BitTorrent using MPI Documentation

Version 1.0.1

Dragomir Andrei-Mihai

January 6, 2024

1 Introduction

1.1 General

BitTorrent is a peer-to-peer protocol for exchanging files over the Internet created by Bram Cohen in 2001. The company responsible for its expansion is BitTorrent Inc. Corporation, founded by Cohen and Ashwin Navin in 2004.

BitTorrent Inc. has built a large ecosystem around the protocol, including a desktop client, a mobile client, and a web client. It is used by hundreds of millions of people around the world.

1.2 Abstract

Bittorent is a **peer to peer** protocol. It enhances transfers of large files and increases the speed of transfers, especially for popular files which can be found on multiple endpoints.

It is a **descentralized** protocol, which means that no main server exists to create and manage the transfers. The coordination unit called tracker manages the connections between peers but it is not directly involved in the file transfers.

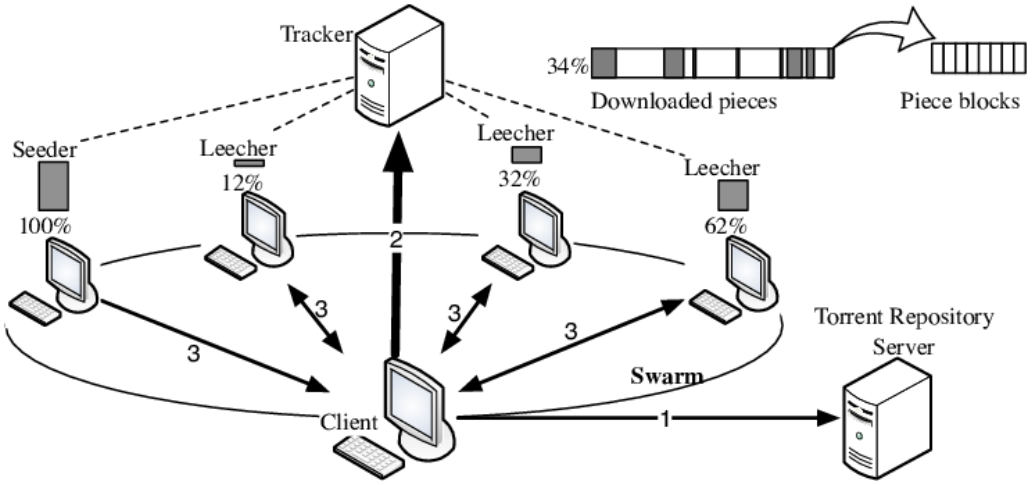
BitTorrent can also share files between **untrusted** parties. The protocol base architecture is a network of computers. That is why it bypasses the problems that a singular server would have, such as the possibility of a server **crash** or **overload**.

This protocol makes use of every endpoint upload and download bandwidth to balance data transfer across all participants. BitTorrent is based on a file sharing system, which means that the **more users** are downloading a file, the **faster** the download will be.

BitTorrent manages files by dividing them into small pieces identified by a **hash**. The hash represents an unique identifier for each piece and can verify the integrity of the pieces.

As an overview, BitTorrent came as a second generation P2P file sharing system, which focuses on organizing clients that are interested in downloading the same file into an **overlay network**, called a Torrent. The best advantage was that it represents a solution to the classic client-server architecture by overcoming the scalability and reliability problems.

2 Protocol roles



2.1 Tracker

The tracker is a **coordination server** that keeps account of the peers participating in the file transfer. It is the central unit of the protocol and it **mediates** the connections between clients.

As presented, it is not involved in the file transfers but knows at all times what peers are participating in the transfer and what pieces of the file they have. It is also used to find new peers that enter the transfer.

The **tracker** knows about each client by memorising a data structure called **swarm**. The swarm of the file contains all the clients that have segments of that specified file and most important what segments they have.

The tracker has support for handling 5 types of scenarios:

- **Request for file**: When a client wants to download a file, it sends a request to the tracker. The tracker responds with a list of peers that have segments of the file. The client then connects to the peers and starts downloading the file.
- **Update from peers**: When a client has downloaded a segment of a file, it sends an update to the tracker. The tracker adds the client to the swarm of the file if not already present. These updates can be done at intervals of 10 downloads for example to not overload the tracker.

- **Finalise download from peers**: When a client has downloaded all segments of a file, it sends a finalise to the tracker. The tracker keeps the client as a seeder for that file.
- **Finalise all downloads from peers**: When a client has downloaded all segments of all files, it sends a finalise to the tracker. The tracker keeps the client as a seeder for all files.
- **Last finalise from peers**: When all the clients connected in that session have sent a finalise, the tracker stops all the seeders and then stops itself.

2.2 Seeder

A seeder is a client that has all the segments of a file and its role is to share the file with other clients. It acts as an **upload server** for the file. A client can become a seeder after downloading all the segments of a file or can start as a seeder from the beginning.

2.3 Peer

A peer is a client that has zero or more segments of a file. It is used to download and upload segments of the file. It acts as a **download** and as an **upload center**. Usually two different threads are used to download and upload segments in order to parallelize the process.

2.4 Leecher

A leecher is a client that starts with **zero segments of any file** and tries to download segments from other peers. It can become a peer after downloading some segments or can directly have segments that it does not share with other peers.

3 Implementation

3.1 Efficiency Considerations

In a well implemented BitTorrent protocol, the download speed increases with the number of peers if only the peers have a good **decision making algorithm** for choosing download endpoint.

The algorithm should take into consideration the **download speed** of the peers, the **number of segments** they own and the **number of pending requests**.

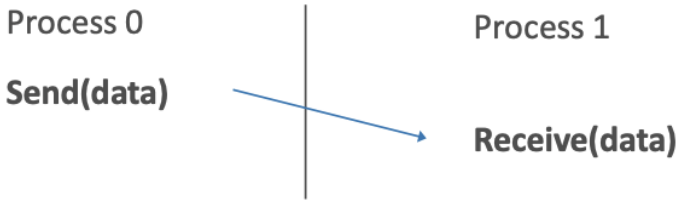
This specific implementation uses the **MPI API** in C++. File transfers are simulated solely by passing the hash of the segment to the peer that requested it. Sending only operates at unit level, meaning that no complex structures are involved in the transfer, just **standard primitives**.

In the following implementation, the algorithm is based on the number of **requests pending**.

3.2 MPI Communication

The MPI communication is spread across different so called channels implemented using MPI tags. The channels are:

- **INIT** - initialisation of the tracker and peers
- **COMMANDS** - communicating actions between peers and tracker
- **UPDATE_COMMAND** - communicating updates between peers and tracker
- **DOWNLOAD_REQUEST** - communicating download requests between peers
- **KILL** - killing the peers
- **WORKLOAD** - communicating the workload between peers to know what peer to download from
- **ACKNOWLEDGEMENT** - communicating the acknowledgement that the segment has been sent.



3.3 Initialisation

3.3.1 Tracker

The tracker waits for all the clients to send the information about the files they have. It responds to each client with an acknowledgement that it has received all the information. The tracker adds each client to the **swarm** of the files it has.

```
void receive_initial_holders(int nr_peers, tracker_info *mem)
    for (int i = 1; i < nr_peers; i++)
        MPI_Recv(&num_files_owned, ..., tag::INIT, ...);

        for (int k = 0; k < num_files_owned; k++) {
            receive_filename(filename, i, tag::INIT);
            receive_num_segments(&num_segments, i, tag::INIT);
            receive_segments(segments_owned, i, num_segments, tag::INIT);
```

3.3.2 Peer

The peers start by reading their config or input file which contains what files they have with what segments and what files they want to download. Each of these clients **informs the tracker** about the segments they own and waits for acknoledgement to start downloading.

```
void peer(int numtasks, int rank, logger *log)
    peer_info *input = read_peer_input(rank);
    send_num_files(num_files_owned);
    send_each_file_owned(input);
    receive_ack(ack);
    if (strcmp(ack, "ACK") != 0)
        exit(1);

    thread download_thread(download_thread_func, rank, input, numtasks, log);
    thread upload_thread(upload_thread_func, rank, input, log);

    download_thread.join();
    upload_thread.join();
```

3.4 Structure

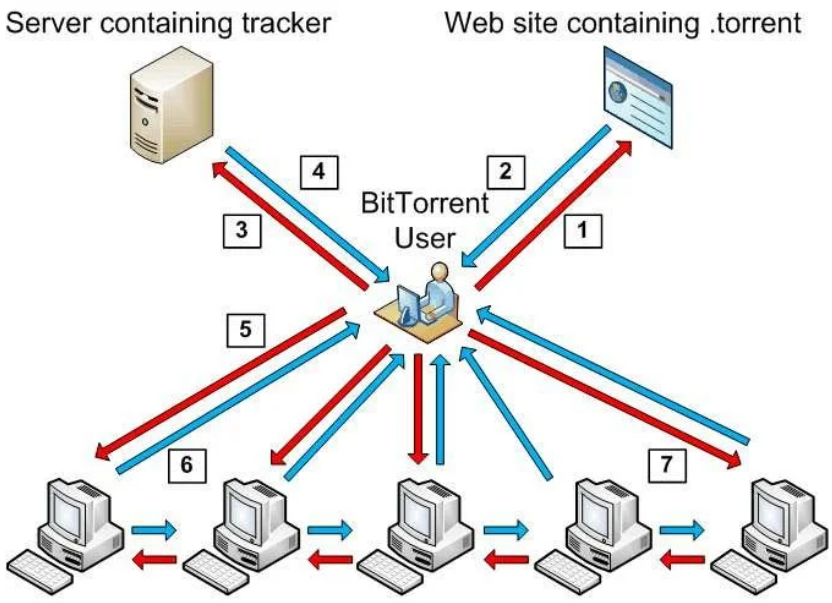
3.4.1 Tracker

Tracker has a designated structure pairing file to their swarm. It also takes into consideration what segments compose each file and memorise them in a data structure.

```
class tracker_info
private:
    map<string, swarm_info> file_to_peers_owning_it;
    map<string, vector<string>> segments_contained_in_file;
```

The swarm represents the list of peers that have segments of the file and what segments they own. Swarm is located **in the tracker** and is **updated based on what updates the peers transmit** to this tracker.

```
class swarm_info
private:
    map<int, vector<string>> client_list_and_segments_owned;
```



3.4.2 Peers

Peers have their designed class responsible for holding in memory the owned files, the wanted files and what segments it downloaded until that point.

```
class peer_info
private:
    map<string, vector<string>> files_owned;
    vector<string> files_wanted;
    map<string, vector<string>> segments_downloaded;
```

3.4.3 Download

The download thread has an important structure to discuss: the **request times** vector. This vector stores the time elapsed in the last request sent to each peer. This array is crucial in the decision making algorithm for choosing what peer to download from.

A smaller elapsed time means that the peer has a smaller number of requests pending and is a **better choice** for downloading from.

3.4.4 Upload

The main structure to discuss in the upload thread is the **request queue**. In order to keep track of the requests that are pending, they are pushed and popped in this queue. The implementation uses **non blocking MPI calls** to receive the requests and then adds them to the queue.

The kill channel makes requests and handles them via non blocking MPI calls. The request structure is firstly initialised and then it waits to receive the correct message from the tracker on the specified tag.

```
MPI_Request killReq = MPI_REQUEST_NULL;

void initialize_kill_signal_request()
    if (killReq == MPI_REQUEST_NULL)
        MPI_Irecv(&kill_message, ..., tag::KILL, &killReq);

void upload_thread_func(int rank, peer_info *input, logger *log)
    std::queue<MPI_Request> requestQueue;
    initialize_kill_signal_request();
    // ... Upload loop logic
```

3.5 Logic

3.5.1 Tracker

The tracker firstly establishes what files are in the topology and then waits for **commands** from peers. It is also responsible for **killing the upload threads** of the seeds when all the peers have finished downloading.

```
receive_initial_holders(numtasks, tracker_info_local);
send_acks(numtasks);
while (!all_peers_finalized(finished_downloading, numtasks))
    int source = recv_command(action);
    case action::REQUEST:
        handle_request(source, tracker_info_local);
    case action::UPDATE:
        handle_update(source, tracker_info_local);
    case action::FINALIZE:
        handle_finalize(source, finished_downloading);

send_message_to_upload(numtasks, action::KILL_UPLOAD_THREAD);
```

3.5.2 Peers

As described, peers have two threads: one for downloading and one for uploading.

The download thread **sends requests** to the peer with the fastest response time who has the segment and the upload thread **loops waiting for requests** from other peers.

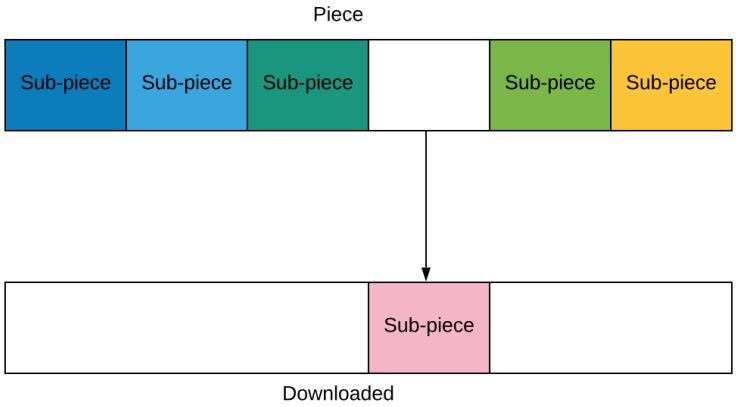
The tracker is responsible for killing the upload thread, while the peer itself kills the download thread when it finishes downloading the target file.

3.5.3 Download

This thread loops until it has downloaded all the files it wants. It throws request for each file to the tracker. The tracker responds with a list of peers that have segments of the file.

The algorithm chooses from the list of peers the one which had the fastest response time to the last request. It then sends a request to the peer for the segment. The peer responds with an acknowledgement that the segment has been sent.

```
void download_thread_func(int rank, peer_info *input, int numtasks)
    vector<double> request_times;

    while (!all_files_are_downloaded(input))
        for (auto &file : input->get_files_wanted())
            request(file);
            clients = handle_response_to_request(segments_contained);
            find_best_client(...);
            check_if_whole_file_was_downloaded(...);
```

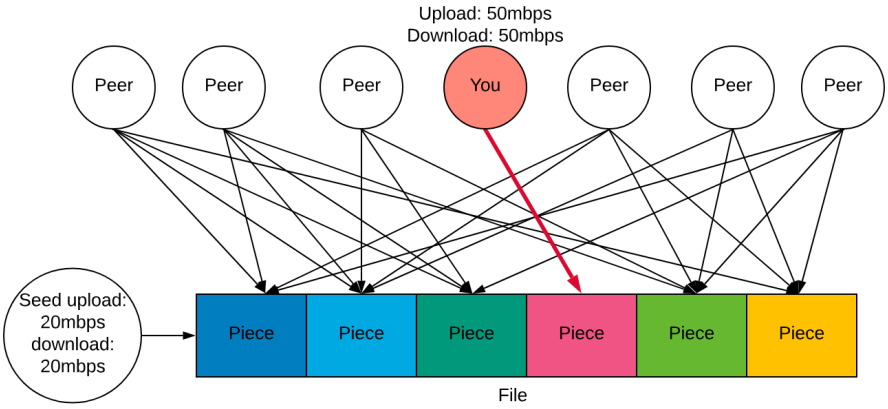



```
void find_best_client(file_segs, &clients, file, &cnt, &times)
    for (auto &segment : file_segs)
        if (cnt > 9)
            send_update_to_tracker(...);
            clients_update = handle_update_response();
            update_clients(clients, clients_update, file);

        if (segment not in segments_downloaded)
            for (auto &client : client_list_and_segments_owned)
                if (segment not in segments_owned)
                    if (times[client_id] < min_workload)
                        min_workload = times[client_id];
                        best_client_id = client_id;

            request_segment_from_best_client(best_client_id, ...);
            if (receive_requested_segment(best_client_id, times))
                download_counter++;
                peer_info_local->add_segment_downloaded(file, segment);
```

3.5.4 Upload

As presented, the upload thread loops until receiving a kill request from the tracker. It uses **non blocking MPI** calls to receive requests from the peers and to test for kill messages from the tracker.

When a request comes, the upload thread adds it to the queue. Then, while the queue is processed, the algorithm sends an **acknowledgement** to the source peer to signal the completion of the transfer.

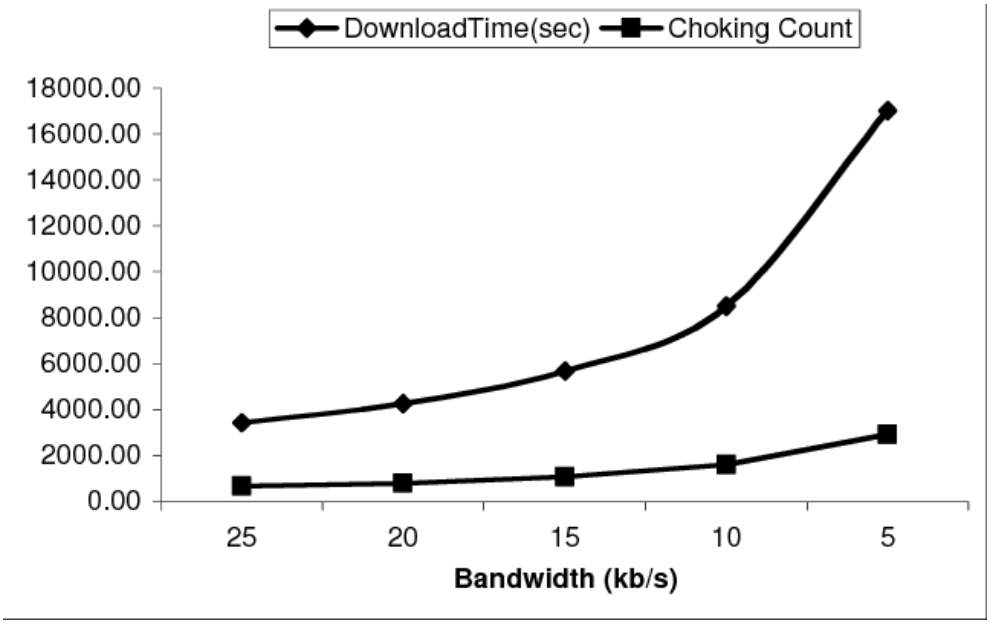



```
void upload_thread_func(int rank, peer_info *input, logger *log)
    initialize_kill_signal_request();

    while (true)
        MPI_Irecv(&action, ..., tag::DOWNLOAD_REQUEST, &recvReq);
        requestQueue.push(recvReq);

        while (!requestQueue.empty())
            if (check_if_received_kill())
                while (!requestQueue.empty())
                    auto req = requestQueue.front();
                    MPI_Cancel(&req);
                    requestQueue.pop();
                return;

            MPI_Test(&requestQueue.front(), &flag, &status);
            if (flag)
                int source = status.MPI_SOURCE;
                MPI_Send(ack, ..., source, tag::ACKNOWLEDGEMENT, ...);
                requestQueue.pop();
```

4 Comments

The current documentation proposed a simple C++ implementation using MPI, but the subject can be extented in various other key points such as:

- strategies for **maximizing the service capacity** of the system
- efficient **piece selection** strategy.



Peer Selection Strategy: This involves mechanisms to choose which peers to upload to and download from, aiming to optimize the network's performance and fairness:

- **Tit-for-Tat (TFT)**: Incentivizes sharing by preferentially uploading to peers that offer the best download rates.
- **Optimistic Unchoking (OU)**: Randomly unchokes a peer every 30 seconds, allowing for the discovery of peers with potentially higher upload capacities.
- **Anti-snubbing**: If a peer stops receiving data from another, it will cease uploading to that peer, preventing exploitation.
- **Upload Only**: Used by seeds (peers who have the entire file) to continue contributing to the network by uploading based on the upload rates of other peers.

Piece Selection Strategy: Focuses on the order and preference of file piece downloads to optimize download speed and file availability:

- **Strict Priority**: Prioritizes downloading full pieces in sequence, ensuring that complete pieces are available for sharing as soon as possible.
- **Rarest First**: Prefers downloading the rarest pieces available among neighbors, promoting file availability diversity across the network.
- **Random First Piece**: A new peer randomly selects its initial piece to download. The goal is to quickly acquire a complete piece, enabling the peer to start participating in the network through the Tit-for-Tat (TFT) algorithm.
- **Endgame Mode**: A peer requests all remaining blocks from all its neighbors. Once a block is received, requests for that block to other neighbors are canceled. This strategy reduces the time to complete a download and minimizes bandwidth waste from redundant downloads.

Overall, these strategies collectively form the basis of the **choking algorithm**, which is crucial to BitTorrent's efficiency. It aims to balance the network's resources and ensure **fair distribution of bandwidth** among peers

These techniques are designed to improve the experience for contributing peers, **discourage free-riding**, and ensure a more **robust** and **faster** distribution of files across the network.

5 References

1. APD Project, 2023 - UPB
2. A Survey of BitTorrent Performance - ResearchGate
3. BitTorrent Protocol Implementation - GitHub Repository
4. BitTorrent - Wikipedia
5. BitTorrent (BTT) White Paper - Feb 2019
6. About BitTorrent - Official Company Website
7. BitTorrent Protocol Analysis - Wireshark Wiki
8. Open MPI Documentation
9. MPICH Documentation v4.1.x