Further Development of BitTorrent Simulator in Erlang

Karwan Jacksi
Abstract

**Further Development of BitTorrent Simulator in Erlang**

Karwan Jacksi

Among many P2P file-sharing protocols in existence, BitTorrent is one of the few that has attracted significant attention by a wide range of users. It uses variety of algorithms for peer selection, piece selection and other tasks. Having a simulator that facilitates investigating of applying different strategies in implementing components of a P2P would be of great advantages.

An Erlang based BitTorrent simulator was developed by IT department at Uppsala university. The network side of the project had been rewritten in order to improve the functionality of the application. In this thesis work, a new and modular design approach for the client side of the implementation was employed, documented and incorporated into the application.

All nodes run in parallel, and they communicate with each other through the newly developed network module. A variety of options for the BitTorrent simulator are supported in the implementation, algorithms of the typical structure can easily be exchanged and used to experiment new ideas to find out how the swarm is affected by different approaches in implementing BitTorrent clients and trackers.

The report also reviews the structure of the previous thesis work, and explains the modifications made to the previously developed network module.
Acknowledgements

I am warmly thankful to my supervisor, Justin Pearson, for offering me the chance to work with him, and for his encouragement, guidance and support which enabled me to develop an understanding of the subject.

I also would like to show my deepest gratitude to Parosh Abdulla for his continues help and invaluable advices during my master programme.

I owe so much to my best friends Sobhan, Salah, Miran, Hashar, Amanj and Halgord for their great support and friendship.

I am also offering my kind regards to all of those who supported me in any respect during the completion of the project.

Lastly, the greatest thanks are to my family.

Karwan Jacksi
# Table of Contents

1 Introduction.......................................................................................................................... 1  
2 Problem description............................................................................................................ 2  
3 Background.......................................................................................................................... 3  
   3.1 BitTorrent terminology.................................................................................................... 5  
      3.1.1 Torrent ................................................................................................................ 5  
      3.1.2 Peer.................................................................................................................... 5  
      3.1.3 Seeder................................................................................................................ 6  
      3.1.4 Leecher.............................................................................................................. 6  
      3.1.5 Tracker.............................................................................................................. 6  
      3.1.6 Swarm................................................................................................................ 7  
      3.1.7 Interested.......................................................................................................... 7  
      3.1.8 Choke............................................................................................................... 8  
      3.1.9 Optimistic unchoking.......................................................................................... 8  
      3.1.10 Snub............................................................................................................... 9  
      3.1.11 Piece............................................................................................................... 9  
      3.1.12 Piece selection............................................................................................... 10  
      3.1.13 Peer selection................................................................................................. 11  
      3.1.14 End game...................................................................................................... 11  
4 Methods................................................................................................................................. 12  
   4.1 State............................................................................................................................ 12  
   4.2 Pieces and sub pieces................................................................................................. 13  
      4.2.1 Structure of pieces............................................................................................ 14  
      4.2.2 Set up pieces.................................................................................................... 15  
   4.3 Main Module............................................................................................................... 16  
   4.4 MasterClient module............................................................................................... 19  
      4.4.1 Runtime environment....................................................................................... 20  
   4.5 Client modules.......................................................................................................... 21  
      4.5.1 Client module................................................................................................... 21  
         4.5.1.1 Life Cycle.................................................................................................. 22  
            1.1.1 Check availability .................................................................................... 22  
            1.1.2 Incoming packets..................................................................................... 23  
            1.1.3 Send alive................................................................................................. 23  
            1.1.4 Check neighbours................................................................................... 23  
            1.1.5 Choking algorithms............................................................................... 23  
            1.1.6 Check pieces........................................................................................... 24
1 Introduction

BitTorrent is one of many peer to peer (P2P) file sharing systems that have been proposed and implemented e.g. Gnutella, Kazaa, etc., and it is one among very few P2P protocols that has managed to stand the test of extensive daily use by a vast user community [1].

The power of the BitTorrent protocol comes from splitting the file into several smaller pieces, once a piece is obtained by a peer, it can be shared with other peers in the swarm while the download continues. The effective mechanism of the BitTorrent protocol is to force the peers to use tit-for-tat strategy which imposes peers, but not extremely, to contribute to the swarm by transferring pieces to each other. Hence, the system scales well when the number of the peers increase. Additionally, the contribution of each peer increases the flexibility of the system toward the failure or the departure of peers [2][3].

BitTorrent is a distributed protocol, and implementations that function in a distributed manner give way to the BitTorrent to function substantially [2]. Erlang is one of the desired programming languages for distributed applications. It is main strength is support for concurrency. Erlang is a general purpose concurrent programming language, it provides a small and powerful set of primitives for creating and managing processes. In Erlang, the processes are neither operating system processes nor threads, but lightweight processes. Communication among processes are very fast and simple, they use message passing to communicate with each other instead of shared variables, which eliminates the need for locks. Every process has its own mailbox, which is a queue that keep messages that have been sent by other processes [4][5].

Having a simulator that provides users with practical feedbacks and studies a problem at different levels of abstraction would be of great advantages and could easily produce fundamental results [6]. Different behaviours of BitTorrent clients such as various piece selection algorithms were main reasons for coming up with the idea of setting up the simulator in the first thesis.

Strong concurrency and distributed features of Erlang, led to choose this programming language for the simulator. Besides, the sequential subset of the Erlang is a functional language which gives a joy during programming.
2 Problem description

The main goals for this thesis were to rewrite the client side of the code in a way that clients could work and communicate with each other through the implementation of the new network. Data structures should be used in the new implementation that facilitate future developments. Rewrite things in the implementation that do not make much sense, remove redundant data and update old functions and algorithms to make them more efficient. Incorporate the new implementation of the code into the application.

Analysing the source code of the original project have resulted in general ideas of how the original application was working. The modified application, however, was not working due to the implementation of the new network. Therefore, developing the client core was needed in an attempt to allow the BitTorrent nodes interact with each other by using the new design of the network core.

The code of the project was quite large, very large state tuples with only global definitions to access elements in these large tuples, many different functions were used to send/receive packets to/from the network, and many Macros were used in header files to define the sizes of packets a client is sending to other nodes in the network. Consequently, the code was unnecessarily complex, hard to understand and difficult to modify. The covered documentation in the original thesis was rather poor, so a part of this report goes into describing this system, but the major part of the thesis is the description of new modules for BitTorrent nodes and their interactions with the implementation of the new network.
3 Background

When we download a file from the internet we, usually, use HTTP/FTP protocols to download the file from the server that holds it [7], this is when we use client-server architecture, illustrated in Figure 1. Here we can notice that the only place that has the file is the server which has to upload this file to all clients that are requesting the file. Obviously the bandwidth of the server would become a bottleneck if there are too many clients trying simultaneously to download and get the same file from the server. Therefore, client-server paradigm can become congested and overload the server with too many requests when the number of concurrent client requests to a given server increases [8]. On the other hand, the client-server model lacks the robustness, since it has a single point of failure [8]. Solutions to these issues are to increase the number of servers and load balancers to let the performance and robustness grow, but these solutions would rather fall into a costlier system in the hardware point of view.

![Figure 1: Client-Server Model](image1)

In contrast to the client-server model, P2P model offers more than a single source for the files to be downloaded, instead it distributes resources among many nodes, as in Figure 2. Thus, if one node fails or rejects to upload the file, peers can get the leftover pieces of the file from the remaining nodes in the network. Opportunities to get pieces from other peers increase when the number of nodes
increase, accordingly, the bandwidth is used in an efficient manner when the number of clients increase [9].

With the BitTorrent protocol, when multiple people are downloading a file at the same time it makes the use of the fact that rather than downloading the file from a single source, each client is capable of uploading the pieces of a file (which it has already been downloaded) to other peers that are asking for them simultaneously. In other words, each client becomes a source for the new pieces that has been received [7].

When a client wants to share a file, first it creates a small descriptor file called “torrent” and puts it on an ordinary web server. This file contains the descriptions about the file(s) to be shared and about the BitTorrent server(s) called Tracker. The trackers are responsible for helping clients to find each other rather than keeping and distributing the actual data, this is how the protocol reduces the demand of the bandwidth of the server (tracker) [10].

The BitTorrent client connects to the tracker(s) that is specified in the torrent file, and asks the tracker to get some peers. The tracker will reply with a number of peers’ IPs that have pieces of the desired file. The client (that receives a list of peers) tries to connect to these peers and starts requesting for various pieces. Peers (that want to download a specific file) should first get the torrent file for the file they want to download, and connect to the tracker that is specified in the torrent file, then they should follow the directions from the tracker which determines where to get and download pieces of the requested specific file.
The BitTorrent clients can only upload to a limited number of peers because they have a limited number of upload slots to assign to other peers. Consequently, when an upload bandwidth of a client is saturated, BitTorrent clients use a strategy called tit-for-tat to optimize their download speeds [11].

The participation of sharing a file is achieved when the upload bandwidth is exchanged with the download bandwidth. Therefore, when a peer refuses to upload in return to the client that is downloading from it, the BitTorrent protocol will choke the connection with the peer that refuses to upload, and allocate this upload slot to another peer that is hopefully participating more in uploading [11].

3.1 BitTorrent terminology

In order to get more familiar with BitTorrent protocol definitions, this section is going to give an overview of some common BitTorrent terms.

3.1.1 Torrent

A small metadata file with .torrent extension. The torrent file contains the address (URL) of the tracker that coordinates the action of all peers engaged in downloading in the swarm. It also contains information about all the files it makes downloadable including their names, lengths and hashing information, all of which are used by clients to confirm the state of the data they receive. Torrent file does not include any portions of the data itself [12].

3.1.2 Peer

Any computer on the internet that is running a BitTorrent client and sharing files to which other clients connect and transfer data is called a peer. A peer or client, commonly, does not have the complete file but only parts of it. If a peer has a complete file then it is called a Seed as shown in Figure 3. Peers are also called Leeches or Leechers, but these terms are usually used for peers who have a negative effect on the swarm, which they are downloading much more than they upload. However, the term peer can be used to refer to any participant in the swarm but this is an informal definition [9]. In this report, the term client is mostly used for the sender, while the term peer is used for the receiver (recipient).
3.1.3 Seeder

A peer that offers a complete copy of a specific file to be uploaded to other peers in the swarm is called a seeder. Once a peer downloads all pieces of a file it becomes a seeder (Figure 3). Seeders are totally not interested in downloading any pieces of the file since they already have the complete copy of that file, but instead they only upload to leechers. A swarm that has a large number of seeders has a better chance of getting a higher downloading speed [9].

3.1.4 Leecher

Leech or leecher is a peer that does not have the complete copy of a file to be shared but it only has some parts of the file (Figure 3). This term has the same meaning as a Downloader, since all of them do not have the entire copy of a file of interest [9].

3.1.5 Tracker

Tracker is a server on the internet that keeps track and coordinates the action of the BitTorrent clients. It acts as a navigation centre for the swarm and it is responsible for helping clients to find each other in their swarm (Figure 3). Without the tracker, clients cannot find each other. Thus, peers cannot form their neighbourhood, and eventually no download and upload occur among them [7]. Peers get the URL of the tracker from the torrent file. Trackers do not have any pieces of the actual files to be downloaded by the clients and they are not directly participated in the data transfer [9].

Peers periodically send their information to the tracker and in return they receive information about other peers to which they can contact. These information are important for the administration purposes and can be used to view the overall performance of the swarm [7].

Trackers often temporarily go down, therefore peers cannot connect to them. If the tracker goes down, the client (that is already in the swarm and it is in contact with some other peers in its neighbourhood) is capable of continuing data transportation with those peers, but no new peers are able to get into its neighbourhood [13]. Therefore, the client may miss to download all pieces of the file if there is no seeder in its neighbourhood or if all connected peers do not have all pieces of the interested file. But, when the tracker returns back from the error (since its errors are often temporary) and supplies peers with new IPs, the
cooperation process will get resumed and the downloading activity will continue [12].

![Figure 3: Tracker, Seeders and Leechers](image)

### 3.1.6 Swarm

A group of peers that are collectively connected for a particular torrent file form a swarm. Peers in the swarm could be an ordinary peer (leecher) or a seeder [12]. For example, five ordinary peers and two seeders (as in Figure 3) make a swarm of seven.

### 3.1.7 Interested

This term refers to the state of a downloader. The peer that wants to get some pieces of a file from the other end of the link (that has any pieces) is marked as interested. In other words, a client who would upload pieces of a file to a downloader, will flag the downloader as 'interested', otherwise the flag will be marked as not interested or 'uninterested' [12].
3.1.8 Choke

Describes a client which refuses to upload pieces of a file to another peer. When a peer gets choked by a specific client, the peer cannot download from that client while uploading can still happen. Peers cooperate in sharing when they upload, but they are not cooperating when they choke peers. Choking temporary stops uploading, while downloading can still happen [10].

A peer chokes another peer when the other end is a seeder which does not need any pieces to be downloaded, or when the peer reaches the value of $\text{max}_\text{uploads}$ (see Illustration 1) which is uploading at its full capacity [9].

There are many algorithms for how to choke a peer, and they are not technically part of the BitTorrent wire protocol, but they are needed for a good performance [10].

A good choking algorithm should meet some criteria [14]:

- It should utilize the available resources in a good way which means it has to provide a fairly consistent download rates.
- It should pass the number of simultaneous uploads for a good TCP performance for everyone.
- It should periodically use the optimistic unchoking
- It should avoid the fibrillation\(^1\) [15].

3.1.9 Optimistic unchoking

Periodically a client should find out a number of peers (that were previously choked) and try to select one of them to get unchoked regardless of the peer’s current downloading rate, see Illustration 1. A peer that gets optimistic unchoke rotates every certain period. This approach gives three times more chance to the newly connected peers to get unchoked, this gives them a decent chance of getting a complete piece to upload [15].

\(^1\) A term used when peers choking and unchoking too quickly.
3.1.10 Snub

If a client does not receive any pieces in a certain amount of time, it marks the connection as snubbed, see Illustration 1. Occasionally a client that is connected to many peers in the swarm will find itself in a state where it is choked by all peers it was formerly downloading from [15]. This client will use a snubbed flag in an effort to prevent this status, this process is called Anti-snubbing. It notices that the peer with whom it would like to transfer pieces has not sent anything in a while, and rather than leaving it for the optimistic unchoking to eventually unchoke that peer, the client instead reserves one of its upload slots for sending to that peer. The purpose of keeping track of this variable is to improve download speeds [12].

3.1.11 Piece

Is a term that refers to the torrent file divided up into equal specific sized pieces. The amount of pieces depends on the size of the piece and the size of the file. Pieces are distributed in a random way among peers in order to optimize the transmission of the data in an efficient manner [9].
3.1.12 Piece selection

Which piece of the file should be selected by a peer to start downloading next. The selection of the pieces has great impact on the performance [10]. Selecting pieces for downloading in a good order is very important not only for that particular peer but for the whole swarm. Therefore, it is necessary to select a piece in a way that there should not be any missing pieces of the file in the swarm. Besides, the goal is to distribute the pieces (a peer already downloaded) directly to various peers in order to increase the download speed. This helps for keeping a complete copy of a file even if a seeder(s) leaves the swarm at some point [7].

There are several algorithms for how to select pieces to download:

- **Strict priority**
  When a client requests a piece of a file from another peer, it has to download and complete all sub-pieces that belong to that piece before requesting any other pieces of the file. In other words, the client selects the remaining sub-pieces from that particular piece before requesting sub-pieces from any other pieces. By using this policy, it will help the client to get the complete piece as quickly as possible [10].

- **Rarest first**
  In this algorithm, if a client wants to select a piece to be downloaded next, it selects the piece with the lowest number of providers. Rarest first algorithm prevents pieces from disappearing from the swarm, because the pieces that are not selected for downloading become more infrequent which eventually will be selected by peers [10]. This technique makes sure that the most common pieces are left to be downloaded later. Therefore, the peer that is currently offering an upload will later reduce the probability of having nothing of the interest [10]. This algorithm is highly used after the client gets a complete copy of the piece.

- **Random first**
  Selects a random available piece to be downloaded next. This algorithm is very important for the client when it starts downloading, since, at that time, the client has nothing to upload. Thus, its chances for getting pieces to download is very poor due to tit-for-tat strategy. Hence, getting a complete piece as quickly as possible is essential. The technique is commonly used for peers when they are newly connected to the swarm.
and have nothing to contribute with it, therefore they start to select pieces at the random until the first piece is assembled, and then they change the strategy into rarest first [10].

3.1.13 Peer selection

When the process of the piece selection has been done, a neighbouring peer with whom to download from is needed to be selected. All peers are periodically rated depending on their download/upload rates and some other criteria [16]. Peers are sorted according to their rating values. Assume only 5 peers can download simultaneously. The peer selection algorithm will unchoke as many of the best rated peers as needed so that exactly 5 of these peers are interested. When a peer from the top rated peers becomes interested in the future, the algorithm will pick the worst unchoked peer and choke it. This selection method has a negative influence for the newly connected peers since it cannot show a fair scheme on their downloading rate [16]. To make sure that one of the new peers will have chance of being unchoked, a periodical random peer selection regardless of their rates is necessary.

3.1.14 End game

Sometimes a client with a very low download rate will request pieces from its neighbouring peers, which makes very unlikely to get the pieces soon [9], since it has a poor downloading rate. This is not a problem in the midway of the download, but it could possibly delay the downloading process to finish [10]. Therefore, when the download is almost complete, there is an attitude for the last few pieces to run in slowly. To get rid of this phenomena, the client requests its missing pieces from all neighbouring peers at the same time, and when it gets the requested pieces, cancel messages are sent for the pieces which arrive in order to save the bandwidth from the redundant data. Not much bandwidth is wasted when sending cancel messages since the endgame period is very short and the last pieces of a file is always downloaded quickly [10].
4 Methods

This section describes the whole structure of the system and how things were implemented in the project mostly for the BitTorrent nodes. Many things were needed to be changed in the network interface module as well as the router module. The reason for that was to get the ability of making both the network and the BitTorrent nodes work and communicate with each other. The client side of the project has been developed to be more modular, understandable and can deal perfectly with the implementation of the new network. Hopefully this section will give a good idea of how things were structured and why.

4.1 State

A state is a collection of information that directs a client or a tracker to what behaviour they should follow. Without the state they cannot do anything. The state holds all information about what pieces of a file the client has, what algorithms and custom functions the node should use, their IPs, process IDs, packets they are sending and receiving, etc. the state uses several records, ets maps and various macros to hold all of these information. See Illustration 2 for client's state and Illustration 3 for tracker's state.

```
-record( clientstate,
    { ip, ---- IP: IP address of the client.
    id, ---- ID: the Process ID of the client.
    router, ---- The name of the router the client is connected to.
    Tracker, ---- IP of the tracker the client is connected to.
    handshakeMsgSize, ---- The size of a handshake message.
    Seeder, ---- Seeder: true if the client is a seeder.
    ...
    } )
```

Illustration 2: Client's State

---

2 Ets map is an Erlang’s built-in term storage, it provides the ability to store very large quantities of data in an Erlang runtime system [5].
To transfer a file from a peer to another one through the network, it needs to divide this file into smaller parts called pieces. Each piece is in turn divided into smaller pieces called sub pieces. The amount of pieces a client has depends on the piece size and the size of the data to be downloaded, which are specified in the torrent's configuration file (a simplified torrent file). The size of the sub pieces and the amount of the pieces a client should start with are defined in the client's configuration file. Pieces are generated in a random mode, therefore, the pieces of a specific client are different from the pieces of another one even if they are belonging to the same group of clients (more about clients in section 4.4). Since pieces are random, a range for generating these random pieces is required. The range should be the number of the pieces a file is divided to. Thus, if the size of the data to be shared and the piece size are changed in the torrent file, a change in the range is needed as well. In Example 1, it describes the process of how many pieces the client will start with and from what range.
4.2.1 Structure of pieces

A binary tree is used for pieces in the client's state. Each piece of the file has an integer number which is used as a piece number and also as a key for the node in the binary tree. While the value of the node is a boolean which indicates what pieces the current client owns. See Illustration 4.

The pieces that the client owns, are flagged as true, while the pieces it is missing are flagged as false. Each node that has the false value will be replaced with a binary tree, which is used for sub-pieces that exist for that piece. Key value for the sub-piece tree will be the sub-piece number itself while the value of the node will either be true or a list of items. The item in the list is a tuple of an IP and a sub-piece's status. As shown in Figure 4.

```
{25,
  {13,true,
   {7,true,
    {4,true,
     {2,false,{1,false,nil,nil},{3,true,nil,nil}},
     {6,false,{5,false,nil,nil}},
     {10,false,{9,true,{8,true,nil,nil},{12,true,{11,true,nil,nil}}}}},
   {20,false,
    ....}}}
```

Illustration 4: Binary Tree of Pieces

The binary tree consists of 25 nodes, as shown in the first line. This number is also used as the number of pieces that the file has been split to. The pieces that have false values means this client does not have these particular pieces.
When the value of the sub-piece node is true, it means this peer already owns that sub-piece and it does not need to send a request for downloading that sub-piece, but when it is a list of items, then it means this peer does not have that particular sub-piece and the IP in the item has it. Therefore, the client needs to request and download the sub-piece from the peer that is in the item.

The status in the tuple is an atom() which is used to indicate the status of the sub-piece is the item, see Figure 5.

The status in the tuple could one of the followings:

- **Requesting:** when the client is requesting the sub-piece from the peer in the tuple.
- **Waiting:** when the client has already sent the request to the peer and is waiting for an answer.
- **Downloading:** means the client is downloading the sub-piece from the peer in the tuple.

### 4.2.2 Set up pieces

The client needs three properties from the configuration files to initialize its pieces. These properties are the size of the file, piece size, and the pieces that the
client should start with. A range should be followed by the pieces that the client is starting with as well.

The client first calculates the number of pieces a file is divided to by simply dividing the size of the file by the size of the piece, then it creates a binary tree of the same size as the number of pieces. The Values of the nodes in the tree are all set to false. The client, later on, creates a list of random numbers in the range that was given to it, and modifies the value of nodes to true according to the generated numbers in the list. Afterwards, the client will start to set up sub pieces for the nodes that were kept as false. It will calculate the amount of the sub pieces needed for each piece and the amount of the sub pieces needed for the last piece, since the last piece could be smaller than the other pieces. For each node that has a false as a value, a client creates a binary tree for sub pieces and replaces all false values with this new tree. The key value for each node in the sub pieces' tree, as it mentioned in 4.2.1, is the number of the sub piece, while the value of each node is an empty list which is used to keep IPs of the peers that have these sub pieces and their status.

4.3 Main Module

It is the module that the application starts with. Its function is to spawn main processes needed for the system such as random process that generates random
numbers, network processes and master client process which is responsible for spawning all BitTorrent clients in the project. It provides processes with all required data and settings for the initialization. It takes these data from the files called configuration files, Illustration 5. The configuration (or Config) files are used to set up all processes in the system. They are used to install network nodes with different properties and load different settings and algorithms for the tracker and clients.

Illustration 5: Network Configuration File

<table>
<thead>
<tr>
<th>Section R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>#following IP will give the router IP address 10.8.0.1 and will give clients IPs from 10.8.0.2 to 10.8.254.254</td>
</tr>
<tr>
<td>Net=10.8.x.x</td>
</tr>
<tr>
<td># Target=Sendspeed, Errorrate, Destination IP, Netmask</td>
</tr>
<tr>
<td>R2=700, 0.5, 30.0.0.0, 255.0.0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net=30.x.x.x</td>
</tr>
<tr>
<td>R1=700, 0.5, 10.8.0.0, 255.255.0.0</td>
</tr>
<tr>
<td>R3=700, 0.5, 10.1.1.0, 255.255.255.0</td>
</tr>
</tbody>
</table>

The main module starts from the network side of the system by initializing all nodes needed for the network. It starts by making all routers (Illustration 6) and the connections between them (Illustration 7). It takes the network config file and tries to get all settings (by the help from the routing module) for the router processes such as router names, their speeds, IP-range, etc., and forwards these settings to the module named dijkstra which is responsible for the network side of the implementation to get initialized. The module dijkstra will initialize the states of the router processes and will make the connections between them, then it will turn into the initial state of the router where the clients and the other nodes such as other routers will be added into the router’s table. Finally, the router processes will start their normal functionality after adding all nodes in the network to its table [2].

---

3 The network side of the application has been developed by Niclas Stensbäck [2].
When the network nodes become ready, the main module tries to set up the tracker process, it gives the routing module the tracker and the torrent config files together with all available routers (that have been initialized in the network module). The routing module will get the necessary settings (the router that the tracker should connect to it, the tracker's bandwidth and the process ID of the router) from the config files, then it will spawn the tracker process. The routing module will spawn another process as well which is called link process. The link process is needed to connect and link the tracker with the router to transfer data between them. The PID of the tracker together with the link PID are sent to the router process in order to be added in the router's table, and a copy of the link PID is sent to the tracker process as well which is waiting for this PID to get started its normal task.

The main module, finally, will spawn another process called MasterClient which is in the charge of spawning all BitTorrent clients in the simulator. The client's config file, the torrent file and information about all available routers are

Illustration 6: A List of Sections with their Variables after Reading from the Network's Configuration File

```
["R1",
["Net","10.8.x.x"],
["R2","700","0.5","30.0.0.0","255.0.0.0"],
["R2",
["Net","30.x.x.x"],
["R1","700","0.5","10.8.0.0","255.255.0.0"],
["R3","700","0.5","10.1.1.0","255.255.255.0"]]
```

Illustration 7: A List after Making Connections between Routers

```
[["R1","R2","700"],
["R2","R1","700"],
["R2","R3","700"],
["R3","R2","700"]
```
sent to the MasterClient process.

4.4 MasterClient module

The main function of this module is to spawn all BitTorrent clients in the simulator and set up their states from the configuration files. Clients are spawn in groups, the number of groups (as in Illustration 8) depend on how many sections are placed in the client's configuration file, while the number of clients rely on how many clients are specified in each section or group. Each section creates a group of clients with identical properties, but the clients in this group are different from the clients in other groups that have been created from other sections.

A general section must be specified in the config file so all clients can setup their states according to settings specified there. If there are some other sections specified in the config file, each section creates a new group of clients with the same properties as specified in the general section but the properties in the new sections will override the properties that have been taken from the general section.

Another function of this module is to spawn the link processes for every spawned clients, and send the PID of each client with its link PID to the router that is supposed to connect to it. Finally, it sends the initial initialized state to each spawned client (each client is a single process) that is waiting for its own state to start with it as a singular client.

Settings in the client's configuration file include the number of clients, what algorithms they are using, their bandwidth, piece size, etc.
4.4.1 Runtime environment

The MasterClient process starts from the method `new_init/3` which takes three arguments. The first two arguments are the config files for both the client and the torrent file. The third argument is a record where one of its fields is a list of all available routers that are needed for the clients. The process initializes an initial states for the clients through the help from the `csetupstate` module by getting some properties from the client’s config file and loading all sections that have been specified in that file. Afterwards, it gets all variables for the section `General` which is necessary to be specified in the config file together with at least one of the custom sections. It constructs a record and initializes all variables in the record with values that have been loaded from the section `General`. Then, it loads all values from other custom sections in the client’s config file and puts them in a list called main list in a way that each section has its individual list nested in the main list (Illustration 9). Each section in the main list becomes a group of clients with identical properties.

Each group of clients is supposed to connect to a specific router. The method `send_to_tick/3` finds the name of the router for each group, then it sends a signal `givebirth` to the method `tick/1` which sends a number of clients that was given in each section to the function `make_children/4`.

The function `make_children/4` will spawn the clients and their links to the network for each section and it will send the PIDs of the spawned clients and their
spawned links to the router that they are supposed to connect to it. Finally, it sends the initialized state to every spawned clients that they are waiting for it in their processes.

4.5 Client modules

The client process uses several modules and algorithms to simulate the behaviour of the BitTorrent client. This section will give an overview about different modules the client is using and what are the functions of these modules. The algorithms and runtime of the process is also described in this section.

4.5.1 Client module

This is the main module for the client that behaves like a BitTorrent client in this simulated environment. The module starts when the MasterClient process spawns the client process. It tries to connect the client with the simulated network and gets the IP of the client from the router. Furthermore, the client will look up for the DNS and register itself at the tracker. Then, it moves into its main loop which is called client's life cycle (as in Figure 6). In the life cycle a continual behaviour is

Illustration 9: A List of Sections and their Properties after Reading from the Client’s Configuration File

[["General", "Router","R1"],
 ["PieceSelection","mod_2q_ps","ps_random"],
 ["PieceSelectionInterval","3000"],
 ["MaxAmountOfPeers","17"],
 ["Choking","mod_smartchoke","smartchoke"],
 ["Section_name1", 
 ["Router","R2"],
 ["Amount","4"]],
 ["Section_name2", 
 ["Router","R3"],
 ["Amount","5"]]]
executed, the client updates its state, which it is working on, depending on what custom functions are returning and what various messages it is receiving from the network.

4.5.1.1 Life Cycle

The client's life cycle, as shown in Figure 6, starts when the connections and the registration with the router and the tracker goes successfully, in other words, this loop starts when the client successfully connects to the swarm. At each cycle, the client checks whether or not it is necessary to use any of the functions or algorithms required to update the state that it is working on. All custom functions and algorithms that the client is running during its life cycle are described in details in the following descriptions.

Figure 6: Client's Life Cycle
1.1.1 Check availability

The first action the client performs in its life cycle is checking the availability of itself. Clients can disconnect from the network and sometimes reconnect to it once again. In this stage of the cycle, the client uses the connection handler module to perform this functionality which, afterwards, it will be clear whether to disconnect, die or continue as it is.

1.1.2 Incoming packets

When the client gets the signal from the availability custom function to continue on its life cycle, it checks whether there is any incoming packet. The module packet handler is responsible for handling incoming packets. A new state is returned to the client after processing the packet.

1.1.3 Send alive

When a peer in the neighbourhood does not sent any messages in a while, this peer is no longer listed as available peers to the client. Therefore, the client closes the connection with the peer. To fulfil this functionality, the alive handler module is invoked by the client, which periodically checks (the interval given by the client’s config file) if it is time to send alive signals. It collects all IPs from its neighbourhood and sends alive messages to all of them so to maintain the connection alive.

1.1.4 Check neighbours

Every now and then, the client needs to make an announce and request new peers from the tracker in an attempt to form a bigger neighbourhood of peers. To achieve this, the client calls the neighbourhood handler module in order to check if it is required to get new peers for its neighbourhood or it has sufficient number of peers. More about neighbourhood handler in the section 4.5.1.5.

1.1.5 Choking algorithms

In this stage of the client’s life cycle, the client calls the choking algorithms that are given by the configuration file. It checks if enough time has passed since the last
time this function was called, then it calls the choking handler module to perform choking algorithms. Details and algorithms that are used in the simulator are described in section 4.5.1.6.

1.1.6 Check pieces

The client in the last stage of its life cycle checks whether it needs to make a request for the missing pieces, and from which peer(s) it should request the pieces from. It uses the module called pieces handler which is responsible for this task. A new state for the client is returned after calling the pieces handler then it returns back to its life cycle.

4.5.1.2 Alive handler

Peers may close the connection with the other end if they do not receive any messages for a certain period of time. To find links that do not work or indicate links that should be preserved, short alive signals\textsuperscript{4} are sent from the client to its neighbours. Alive signal makes the connection between peers active and prevents the link between them from being broken. Alive signals are short and do not take much bandwidth.

This module is performing this role for the clients. At every predefined intervals, the client checks all peers in its neighbourhood while if they are still alive or the link between the two is not operating. If the client, in a while, does not get messages from a peer and it has not received alive signal from the same peer, it will assume this peer has been died and it closes the connection with the peer.

4.5.1.3 Connection handler

The connection of clients is not only important for the client itself but it is significant for the whole swarm, since each client becomes a new source for the pieces it has downloaded.

Clients, in general, have unsteady attitudes, they are occasionally disconnect and reconnect once more to the network. The connection handler module gives this behaviour to the clients in the system. An examination for the connection of the client is performed to verify what is the availability of the client. It calls the availability custom function (see the custom functions 4.7.1.3) to check the

\textsuperscript{4} Keepalive signal in some textbooks.
current status of the client and whatever it should follow. There are three possible actions for the client to get after calling the availability custom function. These actions are disconnect, die or idle.

If the action idle is returned from the custom function, then the client should get back to the life cycle and continue on the remaining parts of it. While, if the client gets the die, it should prepare for how to kill its process. The client should disconnect in a nice way from the tracker and the network otherwise the simulator would crash. At the first, the client should send the signal stopped to the tracker to disconnect itself from the tracker. Then it should send the disconnect request to the network asking for the disconnection, and finally it should terminate the process and exit.

If the returned action from the custom function is disconnect, then the client sends the signal stopped to the tracker, clears all messages from its state that have been sent and received, disconnects from the network, drops all incoming packets to clear the queue from messages, and waits for the reconnect signal if it wants to reconnect to the swarm. Here, the client will check the availability custom function once again whether to reconnect or terminate the process. If the action die is returned, the client exits from the life cycle and terminates the process, otherwise the client reconnects to the network as in Figure 6. The reconnect process takes the same way as a new client connects to the swarm, so it sends requests for the connections, registration, getting the IP, etc., then it enters its life cycle.

4.5.1.4 Packet handler

The module that takes cares of all packets a client is receiving from the network. Whenever the client receives a packet from any node in the network, it uses this module to process these packets. This is when the packet is not a meta-call packet, since the networkiface module is responsible for the meta-call packets. The client in this module waits for incoming packets from the link process (that connects the client with the router). When a packet is received by the client, a sequential number from the client's state will be given to the packet, which will be used as the packet ID. The client keeps all incoming packets in its state, this task is also done in this module, it saves all incoming packets for the client in a binary tree and uses the packet IDs as the index keys for the packets in this binary trees.

The packet handler module classifies the incoming packets as well. After giving the ID to the packet and keeping them in the client's state, packet handler checks the contents of the packets and sends them to different places to get processed according to their contents.
The other task of this module is to check whether or not the outgoing packets are arrived successfully to the network. Whenever a client sends a packet to the network, it needs to get the acknowledgement for that packet. The client uses this module to check packets, by checking the ID of the packet with the acknowledgement key, then the module reports the status of the packet.

4.5.1.5 **Neighbourhood handler**

The module that handles the process of how the clients form their neighbourhoods. When a BitTorrent client connects successfully to the swarm, it needs to contact some other peers for the desired pieces to be downloaded. So, the peers that have the desirable pieces are important for the client. Therefore, the client forms a neighbourhood in an effort to keep these peers in contact, and continuously requests pieces from them.

To achieve this functionality, every so often, the client should make an announce to the tracker and request new peers from it in an attempt to form its neighbourhood or to make it bigger. In this module, if enough time has passed since the last announce or the last request call that was sent to the tracker, the client checks its neighbourhood by calling the neighbourhood custom function (see section 4.7) to see how many peers it is connected to, how many peers it is allowed to make connections with, and whether or not it needs more peers to add them into its neighbourhood. After calling the neighbourhood function, the client checks the response of the call which is either the signal *don't_announce* when the client does not need any new peers, or the *announce* signal with an integer number. The client uses this number as a number of peers it needs to send the request for. The client, afterwards, makes an announce to the tracker asking for more new peers. The tracker will respond to this announce (more about the tracker in 4.6) with a set of peers' IPs that the client is allowed to contact with them.

4.5.1.6 **Choking handler**

In order to give a good performance to the swarm, choking algorithms are necessary. In this module choking algorithms are handled, it checks the optimistic unchoke, anti snubbing and choking for the BitTorrent clients, and calls the relevant custom algorithms for these functions. Details are described below.
1.I  **Optimistic unchoking**

The first thing the module does is to check for the optimistic unchoking. It checks the default optimistic unchoke for the client to find the peer which has previously been unchoked optimistically. If there is such a peer, the module checks the period that this peer has been unchoked optimistically by the client. If this period has passed (the interval of optimistic unchoke that was given to the client by its config file) then the client chokes the default peer and calls the custom function for the optimistic unchoking to get a new choked peer (if exists) and unchocks it optimistically. The default optimistic unchoke peer must be removed from the client's state.

If the peer is not found in the client’s default optimistic unchoke, the module simply lets the client to call the optimistic unchoke algorithm. The client unchokes the returned peer, if there is such a peer, and gives the unchoked peer a time-stamp so to know when the peer has got unchoked. Later on, the client puts this peer into the default optimistic unchoke. Meanwhile, when the peer’s choking status is changed, the client puts the IP of this peer into a list called *chokediffs* which is used for keeping track of which peers' choking status have been changed during the current cycle of the client's life. New choking status messages are sent to the connected peers whom choking status has been changed.

1.II  **Anti snubbing**

To solve the problems for peers that have not gotten any pieces in a certain amount of time, and rather than leaving them to eventually get unchoked by optimistic unchoke which takes time, the client performs the so called anti snubbing. It starts by going through all the snubbed peers and removes the snubbed status from the peers that have been snubbed long enough. Afterwards, it checks if enough time has passed since the last call of the anti snubbing. If it is time, then the client calls the custom function for anti snubbing to get a list of new peers that are needed to get snubbed in this particular cycle. The client, then, sets the snub status to *snubbed* for all IPs that anti snubbing algorithm was returned, and gives to each of them a time-stamp to know when it got snubbed. Later on, it sets choking status to *choked* with the consideration of adding peers (that their choking status have been changed) to the list *chokediffs*.

The client, afterwards, checks extra the optimistic unchoke list,
exous⁵, and takes away all peers that should no longer be unchoked. Subsequently, it calls the optimistic unchoke algorithm for the additional unchokes and inserts IPs that are returned from the algorithm into the list exous. The number of extra optimistic unchokes depends on how many snubbed peers are there.

1.III  Choking

The final action the client does in the choking module before going back to the life cycle is to perform the choking behaviour for the connected peers. It checks the time interval with last time the client performed choking algorithm, if it is more than the interval that was given to the client by the config file, then the client calls the choking algorithm to get two groups of peers, a group that needs to get choked and a group that needs to get unchoked. The client, afterwards, chokes and unchokes both groups, but for the group that is going to get choked, a doubly check for each peer is performed to make sure that the client is not going to choke the peers that have been unchoked optimistically.

4.5.1.7   Pieces handler

To download pieces of a file from another peer, a peer(s) must have the required pieces in the neighbourhood. To determine the peer, this module is needed. The pieces handler module is managing which peers have what pieces so the client can find its target peers to select and request pieces from them. The module checks if the client is a seeder or not. If it is a seeder then there is no need to check its pieces and call the custom function for the piece selection, since it is a seeder and seeders have all pieces. For leeches or leechers, the module checks the last period it has called the piece selection algorithm. If it is time then it calls the piece selection algorithm to get a specific list. This list contains pieces’ numbers, sub pieces’ numbers and peers’ IPs. The client uses this list to request the missing pieces from the peers in the list. More details about the piece selection algorithm in the section 4.7.1.2.

---

⁵ Holds all peers that should get an additional optimistic unchoke due to the anti snubbing process.
4.5.1.8 Peer handler

The module that is used to keep track of all peers in the client's neighbourhood. It uses its own state which is a record to keep information about peers. These information include the number of peers, their IPs, their choking status, snubbing status, their pieces, etc. This module is used several times during the client's life cycle to get information about the peers and update them.

4.5.1.9 Client interface

This module is used as an interface between the client and its custom functions. When a custom function requires an information about the client, it uses this module to get that information, since custom function does not have the client's states in their modules. If a new custom function has been added to the project (for instance, the end game), the client interface can be used to get the state of the clients.

4.5.1.10 Interesting handler

When peers connect to the swarm, they try to contact other peers to form their neighbourhoods. After the contact, they exchange the information about their pieces with each other. Here, peers get the knowledge about which peers have what pieces. After exchanging their information, peers can find the interesting pieces for themselves. This module is managing this task. When a client sends a handshake to a peer, the client sends its pieces information (after acceptance from the peer) to the peer as well, and when the peer gets the handshake from the client it also sends back its pieces information. Both the client and the peer use this module to check their pieces with each other and then they find the interesting pieces for each other. When a client downloads a piece, it uses this module as well to remove the interesting status from the piece it has downloaded.

4.5.1.11 Runtime environment

The client process starts its processing when the MasterClient process spawns it. It waits for the initial state that has been installed and sent by the MasterClient process. When the client receives its initial state, it heads to set up some extra settings through the help from the extrasettings module. These extra settings include setting up the pieces, sub-pieces and initialize all remaining variables in the
client's state like the trees, lists and flags. Afterwards, the client calls a custom function to initialize its custom state (more about custom states in the section 4.7).

Up to now, the client's state is initialized, thus the client is ready to send the `connect` request via the `networkiface` module to the network, this request includes the link PID and the client PID. The network module sends back to the client a `connect_success` signal as an acknowledgement to make sure that the connection is done successfully. Furthermore, the client asks to get its IP from the network and attempts to locate the tracker via the DNS lookup. Then it will get the IP of the tracker and will start to register itself on the tracker by sending the message `started` to the tracker. When the client gets the success acknowledgement for the registration on the tracker from the network, it will wait and listen to the network until a reply arrives from the tracker. When it gets the message `insert_success` from the tracker, it means the client has been registered on the tracker and it has connected successfully to the swarm. If the client is a seeder, it sends that information to the tracker as well, by sending the signal `completed`.

When the client connects successfully to the swarm, it tries to request a number of peers from the tracker to form its neighbourhood. It uses the neighbourhood custom function (section 4.7.1.7) for requesting peers from the tracker. The tracker, in return, sends back a list of peers' IPs that the client can contact with. The client will check all the IPs that it gets from the tracker and remove all IPs that are already in its neighbourhood, then it constructs a packet which contains a `handshake` concept and sends it to all the remaining peers that are not in its neighbourhood. When the client gets a success acknowledgement for the handshaking message to a particular peer, it initializes a new record in its state (for keeping information about the new peer) and enters the peer IP into its neighbourhood flagged as the unchoked peers. Moreover, the client will build a new binary tree\(^6\) to keep pieces' information for the new peer. Finally, it will construct a new packet that contains all information about its pieces and sends the packet to the new peer.

On the other hand, when the peer (the receiver) receives the handshake message from the client (the sender), it checks the number of the peers it is connected to. If the connected peers are less than the maximum number that the peer is allowed to connect to, which is defined in the client’s config file, the peer constructs a new record for keeping information for the client, and adds the client into its neighbourhood. After accepting the client, the peer creates a new binary tree for the client's pieces and sends its pieces' information to the client. The client and the peer will receive information about their pieces so an exchange of pieces’

\(^6\) The size of the binary tree comes from dividing the file size by the piece size.
information has been achieved between them. During this exchange, the client compares the peer piece’s data with its pieces and marks the desired pieces of the file as interesting pieces, then it keeps these information in the record that is related to that peer. A new packet is sent from the client for informing the other end whether or not it is interested in the peer’s pieces.

### 4.6 Tracker modules

In this section, modules that have been used by the tracker process is explained. The tracker process uses several modules and custom functions to behave as a BitTorrent tracker. The following descriptions explains the modules and functions the tracker is using.

#### 4.6.1 Tracker module

Tracker is the navigation centre for the swarm. It helps all peers in the swarm to find each other and form their neighbourhood. In this simulated environment, the tracker process starts when the routing module spawns it. It starts from the method `init/2` and waits before getting into its normal processing until the PID of the link that connects itself with the network is received. After receiving the PID of the link, the tracker begins to set up its state, which it is working on, by taking all the necessary properties from the configuration files that were sent to it. Afterwards, it initializes all the variables, records and paths to the custom functions. Later, it lets the initial custom function to be called (see section tracker’s custom functions 4.7.2).

The tracker will use the `networkiface` module to try to connect to the network by sending a meta-call message containing a request to connect to the network through the link process (that is received from the network). After getting a success acknowledgement message from the network, it will ask about its IP and then it asks the router to register a DNS name corresponding to the URL of itself given by the torrent file. When every thing goes successfully, then it moves into its life cycle.

#### 4.6.1.1 Life cycle

The tracker’s life cycle is a loop of actions that the tracker does by performing
several functions and various message transmissions. The descriptions below, hopefully, should give a good idea of how the life cycle of the tracker works.

1.1.1 Availability

At every cycle of the tracker's life, the availability of the tracker is checked. It calls the availability module to state whether the tracker is still connected, died or disconnected.

1.1.2 Message transmission

The tracker as from its responsibility checks all messages that are received from the network. These messages could be clients' messages or any other kind of messages. It calls the packet handler module to perform this task.

When the tracker receives a packet, it checks its content to determine the type of the packet. If the packet is a request event, then the tracker looks at its content to handle what kind of the request the tracker has gotten. There are three kinds of requests: started, stopped and completed. If the incoming request is the started, then it means a new peer wants to register on the tracker and joins the swarm. The tracker initializes a new record for this new peer to keep the peer's information, and it will place this record into a specified table (peers' table) in its state. Furthermore, the tracker will reply to this new peer that it has successfully joined the swarm and give it a request interval that the peer needs to wait before sending a new request to the tracker. The purpose of this interval which is called minimum request interval, is to avoid the tracker of getting spammed with too many requests.

If the request is the completed, the tracker will make the peer who has sent this request into a seeder. But when the stopped request is arrived, the tracker easily removes the record that is related to this peer from its state, which means this peer is no longer connected to the swarm. For each request, the tracker uses time handler module to update the request interval to the current time for the peers that have sent requests.

When an announce is received by the tracker, it means the client that has announced is asking for some peers to contact them and add them into its neighbourhood. Therefore, the tracker checks the validity of the client for which if it is registered or not. If so, it checks the last announce time and the last request
interval for this client whether to accept the announce or not. If everything is fine then the tracker accepts the announce and calls the custom function for the neighbourhood to get a list of available peers' IPs for the client, and then these IPs are sent in a packet to the announced sender.

Whenever a packet is sent to the network, the tracker keeps the packet into its state until the packet is received by the network successfully. Tracker also keeps a counter in its state for keeping track of the number of sent packets. It, in addition, keeps the failed packets that have failed to sent to the network in an attempt to check their contents and perform according to the packet's content. Keeping these packets and their contents could be used for logging and statistical purposes as well.

4.6.2 Availability handler

In this module, the availability of the tracker is checked. It states whether the tracker is still connected, died or disconnected. It calls the availability custom function for the tracker to check the current status of the tracker and what it should follow in the next cycle of its life. This module also guarantees that the tracker is not getting a connect signal when it is already connected or a disconnect signal when it is already disconnected. In the current implementation the availability custom function always returns the signal **idle** which means it is still connected, because early on (in the planning stage of this thesis) we agreed that a single tracker is enough for the simulator.

4.6.3 Peer handler

This module keeps track of information about the peers that are in contact with the tracker. These information include peers' IP, last announce time, last request interval, etc. The insertion/deletion of the peers to/from the tracker's table is done in this module as well. The IP address of every clients are saved in the tracker's table in order to give this IP to other peers when they announce. This module uses **ets maps** or **tables** for keeping the information about the peers. Ets maps provide the ability to store very large quantities of data in an Erlang runtime system [5].

4.6.4 time handler

The module that is responsible for setting and updating the announce times and
request intervals for the peers. When peers announce, the tracker uses this module to check their last announce time and request interval whether to accept their messages or not, since when they contact the tracker, they get two arguments from the tracker which are the announce time and the request interval. The tracker uses these arguments to avoid to get spammed with too many announcements.

4.6.5 packet handler

The tracker's packet handler works almost the same way as the client's packet handler. It listens to the network for any incoming packets, gives them their IDs, saves them in the binary tree and forwards them to the particular place for processing. The validity check for the packets that are sent to the network is also done in this module, by checking the ID of the packet with the key of the returned acknowledgement, and reporting the result.

4.6.6 tracker interface

The interface between the tracker and any other functions that are using tracker's state. this module is used by the custom functions when they require any information about the tracker.

4.7 Custom functions

Custom functions, custom behaviours or custom algorithms have all the same meaning. They are set of functions that are necessary for both the clients and the tracker to work properly. All these algorithms can easily be exchanged and used to experiment new ideas to find out how the swarm is affected by them. Some minor changes have been made to these algorithms from the old implementation which granted the current development the ability of reusing them. Followings are the descriptions of each function.

4.7.1 Client's custom functions

In this section, the custom functions that are used by the clients are focused. For each algorithm a brief description on how it is working and what behaviour it is
returning is explained.

4.7.1.1 Initial custom state

Each client in this algorithm, creates a new small state in its main state. This new state is used for various custom functions that is used by the client. It creates a database that keeps information about all the pieces the client is working with. These information include the progress of the pieces, missing pieces' and IPs of the peers the client is allowed to request pieces from. The custom state is also used for checking client's consistency, piece selection and peer selection. The client initializes the custom state after initializing all its variables and functions in its main state.

4.7.1.2 Piece selection

The piece selection custom function is responsible for the clients to select and request pieces from the other peers. There are three available algorithms in the module for how to select pieces from other peers: random selection, ordered selection and rarest first. All algorithms try to find out the missing pieces (in different ways depending on what algorithm the client is using) for the client, and return back a list which contains the piece number, sub-piece number and the peer IP that has the missing pieces.

4.7.1.3 Availability

This function checks the connection of a client whether it is available or not. There are three possible states for the function to return: it could be disconnect, die or idle. When the client gets the disconnect state, it tries to disconnect from the network with the possibility to reconnect to it once more, but when the client gets the state die, then it should disconnect permanently from the network without any chances for the reconnection. When the client gets the idle state, then it returns back into its tasks and completes the remaining sections of the life cycle. The availability custom function can be found in the mod_mymode module.

---

7 The pieces that the client is missing and need to be downloaded from other peers.
4.7.1.4 **Anti snubbing**

This custom algorithm prepares a list of peers that are not downloading any pieces for a long time (usually 60 seconds). For all leechers that have not had the optimistic unchoke, the algorithm checks the duration the last piece of a file was downloaded by the client, if this duration is more than the interval (which was given to the client by the configuration file), the algorithm adds this peer into the list. The anti snubbing algorithm can be found in the module *mod_mymode*.

4.7.1.5 **Optimistic unchoke**

This algorithm returns a peer which needs to get unchoked by the client. It takes all the choked peers from the client's state and selects one of them randomly to get unchoked. The algorithm offers more help for new peers (recently connected to the swarm) by giving them three times more chance of being selected to the optimistic unchoke. The newly connected peers, usually, have poor upload rates, because they do not or have few pieces to share within the swarm and improve their upload rates. Consequently, this will also cause them to have poor download rates because of the tit-for-tat strategy. If the newly connected peers get unchoked more frequently, it will help them to download some pieces and upload them to other peers, so eventually they can improve their upload rates and participate more in the swarm.

4.7.1.6 **Choking**

The choking custom behaviour is needed to perform the choking and unchoking process for peers in the simulated environment. It starts by picking the peers that have not had the optimistic unchoke and are not snubbed. Afterwards, it ranks these peers according to their downloading speeds, uploading speeds and the interesting status. The algorithm will split the ranked peers into two groups (a group to get choked and the other one to get unchoked), then it will return them to the client.

4.7.1.7 **Neighbourhood**

This custom function is used to check whether or not a client needs to add more
peers into its neighbourhood. It checks the number of peers the client is connected to, and decides if the client requires to make an announce to the tracker. The algorithm also returns a certain number of peers the client needs to request.

4.7.2 Tracker’s custom functions

In this section, custom functions or algorithms that are used by the tracker are explained. There are three custom algorithms the tracker is invoking during its life cycle, these functions are custom_state, availability and neighbourhood.

The custom state function for the tracker is set to none in the current implementation but it could be used for statistical purposes in further developments. The availability custom function is responsible for determining the availability status of the tracker. The return value from this functions could be a connect, disconnect, die or idle.

Since the current implementation supports only one single tracker, the availability custom function should not get back to the tracker with the disconnect or die statuses, it is limited to only return the idle for the tracker, which means the tracker should continue on its life cycle.

The neighbourhood custom function is used when the clients are announcing to the tracker. In order for peers to form their neighbourhood or make it bigger, they need to announce to the tracker and get some new IPs. This function collects all peers’ IPs from the tracker state (except the IP of the sender), then it sends back a list of random IPs. The number of IPs in the list depends on the number of the peers the sender has requested.

4.8 Modifications

In order to let the code of the client side interact with the previously developed network module, some alterations were needed. In this section, alterations that have been made to the network module are described.

4.8.1 Modifications to the network

The simulated network in the system consists of a couple of routers and links. Every router is its own process, and the way they run depends on how the network configuration file is written. Links are used to connect nodes with each other. The
general idea of the router is fairly straightforward: it should receive packets from
the links, look up the IP of the receiver, and forward the packets to their
destinations.

Extensive testing was not possible to fit into the previous thesis work (the one
which developed the network side of the project) because of the incompleteness
of the client side code. Incorporating the newly developed client side into the
application has resulted in several bugs. Alterations have been made to these bugs
in order to let the network module operate properly.

Integrating the code of the client side into the application revealed that the
network was not delivering packets from peers to their destinations accurately.
Instead, for each packet, the router was creating two new packets, importing some
parts from the original packet into these new packets, and then forwarding these
two new packets to the recipient, ignoring all other contents of the original packet.
This has been fixed by simply forwarding the original packet to the recipient
without constructing any new packets in the network or affecting other contents
of the original packet (except the changes needed by the network, like changing
the header of the packet).

Whenever a peer sends a packet to the network, an acknowledgement
message for that packet should return to the peer. This is when the packet is not a
meta-call packet. When nodes in the network use meta-call packets to
communicate with each other, they do not deal with the bandwidth issues and
packet loss. Consequently, there is no need for acknowledgement messages for
these kind of packets.

Acknowledgement messages in the network have been modified as well
because the network was sending two new packets as acknowledgements to the
sender and to the receiver. It is a bit odd that the receiver should get an
acknowledgement message containing "package_transfer_success". This has also
been solved by sending only a single acknowledgement message to the sender
node determining the status of the packet. The acknowledgement of a packet is
either a "packet_success" when the packet has been successfully received by the
network (after ensuring the existence of the recipient) or a "packet_transfer_fail"
when the packet fails.

Since clients can disconnect from the network any time they want, a new
feature has been added to the router module to perform this functionality. When a
client disconnects from the network (See 4.5 1.1.1), it sends a disconnect
message to the router it is connected to. The router will receive the disconnect
request and remove this client from its table. But, some new problems came up
regarding this new feature. When the client is disconnected from the router, there
might be some peers in the swarm that are still sending packets to the client
already disconnected. If so, the router tries to find out the recipient but it cannot be found since it is already disconnected from the router and its IP has been removed from the router’s table. Consequently, this causes the router process to crash. A property is added to the router module to handle this situation as well, by changing the header of the original packet to "package transfer fail" and sending this packet back to the original sender without letting the router process crash.

The disconnected clients might reconnect to the network once again. An additional feature has also been added to the router module for this purpose. When a client wants to reconnect to the network, it sends a request to the router for reconnection (See section 4.5 1.1.1). The router, afterwards, gives an IP to the client and inserts it into the router table.

The client PID must exist in the router’s table when the client sends its request to connect to the network, but for the reconnection, the client that was formerly disconnected cannot connect to the router once more because there is no such PID for the client in the router’s table (the PID of the client has been removed during the disconnect process). Consequently, the router process crashes through this procedure. The reconnect function has been added to the router module without using the connect function to reconnect clients to the router. The reason for this is that the design of the connect function does not allow clients to be added in the router’s table during the life cycle while it only allows them during initialization, and this was the main reason for not using the connect function.

Some other minor changes have been made to the router module specially to function handle_metacalls/2 to make the network respond the implementation of new client properly and send the required data accordingly.

4.8.2 Modifications to packet record

Nodes in the system use the so called packets for exchanging their information. The packet had been chosen in the previous development to be a shared record that the different modules would use to make sure that the data are transferred in the same form and that the information is available at any time it is required.

Modifications made to this record were to add three more fields to it: msgtype, msgcon and key. The sender client is the main user of the field msgtype of the record. It uses this field when it gets an acknowledgement signal (success or fail) for a particular packet to know what sort of packet was sent. Afterwards, the sender will prepare itself according to the packet sent. For example, when a client requests some pieces from another one, it checks the field msgtype and changes the status of all requested pieces from requesting to waiting until it gets a
response from the other end.

The field \textit{msgcon} is used only by the recipients (clients or tracker). Whenever a client gets a packet from the network, it checks the header of the received packet to determine the category of the packet. When the client recognizes that the packet is from the peers (not a meta-call packet), the field \textit{msgcon} of the packet is checked and the client decides what to do according to the contents of this field.

The last field added to the packet record was the \textit{key} of the packet. This field is of great importance to both the tracker and the clients. The Key of the packet is a sequential number that is given to each packet from the state of the node.

The key has many functionalities:

- It is used as the ID of the packet the node has sent.
- It is used to check whether or not the packet has been delivered successfully (by checking the key of the sent packet with the key of the acknowledgement signal).
- It is used as indexes for the trees that the tracker and the clients are using to store, check, and delete their packets, or to get them from the trees by using this key.
- It could also be used as a counter for how many packets the node has sent yet.

\subsection{4.8.3 Modifications to Networkinterface}

Due to the implementation of the \textit{pkthandler} module which manages the process of receiving and checking packets, the module \textit{networkiface} has been modified in a way which is only kept for dealing with meta-calls. Both the tracker and the clients use meta-calls to avoid packet loss and bandwidth issues in the network. The nodes in the system use the meta-calls when they want to connect, disconnect, reconnect to the network, get their IPs from the routers, register the tracker on the network, and lookup the address of their DNS. The function \texttt{send_noreply/2} is used to allow the nodes to send their meta-calls to the network via the links which connect them with the router.
5 Conclusion and future work

The implementation of the client side of the project has successfully been developed. A modular design approach was employed and incorporated into the application. All BitTorrent nodes (tracker and clients) run in parallel, and they successfully communicate with each other through the implementation of the new network module. Extensible data structures have been used which facilitate further developments and make the code more understandable to navigate. Large number of options are supported for the simulator, algorithms of the modular structure can easily be used and exchanged to experience new ideas of how the different behaviours of the BitTorrent clients and trackers affect the swarm. The simulator should run smoothly due to the lightweight processes and message passing in Erlang.

There is a wide range of functionalities for the BitTorrent protocol to be supported, many of them could be added to the system. For instance, custom functions for the end game and the download cancellation would make the simulations more accurate and more cleaver. Multi-trackers and the tracker disconnection could be of the further developments, since the current implementation supports a single tracker. Any expansion to the project should not be too hard to add because of the implementation's structure and the modular design of the project.

Data presentation and statistics extraction from only the text in the terminal are quite hard to analyse. It is unfortunate that there was not enough time to implement a user interface for the project. In order to extract statistics and present data from the simulation, a GUI is needed. The google charts API have been used in the original implementation for generating graphs, but they are rather limited, so a more powerful application for generating graphs would make the simulation more convenient.
Bibliography


