higher priority, called as group leaders/super nodes. The rest of the peers are assigned to these super nodes. These super nodes are interconnected and the peers under these super nodes inform their respective leaders about their connectivity, IP address and the files available for sharing.

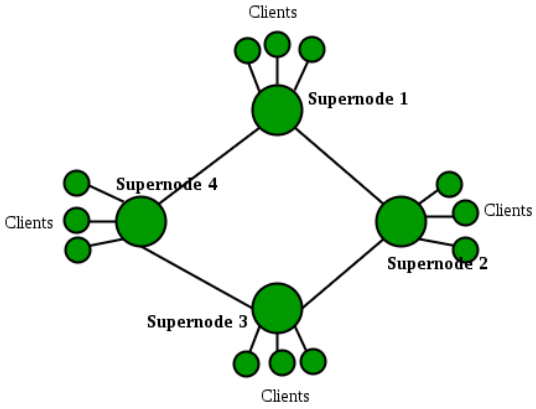

Figure 3: Supernodes

This structure can process the queries in two ways. The first one is that the super nodes could contact other super nodes and merge their databases with its own database. Thus, this super node now has information of a large number

of peers. Another approach is that when a query comes in, it is forwarded to the neighboring super nodes until a match is found. Thus query flooding exists but with limited scope as each super node has many child peers. Hence, such a system exploits the heterogeneity of the peers by designating some of them as group leaders/super nodes and others as their child peers.

1.2 Distributed Hash Tables (DHTs):

A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table: ⟨key, value⟩ pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Nodes can be added/removed with minimum work around re-distributing keys. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs characteristically emphasize the following properties:

- Autonomy and decentralization: the nodes collectively form the system without any central coordination.
- Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- Scalability: the system should function efficiently even with thousands or millions of nodes.

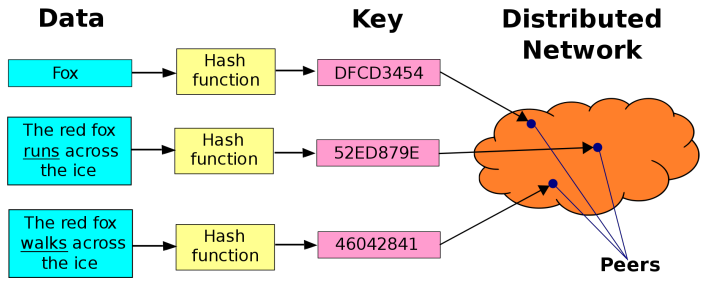

Figure 4: Distributed Hash Tables

Any one node needs to coordinate with only a few other nodes in the system – most commonly, $O(logn)$ of the n participants – so that only a limited amount of work needs to be done for each change in membership.

Some DHT designs seek to be secure against malicious participants and to allow participants to remain anonymous, though this is less common than in many other peer-to-peer (especially file sharing) systems.

Finally, DHTs must deal with more traditional distributed systems issues such as load balancing, data integrity, and performance (in particular, ensuring that operations such as routing and data storage or retrieval complete quickly).

CHAPTER 2: LITERATURE SURVEY

2.1 BitTorrent Protocol:

BitTorrent is a protocol for distributing files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over plain HTTP is that when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load. A BitTorrent file distribution consists of the following entites:

- An ordinary web server
- A static 'metainfo' server
- A BitTorrent tracker
- An 'original' downloader
- The end user downloaders

2.1.1 Bencoding:

Bencoding is the encoding used to store metadata in a .torrent file. Strings are length-prefixed base ten followed by a colon and the string. For example `4:spam` corresponds to 'spam'. Integers are represented by an 'i' followed by the number in base 10 followed by an 'e'. For example `i3e` corresponds to 3 and `i-3e` corresponds to -3. Integers have no size limitation. `i-0e` is invalid. All encodings with a leading zero, such as `i03e`, are invalid, other than `i0e`, which of course corresponds to 0.

Lists are encoded as an 'l' followed by their elements (also bencoded) followed by an 'e'. For example `l4:spam4:eggse` corresponds to ['spam', 'eggs']. Dictionaries are encoded as a 'd' followed by a list of alternating keys and their corresponding values followed by an 'e'. For example, `d3:cow3:moo4:spam4:eggse` corresponds to {'cow': 'moo', 'spam': 'eggs'} and `d4:spaml1:a1:bee` corresponds to {'spam': ['a', 'b']}. Keys must be strings and appear in sorted order (sorted as raw strings, not alphanumerics).

2.1.2 Peer Protocol:

BitTorrent's peer protocol operates over TCP or uTP. Peer connections are symmetrical. Messages sent in both directions look the same, and data can flow in either direction. The peer protocol refers to pieces of the file by index as described in the metainfo file, starting at zero. When a peer finishes downloading a piece and checks that the hash matches, it announces that it has that piece to all of its peers. Connections contain two bits of state on either end: choked or not,

and interested or not. Choking is a notification that no data will be sent until unchoking happens.

Data transfer takes place whenever one side is interested and the other side is not choking. Interest state must be kept up to date at all times - whenever a downloader doesn't have something they currently would ask a peer for in unchoked, they must express lack of interest, despite being choked. This makes it possible for downloaders to know which peers will start downloading immediately if unchoked. Connections start out choked and not interested. When data is being transferred, downloaders should keep several piece requests queued up at once in order to get good TCP performance.

On the other side, requests which can't be written out to the TCP buffer are immediately queued up in memory rather than kept in an application-level network buffer, so they can all be thrown out when a choke happens. The peer wire protocol consists of a handshake followed by a never-ending stream of length-prefixed messages. The handshake starts with character ninteen (decimal) followed by the string 'BitTorrent protocol'. The leading character is a length prefix, put there in the hope that other new protocols may do the same and thus be trivially distinguishable from each other. All later integers sent in the protocol are encoded as four bytes big-endian.

After the fixed headers come eight reserved bytes, which are all zero in all current implementations. If you wish to extend the protocol using these bytes, please coordinate with Bram Cohen to make sure all extensions are done compatibly. Next comes the 20 byte sha1 hash of the bencoded form of the info value from the metainfo file. (This is the same value which is announced as info_hash to the tracker, only here it's raw instead of quoted here). If both sides don't send the same value, they sever the connection. After the download hash comes the 20-byte peer id which is reported in tracker requests and contained in peer lists in tracker responses. If the receiving side's peer id doesn't match the one the initiating side expects, it severs the connection.

2.2 Kademlia and the XOR Metric:

Kademlia minimizes the number of configuration messages nodes must send to learn about each other. It uses parallel, asynchronous queries to avoid timeout delays from failed nodes. While Kademlia takes the basic approach of many DHTs, like opaque, 160-bit keys, ⟨key,value⟩ pairs being stored in nodes with IDs "close" to the key, many of its benefits result from its use of a novel XOR metric for distance between points in the key space.

2.2.1 System Description:

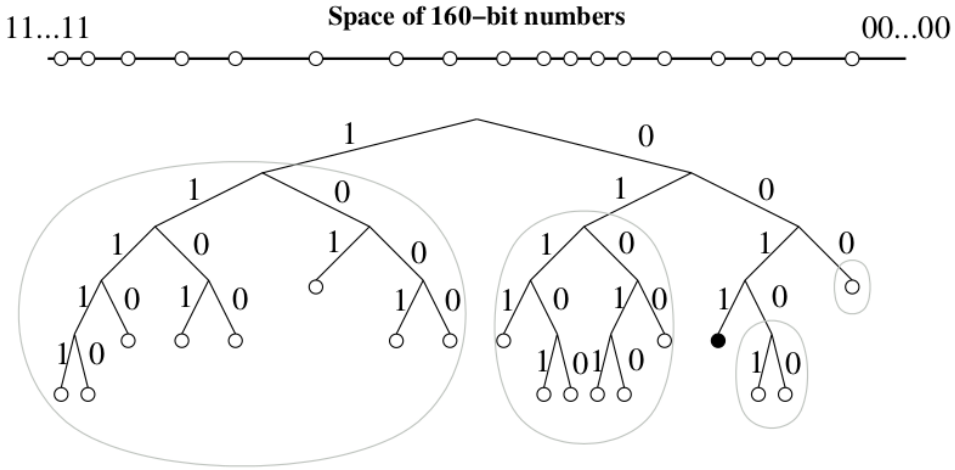

Figure 5: Kademlia binary tree. The black dot shows the location of node 0011.... in the tree.

Kademlia effectively treats its nodes as leaves in a binary tree, with each node's position determined by the shortest unique prefix of its ID. For any given node, the binary tree is divided into a series of successively lower subtrees that don't contain the node. The highest subtree consists of the half of the binary tree not containing the node. The next subtree consists of the half of the remaining subtree not containing the node and so forth.

2.2.2 XOR Metric:

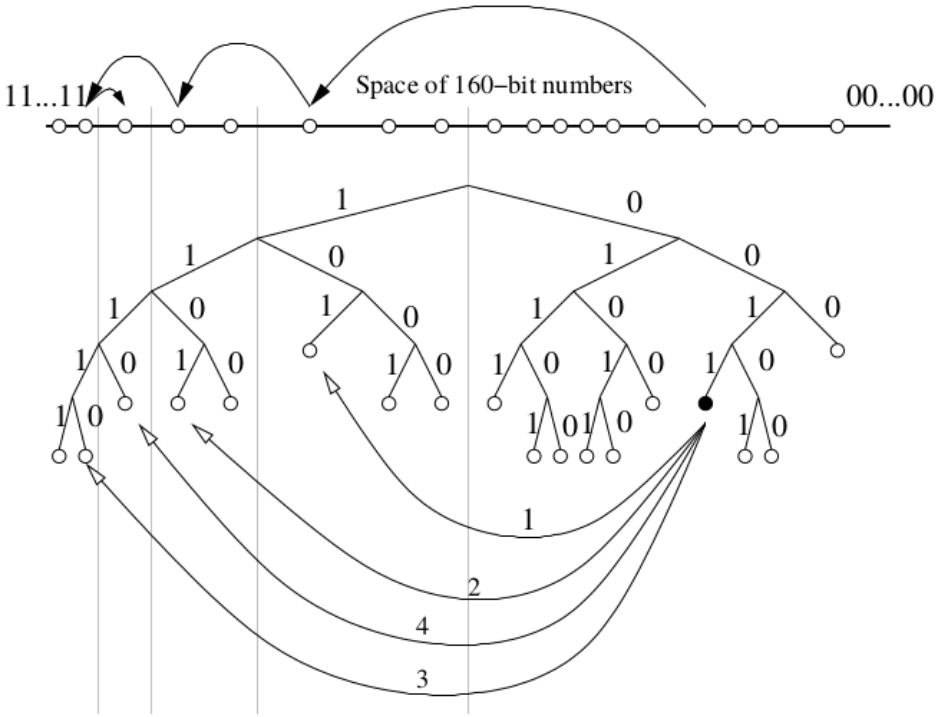

Figure 6: Locating a node 1110 by its ID.

Every message a node transmits includes its node ID, permitting the recipient to record the sender's existence if necessary. Kademlia relies on a notion of distance between two identifiers. Given two identifiers, x and y, Kademlia defines the distance between them as ther bitwise exclusive or (XOR) interpreted as an integer, $d(x, y) = x \oplus y$.

XOR captures the notion of distance implicit in the binary-tree-based sketch of the system. In a fully-populated binary tree of 160 bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing them both. When a tree is not fully populated, the closest leaf to an ID x is the leaf whose ID shares the longest common prefix of x. If there are empty branches in the tree, there might be more than one leaf with the longest common prefix. In that case, the closest leaf to x will be the closest leaf to ID x produced by flipping the bits in x corresponding to the empty branches of the tree.

CHAPTER 3: METHODS DEVELOPED

Here we have presented our implementation of parsing the .torrent file, and piece and block management while fetching and merging bytes of file downloaded. We also check the integrity of the pieces downloaded, block by block. We fixed the block size at 2^{14}. All implementations have been done in python3.

3.1 Bencode parsing:

File: bencoding.py

```
import math, bencode, hashlib, logging, os, time

class TorrentData(object):
    def __init__(self):
        self.torrent_dict = {}
        self.total_length:int = 0
        self.piece_length:int = 0
        self.pieces:int = 0
        self.info_hash:str = ''
        self.peer_id:str = ''
        self.announce_list:str = ''
        self.file_names = []
        self.no_of_pieces:int = 0

    def load_file(self,path):
        # bencode parsing
        with open(path,'rb') as torrentfile:
            parsed_data = bencode.bdecode(torrentfile.read
              ↪ ())

        self.torrent_dict = parsed_data
        self.piece_length = self.torrent_dict['info']['
          ↪ piece length']
        self.pieces = self.torrent_dict['info']['pieces']
        self.info_hash = hashlib.sha1(bencode.encode(self.
          ↪ torrent_dict['info'])).digest()
        self.peer_id = self.generate_peer_id()
        self.announce_list = self.get_trackers()
        self.get_files()
        self.no_of_pieces = math.ceil(self.total_length /
          ↪ self.piece_length)
```

```
        # Printing out tracker list and file names
        logging.debug(self.announce_list)
        logging.debug(self.file_names)

        # Testing to check if total file length and number
          ↪ of files is not 0
        # assert(self.total_length > 0)
        # assert(len(self.file_names) > 0)

        return self


    def get_files(self):
        '''
        Finds the file names and creates the directory
          ↪ structure
        '''
        folder = self.torrent_dict['info']['name']

        if 'files' in self.torrent_dict['info']:
            if not os.path.exists(folder):
                os.mkdir(folder,0o0766) # setting
                  ↪ permission for the folder

            for file in self.torrent_dict['info']['files'
              ↪ ]:
                path_name = os.path.join(folder,*file['
                  ↪ path'])

                if not os.path.exists(os.path.dirname(
                  ↪ path_name)):
                    os.makedirs(os.path.dirname(path_name)
                      ↪ )

                self.file_names.append({"path":path_name,"
                  ↪ length":file['length']})
                self.total_length += file['length']

        else:
```

```
                self.file_names.append({"path":folder, "length
                    ↪ ":self.torrent_dict['info']['length']})
                self.total_length = self.torrent_dict['info'][
                    ↪ 'length']


    def get_trackers(self):
        '''
        Gets the tracker URLs from the .torrent file
        '''
        if 'announce-list' in self.torrent_dict:
            return self.torrent_dict['announce-list']
        else:
            return [[self.torrent_dict['announce']]]


    def generate_peer_id(self):
        '''
        Generates a random Peer-ID with current time as
            ↪ seed
        '''
        seed = str(time.time())
        return hashlib.sha1(seed.encode('utf-8')).digest()

# tor = Torrent()
# tor.load_file("./sintel.torrent")
# print(tor.announce_list)
# print(tor.file_names)
```

3.2 Blocks and Pieces Management:

File: blocks.py

```
from enum import enum

BLOCK_SIZE = 2 ** 14

class Block_State(Enum):
    FREE = 0
    PENDING = 1
    FULL = 2
```

```
class Blocks():
    def __init__(self, state: Block_State = FREE,
      ↪ block_size:int = BLOCK_SIZE,
                 data: bytes = b'',last_seen:float = 0):
        self.state: Block_State = state
        self.block_size:int = block_size
        self.data:bytes = data
        self.last_seen:float = last_seen

    def __str__(self):
        return '%s - %d - %d - %d' %(self.state, self.
          ↪ block_size, len(self.data), self.last_seen)
```

File: pieces.py

```
import hashlib, math, time, logging

from pubsub import pub # creates a publisher-subscriber
  ↪ module for messaging

from block import Blocks,BLOCK_SIZE, Block_State

class DataPieces(object):
    def __init__(self, piece_index:int, piece_size:int,
      ↪ piece_hash:str):
        self.piece_index:int = piece_index
        self.piece_size:int = piece_size
        self.piece_hash:str = piece_hash
        self.is_full:bool = False
        self.files = []
        self.raw_data:bytes = b''
        self.no_of_blocks:int = int(math.ceil(float(
          ↪ piece_size) / BLOCK_SIZE))
        self.blocks:list[Blocks] = []

        self._alloc_blocks();

    def _alloc_blocks(self):
        self.blocks = []
```

```
        if self.no_of_blocks > 1:
            for i in range(self.no_of_blocks):
                self.blocks.append(Blocks())

            # last block may not be divisible by
              ↪ BLOCK_SIZE
            if (self.piece_size % BLOCK_SIZE > 0):
                self.blocks[self.no_of_blocks - 1].
                  ↪ block_size = self.piece_size %
                  ↪ BLOCK_SIZE

        else:
            self.blocks.append(Blocks(block_size = int(
              ↪ self.piece_size))) # only one block in
              ↪ file

    def _save_piece(self): # writes file pieces to disk
        for file in self.files:
            path_file = file["path"]
            file_offset = file["fileOffset"]
            piece_offset = file["pieceOffset"]
            length = file["length"]

            try:
                f = open(path_file, 'r+b') # Already
                  ↪ existing file
            except IOError:
                f = open(path_file, 'wb') # New file
            except Exception:
                logging.exception("Can't write to file")
                return

            f.seek(file_offset)
            f.write(self.raw_data[piece_offset:
              ↪ piece_offset + length])
            f.close()

    def _merge_blocks(self):
        torr_buf = b''
```

```
        for block in self.blocks:
            torr_buf += block.data

        return torr_buf

    def _valid_blocks(self, piece_raw_data):
        hashed_piece_raw_data = hashlib.sha1(
          ↪ piece_raw_data).digest()

        if hashed_piece_raw_data == self.piece_hash:
            return True

        logging.warning("Error Piece Hash")
        logging.debug("{} : {}".format(
          ↪ hashed_piece_raw_data, self.piece_hash))
        return False

    def set_to_full(self):
        data = self._merge_blocks() # merging all pieces

        if not self._valid_blocks(data): # matching hashes
          ↪  of all pieces against torrent file hash
            self._alloc_blocks()
            return False

        self.is_full = True
        self.raw_data = data
        self._save_piece()
        pub.sendMessage('PiecesMgr.PieceCompleted',
          ↪ piece_index=self.piece_index)

        return True

    def update_block_status(self):
        for i,block in enumerate(self.blocks):
            if block.state == Block_State.PENDING and (
              ↪ time.time() - block.last_seen) > 5:
                self.blocks[i] = Blocks() # free block if
                  ↪ status is pending for 5s
```

```
    def set_block(self,offset,data):
        index = int(offset / BLOCK_SIZE)

        if not self.is_full and not self.blocks[index].
          ↪ state == Block_State.FULL:
            self.blocks[index].data = data
            self.blocks[index].state = Block_State.FULL

    def get_empty_block(self):
        if self.is_full:
            return None

        for block_index, block in enumerate(self.blocks):
            if block.state == Block_State.FREE:
                self.blocks[block_index].state =
                  ↪ Block_State.PENDING
                self.blocks[block_index].last_seen = time.
                  ↪ time()
                return self.piece_index, block_index *
                  ↪ BLOCK_SIZE, block.block_size

        return None

    def are_blocks_full(self):
        for block in self.blocks:
            if block.state == Block_State.FREE or block.
              ↪ state == Block_State.PENDING:
                return False

        return True
```

File: piece_mgr.py

```
import piece
import bitstring, logging
from pubsub import pub

class PiecesMgr(object):
    def __init__(self, torrent):
```

```
        self.torrent = torrent
        self.no_of_pieces = int(torrent.no_of_pieces)
        self.bitfield = bitstring.BitArray(self.
          ↪ no_of_pieces)
        self.pieces = self._gen_pieces()
        self.files = self._load_files()
        self.complete_pieces = 0

        for file in self.files:
            id_piece = file['idPiece']
            self.pieces[id_piece].files.append(file)

        # events
        pub.subscribe(self.rcv_block_piece, 'PiecesMgr.
          ↪ Piece')
        pub.subscribe(self.update_bitfield, 'PiecesMgr.
          ↪ PieceCompleted')

    def update_bitfield(self, piece_index):
        self.bitfield[piece_index] = 1

    def _gen_pieces(self):
        pieces = []
        last_piece = self.no_of_pieces - 1

        for i in range(self.no_of_pieces):
            start = i * 20
            end = start + 20

            if i == last_piece:
                piece_len = self.torrent.total_length - (
                  ↪ self.no_of_pieces - 1) * self.torrent
                  ↪ .piece_len
                pieces.append(piece.DataPiece(i, piece_len
                  ↪ , self.torrent.pieces[start:end]))
            else:
                pieces.append(piece.DataPiece(i, self.
                  ↪ torrent.piece_length, self.torrent.
                  ↪ pieces[start:end]))
```

```
        return pieces

    def all_pieces_completed(self):
        for piece in self.pieces:
            if not piece.is_full:
                return False

        return True

    def rcv_block_piece(self, piece):
        piece_index, piece_offset, piece_data = piece

        if self.pieces[piece_index].is_full:
            return

        self.pieces[piece_index].set_block(piece_offset,
          ↪ piece_data)

        if self.pieces[piece_index].are_blocks_full():
            if self.pieces[piece_index].set_to_full():
                self.complete_pieces +=1


    def get_block(self, piece_index, block_offset,
      ↪ block_length):
        for piece in self.pieces:
            if piece_index == piece.piece_index:
                if piece.is_full:
                    return piece.get_block(block_offset,
                      ↪ block_length)
                else:
                    break

        return None

    def _load_files(self):
        files = []
        piece_offset = 0
        piece_size_used = 0
```

```
        for f in self.torrent.file_names:
            current_size_file = f["length"]
            file_offset = 0

            while current_size_file > 0:
                id_piece = int(piece_offset / self.torrent
                  ↪ .piece_length)
                piece_size = self.pieces[id_piece].
                  ↪ piece_size - piece_size_used

                if current_size_file - piece_size < 0:
                    file = {"length": current_size_file,
                            "idPiece": id_piece,
                            "fileOffset": file_offset,
                            "pieceOffset": piece_size_used
                              ↪ ,
                            "path": f["path"]
                            }
                    piece_offset += current_size_file
                    file_offset += current_size_file
                    piece_size_used += current_size_file
                    current_size_file = 0

                else:
                    current_size_file -= piece_size
                    file = {"length": piece_size,
                            "idPiece": id_piece,
                            "fileOffset": file_offset,
                            "pieceOffset": piece_size_used
                              ↪ ,
                            "path": f["path"]
                            }
                    piece_offset += piece_size
                    file_offset += piece_size
                    piece_size_used = 0

                files.append(file)
        return files
```

CHAPTER 4: RESULTS AND CONCLUSIONS

Pending further implementation and metric monitoring.

CHAPTER 5: FUTURE SCOPE

We are yet to implement message passing to indicate choked/unchoked, interested/not interested, and mainly, the DHT.

CHAPTER 6: REFERENCES

[1] Bram Cohen. The BitTorrent Protocol Specification. 2008. `https://www.bittorrent.org/beps/bep_0003.html`.

[2] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, 2002.

[3] James F. Kurose and Keith M. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, Inc., 2013.