

Bittorrent simulator in Erlang

Roland Hybelius
David Viklund



UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet
UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
<http://www.teknat.uu.se/student>

Abstract

Bittorrent simulator in Erlang

Roland Hybelius and David Viklund

This master thesis examined how different behavior of bittorrent clients and trackers affect the simulation in various scenarios such as different network topologies, greedy behavior of downloading clients or choices of algorithms by the downloading clients. The thesis also includes the construction of a network simulator, a modular bittorrent client and a modular bittorrent tracker to support a wide range of options for the bittorrent simulator. These options can be combined to make new scenarios for the simulator to run and results will be presented graphically. Even the algorithms of the modular structure can be easily exchanged to try out new ideas of how things should work to find out how the algorithm behaves under certain circumstances and how it affects the swarm.

Simulations were made to examine behavior when clients had different configurations of the choking algorithm, clients had different sizes of their sub pieces, clients had different distance to the seeder, clients used different piece selection algorithm. In the case of the piece selection algorithms using only a few seeders, the algorithms performed similar to each other which was contrary to our initial expectation that the algorithm rarest first would perform better than the naive order algorithm.

Handledare: Justin Pearson
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 10 005
Tryckt av: Reprocentralen ITC

Abbreviations

Choke: A peer that refuses to upload to another peer.

Choking: When a peer is choked by another peer.

Unchoke: A peer that upload to another peer.

Unchoking: When a peer is unchoked by another peer.

Interested: Another peer that wants a piece from the current peer.

Interesting: When the given peer wants a piece from another peer.

Downloaders: Peers that are unchoked because they are interesting and fast to download from.

Desired downloaders: Peers that are faster to download from then the worst downloader but are not interested.

Choking fibrillation: When peers get choked to quickly.

P2P: Peer to peer.

Bencoding: To structure data in a compact format, usually used for sending light weight messages.

Contents

1	Background	1
2	Problem description	1
2.1	Reading up on Erlang and the Bittorrent protocols	2
2.2	Implementing the simulator	2
2.3	Experiments	2
3	Structure/Documentation	3
3.1	Networkclients	3
3.2	Network	4
3.2.1	Networkmaster	4
3.2.2	Network	4
3.3	User interfaces to the simulator	4
3.4	Logging	4
4	Bittorrent description	5
4.1	Downloading	5
4.1.1	Piece selection	6
4.1.2	Peer selection	6
4.2	Uploading	7
4.2.1	Choking	7
4.2.2	Optimistic unchoking	7
4.2.3	Anti-snubbing	8
4.3	Tracker	8
4.4	Packages	8
4.4.1	Peer to tracker communication	9
4.4.2	Tracker to peer communication	9
4.4.3	P2P communication	9
5	Methods	10
5.1	Tools	10
5.2	Information finding on the Internet	11
5.3	Working environment	11
5.4	Splitting up work	11
6	Simulation results	11
6.1	Piece selection - simulation 1	11
6.2	Choking - simulation 2	12
6.3	Network - simulation 3	13
6.4	Stress test - simulation 4	14
6.5	Individual torrent settings - simulation 5	15
7	Analysis/Discussion	16
8	Problems and solutions	16
9	Related work	19

10 Related literature	20
11 Future work	20
12 Appendices	21
12.1 Appendix A: Program structure	21
12.2 Appendix B: User's manual	21
12.3 Appendix C: Original specification	21

1 Background

Usually, when downloading a file from the Internet, we use protocols such as HTTP, FTP or other server-client protocols. When using those protocols the server is the only source of the file, which implies that the server alone has to upload the file to each and everyone of the downloading clients. This makes the bandwidth of the server an obvious bottle neck when the the amount of clients clients grow. Usually as response to this, additional servers and load balancers are added to increase performance.

Bittorrent, however, is a alternative approach which reduces the demand of bandwidth of the server (which is called the tracker). Bittorrent makes use of the fact that the clients can upload to other clients the parts of the file that they already have downloaded at the same time they are downloading other parts. The tracker itself does not upload any parts of the file thus it's primary task is to help the downloading clients (called leechers) to connect to each other and to the designated uploading clients (called seeders). More on that in section Bittorrent description.

Being a downloading client is not trivial. You will have to have to take decisions when different things occur, as configuration on how to handle other clients you are connected to. When you have many peers, you might not want to upload to all of them simultaneously. Then you need to decide which ones to allow. Also, you have multiple options on what pieces to download, you will have to take a decision of which one to choose, who to download it from and how large chunk to to download. Another thing is when some other client treats you bad, you will need to take a decision on whether or not to punish him for that, and even how to do so. When some one new connects to the swarm, he will have no pieces, then you will have to decide on how to handle such a peer. Will you upload to him to help get him going or will you wait until he has some pieces so that you don't so that you can trade with him?

Anyhow, when using bittorrent instead of the server-client approach, a few different questions arise. How much does bandwidth of the tracker and the downloading clients' affect the total, or individual, results? How many seeders are actually needed and how long do one need to stay and seed? What happens when no one, or just some clients, stays to seed when they are finished? And most important, what happens when different client settings, network topology settings are combined.

Our work was to make a modular simulator to test run different settings and present results of combinations of settings and also to do some minor testing to make sure the simulator actually works.

2 Problem description

We had not used Erlang that much before starting this project nor did we know much about bittorrent. So the first few weeks of the project were reserved to actually learning both Erlang and understanding bittorrent.

After acquiring some general knowledge about the subjects, we quickly realized that this simulator should be divided into 3 larger parts.

A virtual network whose task mainly is to delay transfers depending on the bandwidth capacity and the load of the network that also support different network topology such as different routers and connections between them.

Highly configurable modular network hosts that could act as a bittorrent client and a bittorrent tracker. They should both have a modular behavior and a framework around it that could control the general behavior.

The modular behavior of the clients. It's specified as a module that contained functions for algorithms that has a defined certain set of input data and output for the framework to use when appropriate when appropriate.

However, as it turned out, we also had a fourth large part at the end after extending the thesis work (More on that on problems and solutions). The fourth part was to make it possible to see the results of the simulation graphically, since we realized that just text in a terminal are quite hard to examine.

Graphical User Interface for the simulator to present the results of the simulation and also for debugging. To show graphs of the progress and the clients' and even to monitor how they are doing, real time.

Testing the simulator and see results. The task was to run tests to see what happens when things are enabled/disabled, try out different algorithms for the modular behavior and what happens when things are combined.

Finally, needed to be mentioned. This simulation is not meant to be a complete simulator. The true purpose is just to make the base work to be extended later.

2.1 Reading up on Erlang and the Bittorrent protocols

First off we needed to learn Erlang. For this, we googled a lot and found some random tutorials that we followed. Also, David had attended to a course in Distributed systems which was very useful when learning Erlang. To understand bittorrent we found a very good wikipage where everything was described in detail. Also, we found a specification made by Bram Cohen, creator of bittorrent. More on that in the related literature section.

2.2 Implementing the simulator

The protocol should be bittorrent like and the simulations should generate results. The simulator should be programmable/extendable and support network topologies, different connection speeds, handle large set of clients variations of the bittorrent protocol should include variations of Tit-for-tat, upload to/download from, piece selection.

The requirements for the simulator was quite vague as many things where left open to our decisions.

2.3 Experiments

After implementing the simulator we where to test different settings. Some for network topology, some for algorithms, some for performance and some for settings of the torrent file.

3 Structure/Documentation

The problem was divided into four large parts initially, a network simulator, clients with high modularity, a modular tracker and documentation about the progress and decisions. In the later parts of the project the lack of monitoring and presenting data that was stored emerged. A web interface was added as a fifth large part of the project and it would handle the presentation of data during and after runtime. To keep away confusion all nodes that are connected to the network where give the name networkclient, which includes clients and the tracker.

3.1 Networkclients

A networkclient is something that is connected to the network via a router, it might be a client or a tracker in our case. The task for a networkclient is to simulate sending and receiving transfers over a network. They are however as simplified as possible to be as easy to implement but still giving the desired behavior.

Clients The goal of the clients for the framework to simulate how bittorrent clients behave and load in different behavior algorithms and settings. Client load different modules and settings from a configuration file. There are different algorithms in bittorrent for selecting what part of the file to download first, what other clients to upload to and so on. A user can make their own function for these different algorithms and then specify in the configuration file where the function is and the simulator will load in the algorithm and use it. Other settings could be such as bandwidth up and down, what router it should connect to, how often the different algorithms should be called, etc. The user can load in different algorithms and settings for every single client in the network which makes the approach very modular.

Tracker A tracker's goal is to guide clients to each other so they can create their own neighborhoods in the bigger swarm. Without a tracker there is no way for a client to find each other in the network and no download and uploading can occur. How the tracker chooses to distribute the clients to each other, what bandwidth it has and more is loaded from a configuration file.

Clients usually send information about themselves to the tracker, such as total upload and download, which the tracker sometimes chooses to save. This information can be used to view the overall performance of the swarm which can then be later used for improvements to bittorrent. The default tracker in the simulator stores some data about the swarm, for example how many leechers and seeders there are in the swarm.

Simplified bittorrent protocol Bittorrent uses advanced techniques as Bencoding to make the messages small and hashing for confirmation that the file is not corrupt. In our simulator we use a simplified variant of bittorrent, Bencoding is not needed since the clients can specify how big a message is to the network and hashing is not needed since there can be no corruption of the in file. Another simplification is that the clients have to disconnect in nice manner from the tracker or the system will crash. In overall the simplifications should not have any effect on the simulation result but has the effect that the clients and tracker need to follow the protocol strictly for no crashes to occur.

3.2 Network

The network consist of two processes. The networkmaster and the main network process.

3.2.1 Networkmaster

When the network initially is spawned, it's the networkmaster that is spawned. The role of the networkmaster is to just sit and wait and then spawn the network when the users instructs. Then, when the simulation is completed, it waits once more to spawn the network again.

3.2.2 Network

The first thing the network does when it has initialized it's state is to spawn the masterclient. When this is done we can say that the simulation has started (Network is up and the clients is waiting to spawn).

The network it self is only one process. It receives messages from clients that wants to connect and then adds them to the networks' internal state of the network topology, called the **routersystem**. Then the networkclient's may want to register or look up a DNS which the network has in it's state, the **networkstate**. Later when the networkclients wants to send some **transfer** to each other, the network will store them in a data structure called **transfercons**.

3.3 User interfaces to the simulator

There are two ways to interact with the simulator.

One way is to interact with the simulator is to use the command line. For example, if you type `netl()` in the command line, a brief description of the network state will show. This could come in quite handy when debugging. There are also other commands available, but more on them in Appendix B.

Another way to interact with the simulator is to use the web interface. Here you can see graphs of the last simulation (or the current one if one is running). These graphs are generated from the information in the log files. You can also start and stop the simulation using the web interface. More on the web interface in Appendix B.

3.4 Logging

Both the network and the networkclients have two types of logs. One log is only for debugging, called the debug log. And the other is for storing data that later on can be read to generate graphs. Every networkclient has it's own log and debug log.

Things are logged every second. And to get this synchronized with every process, only the network process knows when it's time to write to the log. When this is the network process will inform all connected networkclient of this and they all will log at the approximate same time.

Note that the debug log could, and should be disabled when running simulations for real and not only for debugging. If it's not disabled the massive writing to files will make the simulator decrease heavily in performance.

4 Bittorrent description

Bittorrent is a distributed file sharing protocol used for downloading and sharing information over a network. When a small part of the file is downloaded it is shared to others immediately, this makes all clients in the network contribute.

A client that wants to download the file needs to connect to a tracker, when a client is connected to the tracker we call it a peer. All peers that are connected to the tracker form a swarm. The goal of Bittorrent is to have the overall swarm speed as high as possible, both up and down. To make peers upload, Bittorrent uses tit-for-tat sharing, if a peer uploads a part it should get something back. This is performed with different algorithms that will be gone through in later sections.

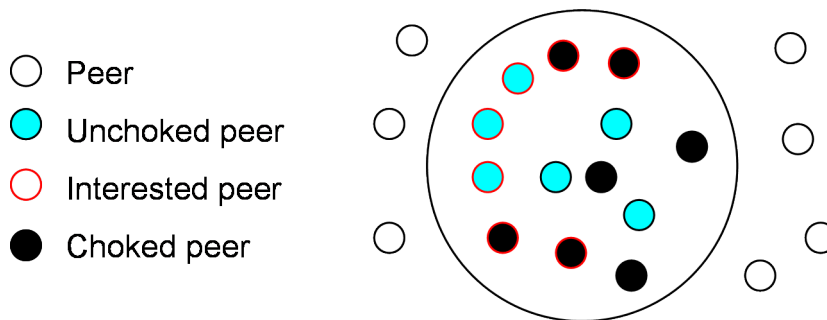


Figure 1: A neighborhood for a peer

Swarms grows very large in many cases due to a lot of clients connecting to it wanting to download the file. Keeping overhead over all the peers in the swarm would slow down a peer and only a small part of the swarm would be of use to it. Dealing with this Bittorrent uses neighborhoods, see figure 1.

A peer has no information about other peers in the swarm when it first connects to the tracker. There are two scenarios for a peer to get in contact with another peer. The peer can announce to the tracker that it needs more peers and the tracker will respond by replying with a set of peers. A peer can get contacted by another peer who has sent its own announce to the tracker as well. All peers that a peer has contacted in these two approaches are called neighbors, together all the neighbors from a neighborhood. All peers have their own unique neighborhoods.

Seeders and leechers are two terms that describes the state of the peer. Leecher is a peer that does not have the complete file and is currently downloading and uploading parts of it. A seeder however has the complete file and is not interested in downloading anymore but still uploads.

4.1 Downloading

To make use of Bittorrent, a torrent file is needed. In the torrent file information is stored about the file's length, URL to the tracker, hash codes and pieces. Hash codes are used for verifying that a piece that completed is correct and not corrupt. The file is divided into

pieces, the amount of pieces depends on the piece size and how large the file is. A peer furthermore divides the pieces into sub pieces, this is not specified in the torrent file so the size differs depending on the client. Sub pieces are used for the peer internal structure to keep track of what part of a piece it misses to get the complete piece. When a peer requests a part of a piece from another peer it translate the sub piece number to a range and requests the range instead. This must be done since the sub piece is an internal number and the other peer does not know anything about your sub pieces. When a piece is completed the peer sends out the information to its neighborhood that it has the following piece now. The peer can now share the piece with its neighbors and contribute to the swarm.

4.1.1 Piece selection

A peer selects what piece to start downloading next with a piece selection algorithm. A good piece selection algorithm is not only important for the single peer and its neighborhood, but also for the whole swarm. There are three standard algorithms that piece selection uses to make the selection.

Random chooses a piece at random to download next. The algorithm is fast and will in the long run make sure that pieces does not disappear from the swarm. In practice it is most commonly used in the beginning of a peers life when it has zero pieces. If a peer does not have any pieces it can't contribute to the swarm, so getting at least one complete piece is critical.

Ordered first takes the pieces in order, for example piece one, piece two, piece three and so on. A few number of peers that runs this algorithm won't affect the swarm as hole. Bigger numbers would result in higher number pieces disappearing as the seeders goes offline. The reason for this is for example when every leecher has downloaded the first five pieces of a ten pieces file. If the seeders then go down everyone peer would have the five first pieces but no one would have the five last pieces. Ordered first is rarely used in present time because of this shortcoming.

Rarest first searches for the rarest pieces in its neighborhood and chooses one of them to download next. Rarest first prevents pieces from disappearing from the swarm and give good performance for a peer.

It prevents pieces from disappearing because pieces that are not downloaded becomes more rare for each other piece that completes. When the piece becomes rare enough the peers will start to download the piece until it becomes more common. The algorithm is highly used after a peer has one complete piece.

4.1.2 Peer selection

When the piece to download is selected a peer to download it from is needed. Bittorrent uses a very simple algorithm to decide what peer to download it from. Any random peer that has the piece not choking the current peer is a valid target. A static queue of five sub pieces are queued up on the peer chosen. The peer queues up requests for different pieces on all the peers it can download from in the neighborhood.

End game is an exception to the usual behavior of peer selection. When a peer only misses a couple of pieces it requests the pieces from every peer in the neighborhood at the same time. When the pieces are completed it aborts the other requests that did not complete. The last pieces a peer requires to complete the file are usually the hard ones to get. Using drastic measures like end game shortens the finishing time.

4.2 Uploading

In its neighborhood a peer usually have more peers then it would desire to upload to simultaneously. Choosing which peers to upload to is done by the following algorithms, anti-snubbing, choking and optimistic unchoking. Anti-snubbing and optimistic unchoking are special cases, the choking algorithm decides in the common case.

4.2.1 Choking

The choking algorithm is performed every ten seconds and it decides which peers to unchoke and which to choke. It is called every ten seconds to avoid choking fibrillation. A peer that gets choked and unchoked too often won't get a chance to show its actual upload speed because it gets choked too quickly again. The upload and download speed to a peer is saved over a 20 seconds interval. Saving the speed for a 20 seconds interval gives a more accurate speed for the present. This makes sure that a peer who had a very high speed when it connected, later on drops in speed and becomes slow does not get chosen over a new peer with high speed for the time being.

Choking can be done in numerous ways, the most basic strategy is to have two groups, one downloaders group, one desired downloaders groups and the rest gets choked.

Downloaders Downloaders is a group of peers with high upload speed that are interested in one or more pieces a peer has. A peer commonly wants 4-6 downloaders so it takes the peers with the highest upload speed that are interested and unchokes them. The rest of the interested peers will get choked until either a peer needs another downloader or the peer gets better upload speed then the worst downloader.

Desired downloaders Some peers might have better upload speed than the worst downloader but are not interested in the peer. This would typically be a seeder who has all the pieces but is still connected to the swarm. They are called desired downloaders and are unchoked. If the upload speed for a desired downloader goes below the worst downloader it gets choked. But if the desired downloader becomes interested in the peer the downloader with the worst upload speed gets choked and the desired downloader is added to the downloaders group.

Seeders uses the same choking algorithm as the leechers but for one exception. Unlike the leechers who unchokes the peers with the best upload speed the seeders unchokes the leechers with the best download speed.

4.2.2 Optimistic unchoking

If the whole swarm on the network uses the choking algorithm above it will be difficult to find better downloaders then the current ones. A peer only unchokes someone that it can

download faster from than the current worst downloader, the same goes for the other peers. If only the choking algorithm applied, then peers would have to hope for new peers, seeders or peers which does not get to download from anyone so their neighbors are all equally bad.

Optimistic unchoking is an algorithm that takes care of this phenomena. The algorithm runs every thirty seconds and takes one peer at random from the neighborhood and unchokes it for thirty seconds. Peers that are new to the neighborhood are three times more likely to get picked by the optimistic unchoking algorithm to get them started.

4.2.3 Anti-snubbing

Optimistic unchoking makes sure a peer does not get stuck with bad performing downloaders, it always searches for better downloaders. In the normal case this is enough, but in some rare cases a peer might have several bad performing downloaders. Exchanging these downloaders with the optimistic unchoke algorithm would take quite some time since it works over a thirty seconds time interval.

Anti-snubbing exists to speed up this recovery process. If a peer tries to download a sub piece from another peer that it has unchoked and does not get in sixty seconds it will consider itself snubbed. The peer that is snubbing gets choked and is not uploaded to, except as an optimistic unchoke maybe, for sixty seconds. When a peer gets choked for snubbing an additional optimistic unchoke is performed. For example if four peers are choked because of snubbing, then there will be five optimistic unchokes. One default optimistic unchoke and four more because of ant-snubbing.

4.3 Tracker

The tracker is a navigation center for the peers. Without it the peers can't find each other and if peers can't form neighborhoods they have no one to download from. A peer connects to the tracker and the tracker saves away information about the peer. Later on when a peer requests peers for its neighborhood the tracker returns a list of IPs that the peer could try and contact for neighbors. Trackers in most cases save away some additional information so swarm speed and other statistics can be calculated.

A message from a peer is called a request and when a peer requests for new neighbors it is called announce. To avoid getting spammed with requests and announces the tracker have two parameters, last request interval and minimum announce time. Last request interval is the time a peer needs to wait before sending another request to the tracker, if a request is sent before the time has passed it will be denied. Sending a list of peers to a peer is heavier than sending a normal request, some trackers allows sending normal requests more often then requests with announce. Last announce time is the time the peer has to wait until it is allowed to send a request with an announce in it.

4.4 Packages

The structure of a package depends on if its a tracker talking to a peer, a peer talking to the tracker or a peer talking to another peer. In these following lists the implemented options are marked with a ✓. Whereas those options that does not have a ✓ is not implemented at all or partially implemented.

Bittorrent uses Bencoding to keep the overhead small for packages.

4.4.1 Peer to tracker communication

A request to the tracker from a peer can include the following parameters:

- ✓ Peer_ID: The ID for the peer sending the request.
- Port: The port the peer is listening on.
- Uploaded: Total amount uploaded.
- Downloaded: Total amount downloaded.
- Left: The amount needed to complete.
- ✓ Event: Must include either started stopped or completed.
 - ✓ Started: Is sent when sending the first request.
 - ✓ Stopped: Is sent if the peer wants to disconnect gracefully.
 - ✓ Completed: Is sent when the peer becomes a seeder.
- ✓ IP: The IP for the peer, not required.
- ✓ Numwant: The number of peers a peer is requesting for. Not all trackers support it.

4.4.2 Tracker to peer communication

The tracker then responds with the following information:

- ✓ Interval: The minimum time a peer should wait after sending a request before it gets to send a new request.
- ✓ Min interval: The minimum time a peer should wait after requesting peers from the tracker before it requests more peers. Not all trackers support this.
- ✓ Peers: Information about peers that the peer can connect to and try to add to his neighborhood.
 - ✓ IP: The IP for the peer.
 - Port: The port this peer is listening to.

4.4.3 P2P communication

Messages peers can send to each other are:

- ✓ Keep-alive: A ping message so a peer knows its neighbors have not disconnected.
- ✓ Choke: Sender refuses to upload to the receiver.
- ✓ Unchoke: Sender accepts uploading to the receiver.
- ✓ Interested: Sender is interested in one or more pieces the receiver has.
- ✓ Not interested: Sender is not interested in any pieces the receiver has.

- ✓ Have: Sent to all neighbors informing them of a new piece the current peer completed.
- ✓ Bitfield: An optional message which informs other peers of all pieces the current peer has.
- ✓ Request: The message peers use to request pieces from other peers in the form of a range.
- ✓ Piece: A small part of the file that is being downloaded.
- Cancel: Cancels requests, used in end game.

5 Methods

During the first part of the project we used a whiteboard to help during planing and discussions. When a larger problems or questions occurred a discussion was held in front of the whiteboard where everything said got noted down onto it. We stored the information drawn by taking pictures of the whiteboard and committed them to our SVN server. The information was reused and altered frequently, the pictures made it easy to keep track of the progressing work. Information on the whiteboard was often drawn in the form of coloured graphs, making them easy to read and understand, and the pictures enabled a log of what had been noted down on the whiteboard even after it had been erased.

5.1 Tools

Different tools where used to code the simulator, run the simulator and document the work and decisions.

The simulator is coded in pure Erlang language, making it parallel to some extent. Erlang message passing and the built in parallelism was easy to use and effective. We felt that the default built in error handling is a drawback in Erlang. The error messages it produced where hard to interpret.

SVN was the tool we used to synchronize the work with each other. Problems that occurred when using SVN where date errors when committing images and wrong revision number when committing files.

Making the simulator easier to boot up was done with bash scripts. The script handled clearing away files, searching for configuration files, etc. Our opinion is that bash is very powerful language, but the syntax is hard to read and understand.

Emacs and Vim are the two text editors used to code the project in. They both have good encoding support for Erlang and is very customizable. There are a lot of built in commands which makes coding easier.

Make in combination with erlc are the tools used for compiling the project. Only things needed to be recompiled will be compiled again to optimize compile time. A proper Makefile will compile larger projects consisting of many files.

Documenting the decisions we made, the structure of the simulator, manuals for the simulator and making this report where done in \LaTeX . Latex is slow to write in but makes a good structure and readability for the document.

Yaws is a web server which can easily be integrated into Erlang projects. we use yaws in embedded mode to show the web interface and it turned out to be very convenient once we understood that we could use ehtml to avoid string handling as long as possible.

5.2 Information finding on the Internet

The Amount of information about the bittorrent protocol is very limited. Most of the information we found came from the Internet, with help from google-scholar, wiki pages and other similar reports. For more general information, such as what web server to use, we consulted our supervisor Justin Pearson for ideas.

We used the Linux manual pages, erl manual pages and searching on google to get help for the different tools that where used the project.

5.3 Working environment

During the first part of the thesis our office was a large project room at Uppsala university, at Pollacksbacken. It allowed access to a huge whiteboard, a lot of free space and a calm working environment. For the second part of the project when the project got extended we where moved to another office in the same building. The office was large but crowded and did not contain a whiteboard. To enhance the new working environment post-its, large notebooks and colored pens where acquired. We furthermore acquired big monitors to help with the development of the new web interface.

5.4 Splitting up work

At the start of the project we decided to split up the work in two parts. The first part being the network, which was assigned to David. The other part being the different clients that where going to live in the network, this was assigned to Roland. With the work split up we decided to still work together as a team, coding at the same location and making large decision together.

6 Simulation results

We performed a lot of simulations during this project. Here, we present the ones that we found to be the most interesting.

6.1 Piece selection - simulation 1

To try out how different piece selection algorithms affect the swarm we made 5 simulations. Here we will show the most relevant facts about these simulations.

The most interesting simulation we did about piece selection is a test where there are 33 clients using ordered piece selection. 33 clients using random first piece selection and 33 clients using rarest first piece selection. There are also 20 seeders.

Results In Figure 2 we will see the piece-graph by time of the simulation.

The colors in figure 2 are defined as follows

- Red: Average of the 20 initial seeders.
- Yellow: Average of the 33 Rarest first clients.

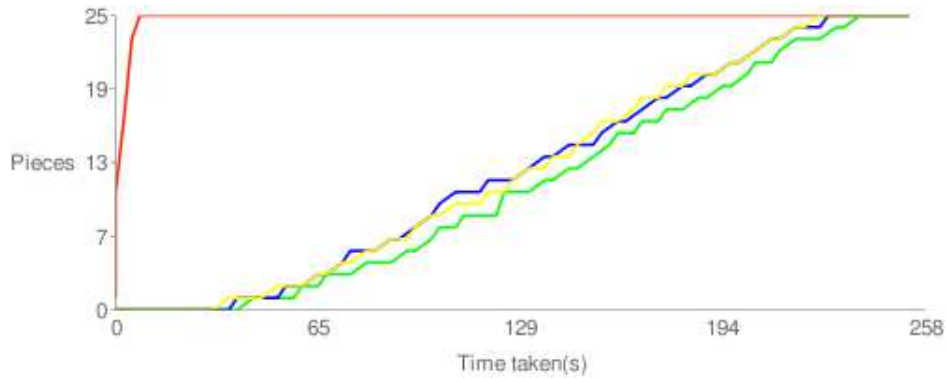


Figure 2: The pieces the clients have

- Blue: Average of the 33 Random clients.
- Green: Average of the 33 Order clients.

Analysis We can see that every client is downloading at the approximately same pace. This is because all clients download any piece that they see. So the actual algorithm does not matter much when there is only one to choose from. Also note that the clients using Average order is slightly worse than the others. This may be due to the fact that those clients will get pieces that they cannot share that easily.

Other simulations We also tried out the same simulation but with only one seeder. That resulted in pretty much the same graph but the total time needed for everyone to download the entire file was doubled.

6.2 Choking - simulation 2

When we made simulations on what choking really does, we mainly tested what happens when someone is greedy. So mainly one test is particularly interesting. We have a normal swarm with generic leechers which implements tit-for-tat behavior. That is, they upload to the peers which they can download from. However, one leecher is greedy. He does not allow anyone to upload from him but he tries to download as much as he can. More technically, he always chokes everyone of his peers.

Results In Figure 3 we will see the piece-graph by time of the simulation.

The colors in figure 3 are defined as follows

- Red: The generic clients which implements tit-for-tat.
- Green: The one greedy peer which does not upload.

Analysis As we can see, the generic clients punishes the greedy client by not uploading to him. This results in him being bullied by everyone and makes his progress slower than the others.

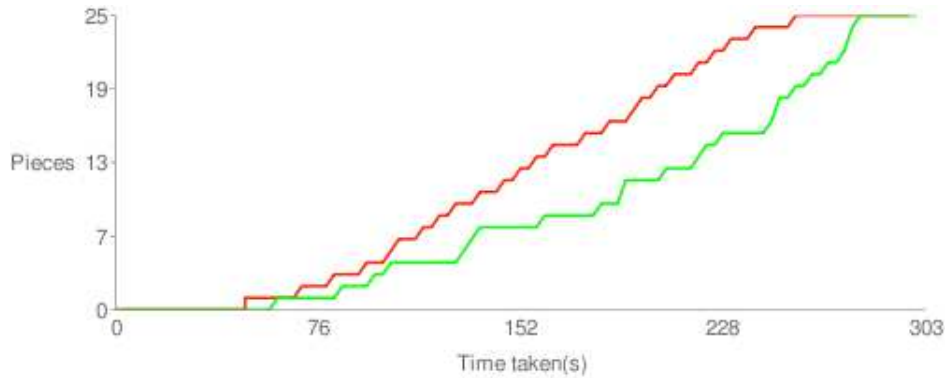


Figure 3: The pieces the clients have

Other simulations When testing choking algorithms we also made some tests about optimistic unchoke. However, we could not really see any difference on why to use optimistic unchoke since we used a good network with all clients being equally good. Optimistic unchoke main purpose is after all to find better peers and get new peers going. Also, we did tests to see what happens when all peers are greedy. That test resulted in not every peer finished. That was because only those peers that initially connected to the seeder finished, while the others did not even get a single piece.

6.3 Network - simulation 3

When we tested the network we mainly focused on convincing the reader that the network actually works. So, the configuration of the most relevant simulations follows. We have two routers. One fast router, with 80 clients and also 10 seeders are connected to this router. The other router, the slow router, can only receive and send to the fast router with the capacity of 5 clients (speed 150/150) but there are 20 clients connected to it.

Results In Figure 4 we will see the piece-graph by time of the simulation.

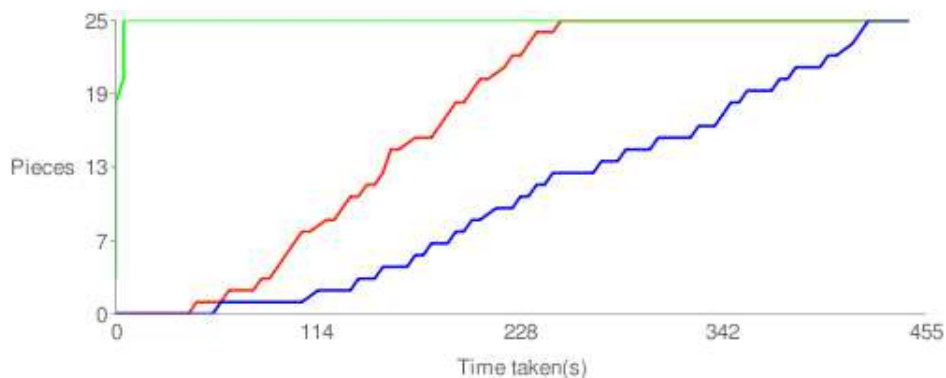


Figure 4: The pieces the clients have

The colors in figure 4 are defined as follows

- Green: The seeders which are connected to the fast router.
- Red: The clients connected to the fast router
- Blue: The clients connected to the slow router.

Analysis We can see that the faster clients complete before the slow clients which was expected. However, When the clients on the slow router starts to get pieces, the speed should increase since they have fast connections between each other. Our explanation to this is that the clients on the fast routers become very generous when they become seeders so that the slow clients will prefer to download from them.

Other simulations We also did some simulations to try out clouds, which may be more realistic. However, since there's a bug with in the network which makes the disconnect very slow, the amount of skips in the network shoots sky-high, which decreases the accuracy of the simulation and thus, makes it unusable.

6.4 Stress test - simulation 4

To see how many clients which are reasonable, we performed a simulation in which clients join until the simulator crashes. In this test the computer we used are relevant. The computer we used is a Intel Celeron M 1.66GHz with 512MB RAM and 1MB cache running Archlinux with kernel Linux-2.6.30-ARCH.

Results In Figure 5 we will see the the amount of skips in the network by time. Note that 10 clients connects every 10 seconds.

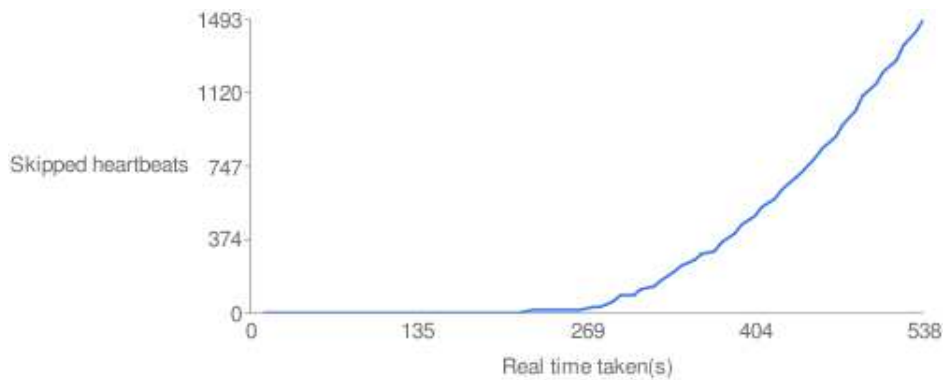


Figure 5: Stress test - skipped heartbeats

The colors in figure 5 are defined as follows

- Blue: The total amount of skipped heartbeats.

Analysis We see that the heartbeat skipping starts around 200 seconds when there are 200 clients. Then it starts to grow exponentially. Also, the amount of packages that's on the network will be at it's maximum at around 160 seconds and then it will slowly decrease. This may be due to the lack of execution time of the client processes. They don't get enough time to register the packages that they want to send.

Other simulations We also ran some simulations to see what heartbeat interval that would be appropriate for this system. A low value, will increase the accuracy of the simulation as long as the network does not have to skip heartbeats too much. Whereas a high value will not use the computation capacity of the machine. A value like 95% is a good choice, since it more or less represents what the system can handle.

We ran different simulations with different heartbeat intervals in 10000 heartbeats to examine where the simulator skip too much.

Heartbeat interval	Percent skips	Comment
10ms	45.0497%	Very inaccurate
20ms	78.8273%	A bit inaccurate
27ms	96.6557%	Good
35ms	99.8797%	Good
50ms	100%	Good, but the simulation could run faster

6.5 Individual torrent settings - simulation 5

Another thing that could alter the performance of the bittorrent clients are the sizes they choose for their sub pieces. The total size to download and the amount of pieces is specified in the torrent file, but the size of the sub pieces are individual for each client. So we decided to run a simulations with many clients that have different sizes of their sub pieces.

Results In Figure 6 the different clients with different settings for their sub pieces are shown.

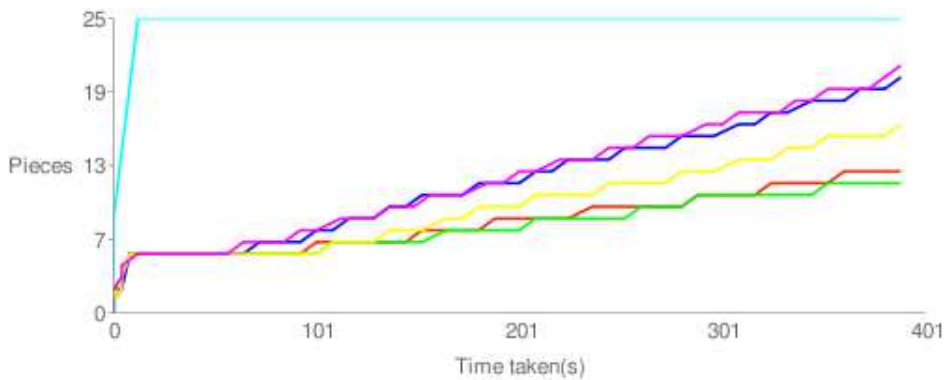


Figure 6: Clients with different sizes of their sub pieces

The colors in figure 6 are defined as follows

- Red: Clients with 1000 bytes of their sub pieces

- Blue: Clients with 5000 bytes of their sub pieces
- Magenta: Clients with 17000 bytes of their sub pieces
- Yellow: Clients with 50000 bytes of their sub pieces
- Green: Clients with 100000 bytes of their sub pieces

Analysis Even though everyone have similar settings we can see that even then there are some differences in performance. Green (sub piece size: 100000) which has very large sub pieces have to download more than the others when they are choked. They loose more data every time they're choked. Smaller sub pieces does not really differ much, but too small (Red, sub piece size: 1000) have too many pieces to download so meta-messages gets to be a too large part of all total messages.

7 Analysis/Discussion

The thesis took longer time than expected due to our lack of estimation experience of a work of this size. If we where to cut down on contents in the work we would increase our odds of being finished on time.

We did not use any hard deadlines, which probably was the main reason for the fact that we went overdue. Nor did we have that much feedback during the work as we would get if we were to use an iterative development method. Also that would perhaps helped us to keep deadlines.

The choice of a serial network (See Serial Network under Problems and solutions) was perhaps not the most elegant, but at least it was completed and it works okay though is does not scale very well.

To make a totally modular bittorrent client means many things. Our interpretation of this was to have the six custom functions and the 22 options in the configuration file. Of course, there are many more things that could be modularized, but for this work, we decided that this was enough. Perhaps it was even too much.

During the time that we wrote the report, we also applied for jobs. This also led to many days going away since we had to go to interviews with different companies.

We chose to make our own simulator. Maybe that was a bad idea since if we where to use an existing one we would have gotten many things for free.

8 Problems and solutions

Here's the largest problems we've had during the project During the project there where a lot of problems and solutions to the problem. When considering a solution we had to make sure it would not take to much time to implement, it was a correct solution and how important it was to correct the problem.

Serial Network

Problem: We had to take a decision on what to make and what not to make. The largest issue here was to make the network serial or parallel.

Solution: We decided to use a serial network due to many considerations. First off a parallel

network takes longer time to implement and since we had to learn Erlang and bittorrent a serial network was good enough. Also in the beginning of the project we had even less time since the initial idea was to make a 10 week project.

Bachelor or master thesis

Problem: After working on the project for 8 weeks, we realized that 10 weeks could be enough, but if we did not continue working on the thesis, it would not be very good and there would be no completed tests. So we had to take a decision on either to complete the work, with no nice way of showing things, or extend the work to a master thesis, 10 weeks more.

Solution: We decided to extend the work. This was a good decision in the sense of quality of the product, but it took a lot of extra time. But we are satisfied with the results and do not regret this decision.

Slow network disconnect

Problem: There's a bug when client disconnects. The network becomes very slow for some reason. We don't know what this may be due to though since it is not critical for the working simulator.

Solution: We did nothing. The bug remains.

Client settings

Problem: Should the clients have the same settings?

Solution: A config file with a section specifying the default behavior of a client. If a different behavior is desired a new section can be created, this section will shadow the default behavior with the given new behavior.

Geographical areas

Problem: If our simulator should support geographical areas. To have slower connections between some clients then others.

Solution: The network uses fake routers to simulate geographical areas.

Extra support for routers

Problem: In our first design of the network, we had no way of expressing more complex network topologies. After a request from our supervisor, we should implement it.

Solution: We implemented support for different routers in the network, but the network is still only one process.

OTP

Problem: Would the time it would take to learn OTP repay itself?

Solution: At first we decided OTP would take too much time to learn, so we chose not to use it. If we had known from the beginning that the thesis would be extended to 20 weeks instead we would definitely have used OTP.

Static routing vs dynamic routing

Problem: Should we use static routing or dynamic routing in the network? How shall we decide on what router to redirect packages to if there's multiple options?

Solution: Due to shortage of time, we implemented the simple way of having routing tables that are static. The user must specify these tables manually.

Yaws query-var bug

Problem: There was a bug in the yaws implementation or documentation. The function `queryvar` returns a tuple like `{Val1, Val2, Val3}` when the specification states that it should return a list like `[{ok, Val1}, {ok, Val2}, {ok, Val3}]`.

Solution: We ignored the documentation and used the values actually returned. If this bug is fixed in latter yaws-versions, a problem will arise here.

Many clients in graph

Problem: When there's multiple dots on a google chart API graph, only some are shown (Since there's a limit on how many character that can be sent as a GET-request to a web server. So to show many clients in one graph in the web interface, only a few coordinates per client are possible. If this number is low, then the graph does not really say much.

Solution: We did not solve this problem. But a solution could be to use another application for showing graphs, or to generate many graphs and then merge them by some image manipulation program.

Speedhandler location

Problem: The clients' needed to know the speed of their ongoing transfers, but they don't know anything about them until they're completed or failed.

Solution: The network tells the clients every second how much they have transferred (in and out) to every other client since the last update.

Clients send much information to web interface

Problem: When the web interface asks for information of the clients, the clients sends a lot of extra information. Even information that is not needed. This may make the clients slower if the web page is refreshed a lot.

Solution: Don't update the clients web page a lot.

Error rates

Problem: Should packages fail to be sent because of an error?

Solution: Due to shortage of time this requirement was dropped. Package loss for a long time can be simulated by lowering the speed of the connection.

Multiple incoming messages

Problem: If a client gets many incoming messages that it has to respond to, it must answer back to the network in the order it received them. If not, the network will crash.

Solution: The clients answers in the order it received the messages.

Max upload connections unused

Problem: In our current implementation the framework of the clients does not respect the max upload connections.

Solution: If a user wants to have a max upload connections, he or she could instead adjust how the choking custom-function works.

Piece and sub piece sizes

Problem: The size of the last piece and sub piece might differ from the other pieces.

Solution: The last piece can be big enough to make a difference so it is calculated correctly. Sub pieces are not since they are so small that it wont effect the simulation. Pieces and sub pieces are explained in chapter 4. In other words, make sure manually that the size of the file is dividable by the total amount of sub pieces.

API not complete

Problem: There are many things supported in the client state that the custom functions have no way of getting.

Solution: Make more functions in the API of the client state, `clientiface.erl`.

Masterclient structure

Problem: `Masterclient` was one of the first modules created. Later in the project we noticed that it was not as modular as we wished for it to be. It was hard to add new settings in the config file and remove old settings.

Solution: Refactoring and rewriting big parts of the `masterclient` module. Changes to the client state was done as well to give more modularity.

Tracker offline

Problem: The tracker may not disappear from the network since clients assume tracker to exist.

Solution: When the tracker decides to go offline, it starts ignoring `numwant` messages, but it works as normal otherwise.

9 Related work

GPS: A General Peer-to-Peer Simulator and its Use for Modeling BitTorrent **Weishuai Yang and Nael Abu-Ghazaleh** **Department of Computer Science, Binghamton University**

This work is focused more on the P2P simulator rather than making a actual bittorrent client. Their implementation, just as our, use the idea of transferring messages instead of transferring every packet by it self. They have also implemented choking, anti-snubbing and piece selection. They also investigate real time and virtual time for their simulation to see how it performs. However it does not seem that they have made any simulations to try how choking and piece selection etc, works together or how it affects the results of the leechers.

10 Related literature

Incentives Build Robustness in BitTorrent by Bram Cohen.

Bittorrent Protocol Specification.

Concurrent Programming in ERLANG second edition by Joe Armstrong, Robert Virdin, Claes Wikström and Mike Williams.

Erlang - Concurrent Functional Programming for Telecommunications. A Case Study of Technology Introduction.

Programming Erlang by Joe Armstrong.

Erlang Programming Language, Official Website - <http://www.erlang.org/>

11 Future work

- Defensive code against user errors in files Test-suite to check that everything is okay Something that forces custom functions to be `mod_` in the beginning of their name so that they don't collide with other existing modules...
- Parallelized network.
- Profiling and optimization.
- Real client.
- Super seeding.
- Custom behavior to be able to cancel downloads.
- State and data structures. The state in both the network and the networkclient consist partially of large tuples with macros to access them. Records would be more appropriate for this.
- Tracker disconnect.
- Faster Disconnect in network.
- Custom function that actually uses the end game support.
- Error rates on connections.
- Time when peers got choked/unchoked.
- Initializing sending a message should take some extra time (since that's what happens IRL).
- Multiple connection penalty in network.
- Config files should be forced to be in `config/`.
- Tracker can disconnect for real.

- Auto compile custom-modules at runtime.
- Client-Tracker communication-protocol should support queries about total seeders/leechers, swarm speed, etc
- Seeder support - support in framework for super seeding and similar.
- Make the code more efficient by updating old functions to use new redundant data. Such as listing different peers depending on a stat.
- Use some data structure for State instead of a tuple which state is changed with macros. Also, we use a lot of `gb_trees`, perhaps other data structures would be more suitable at some places.
- Use `ets` for message handling and what pieces a client has instead of `gb_trees`. It's more powerful with the `match` and `select` functions.
- Messages sizes should be defined in the config files instead of a `hrl`-file.

12 Appendices

12.1 Appendix A: Program structure

This is the structure of our solution of the simulator.

12.2 Appendix B: User's manual

How the ordinary user uses the shell, config files and web interface.

12.3 Appendix C: Original specification

The original specification of the thesis.

Appendix A: Program Structure

Roland Hybelius David Viklund

February 18, 2010

Contents

1	Runtime environment	1
1.1	Directory hierarchy	1
1.2	The boot-up	1
2	Network	2
2.1	Transferpicker	3
2.1.1	Example	3
2.2	Network tick	10
2.3	Speed handling	10
2.4	Networkstate	10
2.4.1	Router system	11
2.4.2	Transfers	13
3	Bittorrent Client	14
3.1	Master Client	14
3.2	State	15
3.3	Framework	22
3.4	Life cycle	23
3.5	Peers	27
3.6	Pieces	29
3.7	Package communication	30
3.8	Custom Behavior / Algorithms	31
4	Process communication	32
4.1	Messages sent to the network	32
4.2	Messages sent to the networkclients	33

1 Runtime environment

1.1 Directory hierarchy

```
documentroot
|-- documentation
|-- spec
'-- src
    |-- config
    |-- debuglog
    |-- framework
    |   |-- network
    |   |-- networkclient
    |   |   |-- client
    |   |   '-- tracker
    |   '-- shared
    |-- log
    |-- mymodules
    |   '-- help\_functions
    |-- mymodules\_objs
    |   '-- help\_functions
    |-- objs
    |   |-- network
    |   |-- networkclient
    |   |   |-- client
    |   |   '-- tracker
    |   |-- shared
    |   '-- webiface
    |-- simulations
    |-- webiface
    '-- webroot
```

1.2 The boot-up

The first thing that starts up is the main-module. It will start up the the yaws embedded mode process, process the random process, the **network master** process. The **network master** process will then spawn the network using the network config file.

The first thing the **network** does is to spawn the **tracker** and then to spawn the **masterclient** which will spawn the **clients** using the **client config file** more on that in the section 3.1 about the **Masterclient**.

When the simulation is finished or stopped, the **network master** can spawn the network again manually using the **web interface** or the Erlang debug interface.

2 Network

The role of the network is to add a small time delay between messages that are sent until they are completed. The `networkclients` (The hosts connected to the simulated network), only communicates by exchanging Erlang messages with the network process. Once connected they will be given an IP-address from the network, which will work as a identifier on the network. The `networkclient` may also register a DNS address to allow other hosts to find out what IP-address this host has. So, whenever this host is to send messages over the simulated network, it will have to specify the IP-address of the other host to the network. The network will then create a transfer package and send out a request to the other host to transfer it. The other host can then either accept it or deny it given the sender's IP and the header of the message. If the message is accepted, it will be placed in a queue with all messages that are transferred over the network at the moment. The message will then wait to be picked by the `transferpicker` to be progressed and eventually completed. When it's completed the receiving host will get the actual contents of the message, also called the body of the message. The amount of time taken to send the message depends on a number of different things described below.

Low bandwidth bottleneck When the route of a transfer contain one or many low-bandwidth connections, the time taken to transfer messages will increase, since every time it gets picked by the `transferpicker`, it will be able to transfer the amount of bytes that the bottleneck can support.

High traffic When multiple packages share some connection path, they also share the bandwidth between the two routers in the path. When one of them is picked by the `transferpicker`, the capacity of the router will decrease during that `heartbeat` which may reducing the bottleneck for other transfers traveling though the same route.

Large packages Large packages needs to be picked multiple times by the `transferpicker` in order to be transferred completely.

Flooding the network process When many messages are sent to the message process, it will only handle them once at the time, and any `heartbeat` messages could be placed after a bunch of other messages which will slow down the network, by forcing it to skip. This should not really be much of an issue, since only a limited amount of messages are sent from the `networkclients`, and sending messages should really be fast between the Erlang nodes. If many `networkclients` sends messages at the same time for some reason, it may happen though.

2.1 Transferpicker

The `Transferpicker` is the part of the network module that progresses the confirmed/pending transfers.

Every time the `transferpicker` is called it will make a copy of the current router system and send transfer over it until all transfer queues are processed. As the transfers are processed, the capacity of the routes the transfers takes will decrease.

The `transferpicker` will then traverse and progress, in random order, all pending transfers.

When a transfer is traversed the head element will be transferred as much as possible on the current copy of the router system. The `transferpicker` will increase the `progress` (a.k.a `to_send`) of the head element of the transfer queue. The amount increased depends on the bottleneck of the path on the router system.

If the progress reaches the `total_to_send` the transfer is completed and the sender and receiver will be informed of this. The receiver will also be informed of the contents of the transfer. Also, if there's still capacity left and other transfers in the picked queue, the next transfer will also be progressed until the capacity of the path is reduced to 0 or there is no other transfers in the queue.

2.1.1 Example

Here follows an example:

The figure 1 is a description of the internal representation of the router system by the

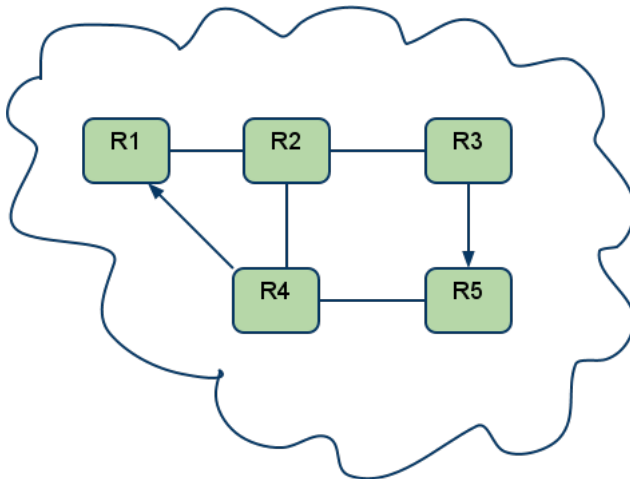


Figure 1: The router system of the network

network process. Notice that it has a set of routers (R1, R2, R3, R4, R5) that are connected to each other. Some connections are bidirectional, but as you can see, $R3 \rightarrow R5$ is one way

only, which means that any transfers from $R5 \rightarrow R3$ will have to take another route, for example $R5 \rightarrow R4 \rightarrow R2 \rightarrow R3$. What route to take will be defined by the routing tables. In that picture there is no `networkclients` are connected to the network.

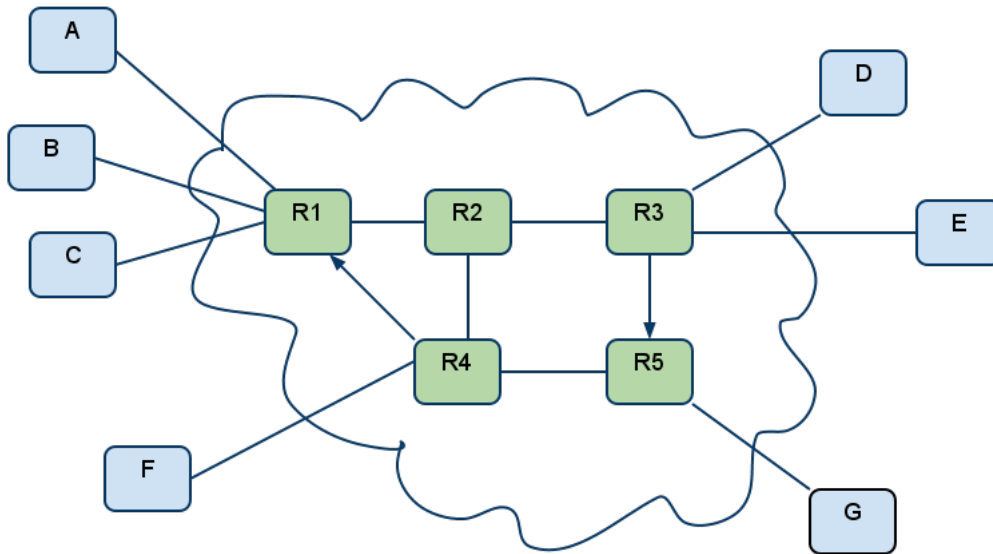


Figure 2: `Networkclients` connected to the network

When clients connects to the network, the network adds them to the router system, see figure 2. All clients are a unique Erlang process which sends a message that it wants to connect to the network. It also have to specify what router to connect to, the upload and download speed it has. The network process will then generate a IP-dress depending on the available range of IP-dresses the router has and give it to the `networkclient`. The network will also update the routing table of the connected router so that packages can be redirected to and from the new host. The default routing table are specified in the `configuration file` of the network along the capacity of the connections between the routers.

The `configuration file` is specified appendix B.

When the connected clients start sending transfers to each another, the network will have to have a system on how to delay them. The transfer must have a destination IP, a size, a header and a body. All transfers registered this way will be placed in a data structure with all other transfers. To progress the transfers, the network has a `heartbeat` process to it's assistance, see figure 3. This process sends messages regularly to the network process reminding it to progress the transfers.

When the network gets these reminders to progress transfers it will do so by trying to

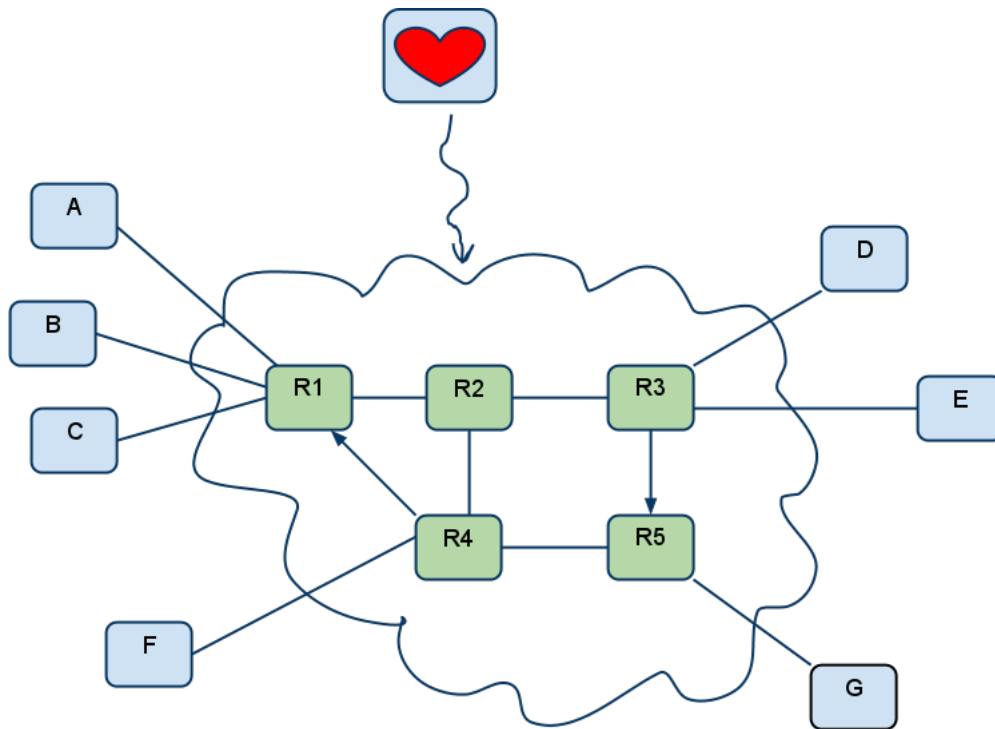


Figure 3: The Network's heartbeat process

A → D	Path: A,R1,R2,D	Hello, 25/100	Piece 4, 0/5000
B → D	Path: B,R1,R2,D	Hello, 95/100	Piece 4, 0/5000
C → D	Path: C,R3,R1,R2,D	Piecelist, 0/100	

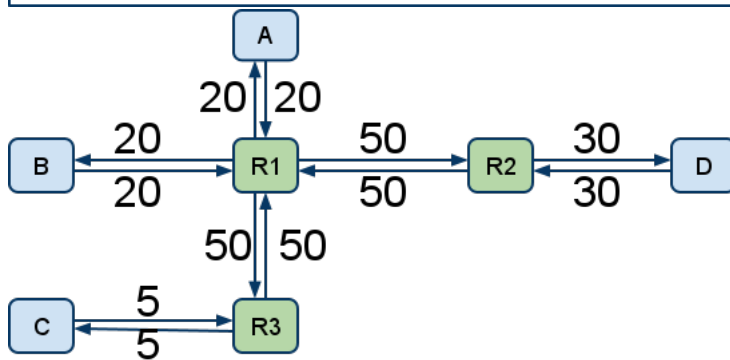


Figure 4: Networks transfers step 1 of 7

exhaust all the capacity the router system has in order to complete the transfers, and after it has traversed the transfers. When the `transferpicker` is done, all reduced capacities of the router system will be reset to their former values. To generate semi-fairness, the network picks transfers at random and let that transfer flow over the network, reducing the capacity of all connection it passes through. The amount reduces is the minimum of the remaining amount to be sent and the capacity of the bottleneck of the connections in the path. It will do so until all transfers are picked and progressed. Traversing these transfers is a recursive progress which has two variables, the transfer list and a copy of the router system. Figure 4 illustrate this. The transfers have paths which the packages will follow to find it's way to the destination host. They also have a "Sent"-field, a "Total to send"-field and a header for that transfer. Note that if there are many transfers using the same sender and receiver, the transfers are queued and will wait in a FIFO-queue to be sent.

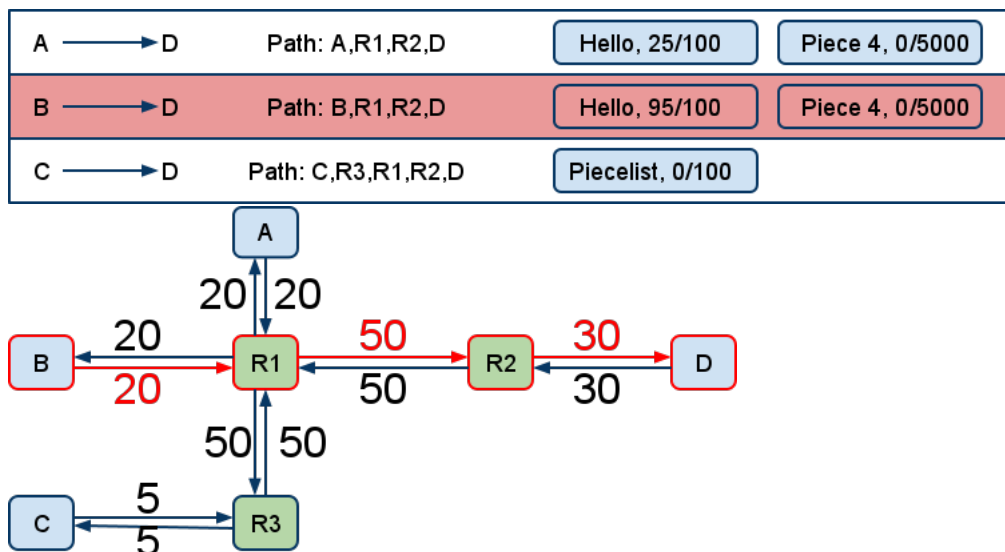


Figure 5: Networks transfers step 2 of 7

In the initial call, one transfer is picked. In this example (see figure 5) the second transfer is picked. As you can see, the bottleneck of this transfer is 20 ($B \rightarrow R1$ has capacity for 20 bytes and it is the bottleneck). The result will then progress 20 bytes for the first transfer in the queue. Since the first transfer in the queue has 95 of 100, it only need 5 bytes to complete and then it will give the remaining bytes the the next transfer in queue, leaving it at 15 of 5000. Also, since the transfer now is completed, the sending host will be informed of this and the receiver will get to see the contents of the message. Then the new state of the `transferpicker` can be seen in figure 6. Note that all connections on the path from B to D have decreased with the amount being transferred and the second

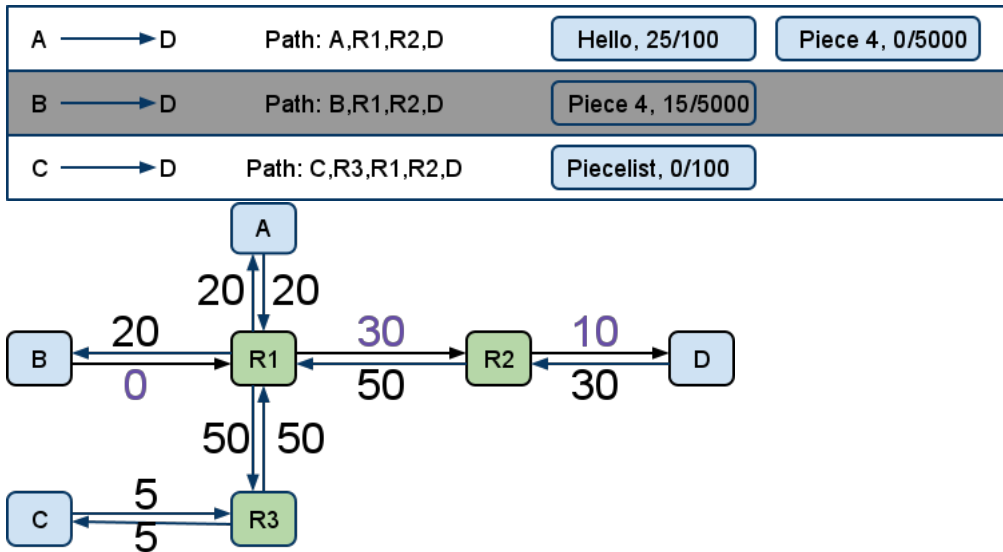


Figure 6: Networks transfers step 3 of 7

transfer is tagged as picked (by being marked as gray in the figure).

The Next recursive call a new transfer is picked at random, this time the first one is picked. It's detected that the bottleneck is now 10 (see figure 7) bytes and the transfer is progressed by 10 ending up in the next state (figure 8)

And the last recursive call we only have one transfer left to pick so it is picked and it is detected that the bottleneck is 0 and we can transfer nothing (figure 9) for it since the capacity of the connection currently is zero.

So we end up in the base case (No transfer left to progress), illustrated by figure 10 and we're done with the `transferpicker`. Note again that all reduction of capacities in the router system will be reset to what it was before the `transferpicker` and the progress of the transfers will remain the algorithm left them.

A → D	Path: A,R1,R2,D	Hello, 25/100	Piece 4, 0/5000
B → D	Path: B,R1,R2,D	Piece 4, 15/5000	
C → D	Path: C,R3,R1,R2,D	Piecelist, 0/100	

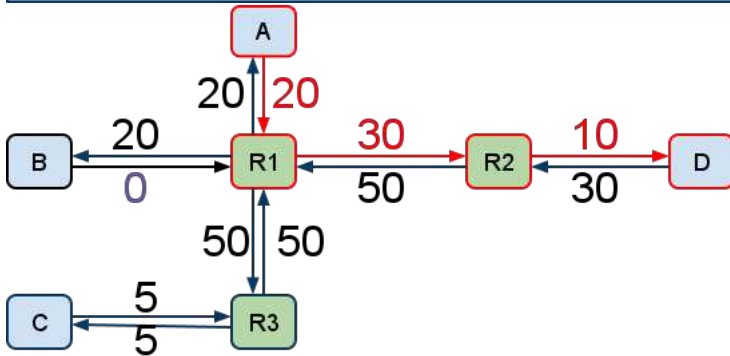


Figure 7: Networks transfers step 4 of 7

A → D	Path: A,R1,R2,D	Hello, 35/100	Piece 4, 0/5000
B → D	Path: B,R1,R2,D	Piece 4, 15/5000	
C → D	Path: C,R3,R1,R2,D	Piecelist, 0/100	

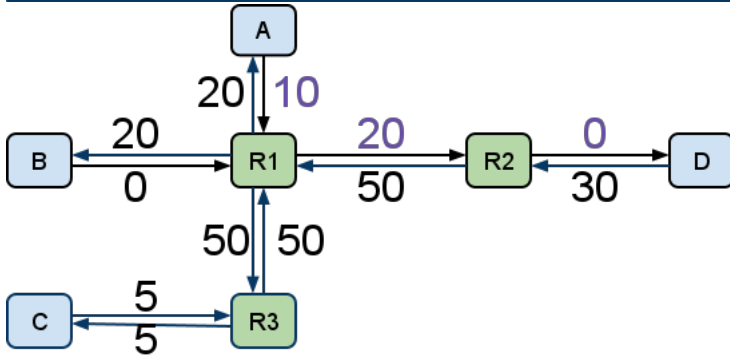


Figure 8: Networks transfers step 5 of 7

A → D	Path: A,R1,R2,D	Hello, 35/100	Piece 4, 0/5000
B → D	Path: B,R1,R2,D	Piece 4, 15/5000	
C → D	Path: C,R3,R1,R2,D	Piecelist, 0/100	

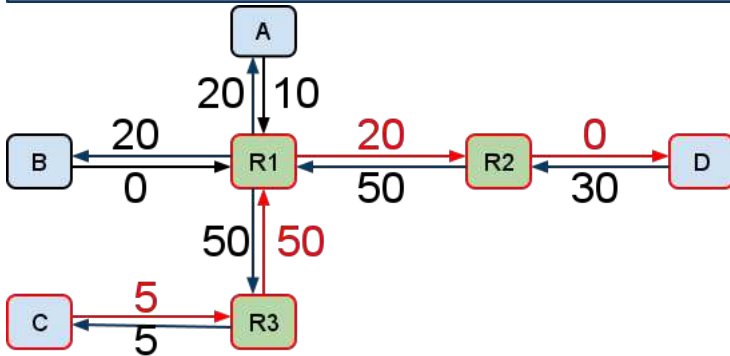


Figure 9: Networks transfers step 6 of 7

A → D	Path: A,R1,R2,D	Hello, 35/100	Piece 4, 0/5000
B → D	Path: B,R1,R2,D	Piece 4, 15/5000	
C → D	Path: C,R3,R1,R2,D	Piecelist, 0/100	

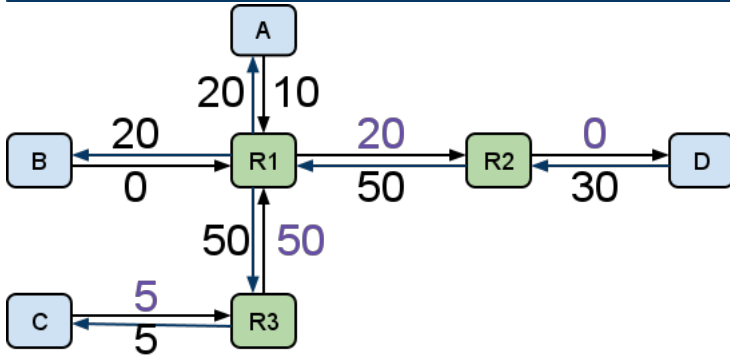


Figure 10: Networks transfers step 7 of 7

2.2 Network tick

`networktick.hrl` is where the main loop of the network is defined. It is the only function inside the `network` module which is not defined in `network.erl`. The reason for this is that that function is the main loop of the network and is very large. Also, it requires access to all functions in the `network` module. This function is tail-recursive and in each iteration handles one incoming Erlang-message.

2.3 Speed handling

Since the clients' does not know anything about the progress of their transfers while they're being transferred, the network has taken on an extra responsibility to tell the clients about the progress. Every second (Can be specified how often in the config), the network informs the clients of this. It sends out information about how many bytes the clients' have sent and received from all other clients that they have an active transfer with. This way the clients can calculate the speed they have between each other. In order not to send out too much data, only the amounts that have been updated since last time network informed is sent out this way.

2.4 Networkstate

The network state is the one argument that is passed around every function in the `network` module. It's just a simple tuple defined in `networkstate.erl`. The position of the elements in the tuple is defined as the following.

- `NET_LOG`
This is the file handle for the data log.
- `NET_ITERATION`
This is a integer which specify what iteration the network tick is on.
- `NET_DNSMAP`
This is the list with all dns that network hosts has registered.
- `NET_TRANSFERS`
This is the transfer object.
- `NET_NEXT_ID_TRANSFER`
This is the next available ID to assign to any new transfer
- `NET_ROUTERSYSTEM`
This is the router system object.
- `NET_HOST_TO_IP_MAP`
This is a data structure containing the network hosts Erlang pids mapped to IP-addresses.

- **NET_UNCONFIRMED_TRANSFERS**
This is a data structure containing all transfers that has not yet been accepted by the receiver
- **NET_SPEED_UPDATE_INFO**
This is a data structure containing all transfer updates that should be sent out the next time it's time to send out updates to the clients
- **NET_SKIPS**
This is an integer counting the amount of skipped `heartbeats`.
- **NET_TIMERS**
This is a list containing the handles of all timers.
- **NET_START_TIME**
This is the `now()` - timestamp from when the state was created.
- **NET_HEARTBEATS**
This is the total amount of `heartbeats` received.
- **NET_DEBUGLOG**
This is the file handle for the debug log.
- **NET_HEARTBEATINTERVAL**
This is the `Heartbeatinterval` currently used.
- **NET_SPEEDUPDATE_PROGRESS**
This is a tuple `{Left, Max}` for remembering when to send out `speedupdates`. `Max` is the maximum amount of `heartbeats` until it's time and `Left` is the remaining amount.

2.4.1 Router system

The router system is the data structure/module that handles the network connection graph. That is to keep track of all routers and `networkclients` in on the network and their connections. The router system also handles the following stuff Routing of transfers, Adding/Removing clients from the router system, Handling DHCP, Creating routers from a config-file. It also handles some descriptive functions to debug the network. It handles decreasing the capacity of a router path and finding the bottleneck of such a path.

Internally is the structure a `gb_tree` containing all routers and network hosts. It's a mapping from `IP` \rightarrow `Host` where `Host` is a tuple containing the following data.

Host information

- **HOST_NAME**
it's either `{router, SName}` if the host is a router created from section `SName` in the network configuration file or `{pid, Pid}` if the host is a network host with the Erlang pid `Pid`.
- **HOST_IP**
it's a tuple containing the ip address of this host. It's represented as a tuple `{A,B,C,D}` for example `{192,168,1,3}`
- **HOST_DHCP_START**
The first Dynamically IP to give out from this host. It's undefined if the host is a pid. It's stored the same way `HOST_IP` is stored.
- **HOST_DHCP_END**
The last Dynamically IP to give out from this host. It's undefined if the host is a pid. It's also stored the same way `HOST_IP` is stored.
- **HOST_CONNECTIONS**
All connections this host has. If it's a pid it should only be a connection to the router it's connected to. If it's a router, it should contain all connections it has to other routers and all connected clients it has. It's actually a list with tuples described in the list below.
- **HOST_RESERVED_IPS**
A list of all reserved IP addresses on this router, it's a empty list if this host is a network host. When hosts disconnect from the network and then wants to reconnect, it will get the same IP since it's reserved in this list.

Connection information The `HOST_CONNECTIONS` mentioned above contain the following data.

- **CONN_NAME**
A name for debugging the router system. A connection to a router is the name of that routers section in the configuration file or if it's a network client it's the IP of that client.
- **CONN_ERROR_RATE**
This is the odds of a package failing to be transferred. As of this moment Dec 14, 09, it's not implemented.
- **CONN_SEND_CAP**
This is the amount of bytes that can be transferred over this connection every heartbeat.

- **CONN_TARGET_MASK**
This is the Mask of the connection in this routing table. for example $\{255, 255, 0, 0\}$
- **CONN_TARGET_IP**
This is the destination IP of this connection routing table. for example $\{192, 168, 0, 0\}$

2.4.2 Transfers

Whenever A sends a transfer to B, B is informed about the incoming transfer, it's header and that it's A that is the sender. Then the transfer is tagged as unconfirmed inside of the network. B can then choose to deny it, which will inform A that the transfer failed and the transfer is removed. B can also choose to accept it in order for the message to be placed in the transfer buffer between A and B. When B accepts the transfer, the transfer will be tagged as confirmed. All confirmed transfer from A to B are stored in a buffer which is implemented as a FIFO queue. As the transfers are completed, they will be removed from this buffer and the receiver will be informed about the transfers' contents.

Note If B does not reply in chronological order of the transfer, there is no longer any guarantee that the transfers will arrive in order they were sent. Note that the transfers from B to A have another buffer.

Network transfers The transfers in the networkstate is stored as a tuple of three elements.

- $\{\text{PathMap}, \text{ConnectionIndices}, \text{NextInsert}\}$ Where **PathMap** is `gb_tree` of $\{\text{FromIP}, \text{ToIP}\} \rightarrow \text{ConnectionIndex}$.
- **ConnectionIndices** is an array containing a tuple $\{\{\text{FromIP}, \text{ToIP}\}, \text{Transfers}\}$. where each element is a connection from someone to someone else, and **Transfers** is the list of all transfers going in in that direction. How the elements in that list are defined are described below 2.4.2.
- **NextInsert** is the next position in the array to insert a new element at.

Transfer A Transfer is defined as a tuple with the following elements.

- **TRANSFER_HANDLE** The ID of the transfers.
- **TRANSFER_FROMIP** The IP of the sender.
- **TRANSFER_TOIP** The IP of the receiver.
- **TRANSFER_PATH** A list of the IPS that the transfer passes through with the senders IP at first and the receivers IP at last.

- `TRANSFER_FROMPID` The pid of the receiver.
- `TRANSFER_TOPID` The pid of the sender.
- `TRANSFER_HEADER` The header of the message. This is what the receiver will see at first when he are to decide whether or not to accept the transfer.
- `TRANSFER_MESSAGE` This is the actual contents of the transfer that the receiver will see first when the transfer is completed.
- `TRANSFER_SENT` The amount of sent bytes.
- `TRANSFER_TOTALTOSEND` The total amount of bytes of this transfer.
- `TRANSFER_SPEEDHANDLER` The speedhandler of the transfer. This is not used any more.
- `TRANSFER_STARTTIME` The time when the transfer started.

3 Bittorrent Client

Bittorrent clients, or peers, have the role in the simulator to download the fake file using different settings and algorithms. Furthermore they should be able to simulate what a user could do to affect the download as well. For example disconnect, reconnecting, connecting when ever the user wants and a lot more. To make this possible there is a `configuration file` for the clients where different settings can be specified and the user can tell the simulator what custom functions to load and use. The `configuration file` is specified appendix B.

When a peer is spawned and ready to go it has the task is to send messages through the network using the API `networkiface.erl`. Exactly how this is done and what other miscellaneous tasks they do as well will be gone though in the 3.4 section.

3.1 Master Client

On boot up when the `networkmaster` process spawns it will spawn another process named Master Client that uses the `masterclient.erl` module. The Master Clients task is to read in the `configuration file` and setup different States for every section in the configuration file. More about States in the 3.2 section below. In the different sections it is specified how many clients that should be spawned and when they should be spawned. Spawning the right amount of clients at the correct time is also the Master Clients task. When every client that was specified in the `configuration file` is spawned the Master Client process will terminate.

To load in the different settings and algorithms the Master Client uses the module `read_config`. With help from the module the different section names are loaded in. Now a

default State will be created for the clients from the general section settings. If nothing else is specified in the sections below these are the settings that will be used to spawn the clients. After setting up the default all the other sections will be gone through one after another, looking up how many clients to spawn and what settings/algorithms to overshadow in the general section. When all the States are setup the Master Client will send Erlang messages to itself with a delay. The delay is the time that was given in the `configuration file` to wait until a client should join the network. After sending all the messages it goes into a receive loop, waiting for the message containing the State and how many clients to spawn with this State. As a message is received it spawns the given amount of clients and gives them their State.

3.2 State

Every client have their own personal State that they work with. A State contains all the information a client knows about its surrounding, such as what peers it has in its neighborhood, what router its connected to, etc. The State consists of a tuple containing two tuples.

```
State = {Masterstate = tuple(), Clientstate = tuple()}
```

`Masterstate` being the state that the `Masterclient` sets up for the clients and `Clientstate` is the state that the clients set up when they are spawned. Another way to view it is that the `Masterstate` contains the information that was loaded from the `configuration file`. Slightly edited though by the Master Client who loads in custom functions for the clients. Working against the State is done by macros, this is not to be confused with a user working against the State. A user should use the `clientiface.erl` module, read about it in section 3.3. There is a setter and a getter macro for each value in the State on the form `SET_MACRONAME{State, Value}` and `MACRONAME{State}`. `MACRONAME` is the name for what it gets, for example `IP{State}` will return the clients IP. The macros are saved in `clientstate.hrl`.

The macros that work against the `Masterstate` and what the macros return are the following.

- `SECTION`
Section name - Returns the section name that was given in the `configuration file`. The web interface uses this information to group the clients. This is needed since every client in one section runs on the same settings, so calculating an average for the clients belonging to that section is of interest.
- `MUC`
Max up connections - defines how many peers a peer can upload to simultaneously.

This is not used since the custom function for choking limits the uploads. However the choking function could use it as the number of how many to upload to.

- **MDC**
Max down connections - defines how many peers a peer can download from simultaneously. This should not be used since in Bittorrent a peer should be able to download from any peer possible. Until it is removed use a high number here, so it does not affect the simulation.
- **BWUP**
Bandwidth up - This is the upload speed a client has to the router it is connected to.
- **BWDN**
Bandwidth down - The download speed a client has to the router it is connected to.
- **ERRUP**
Error up - Not used but was supposed to be the package loss percentage when uploading to the router the peer's connected to.
- **ERRDN**
Error down - Not used but was supposed to be the package loss percentage when downloading from the router the peer's connected to.
- **ROUTER**
Router - The name of the router the client should connect and be connected to.
- **PIECES**
Pieces - When the master client sets it up this will be a string of what pieces a peer has at first. Later on when the `extrasettings` module loads in the `Clientstate` this will be changed to a data structure that `bitfield` works on. Read more about pieces in section 3.6.
- **SPIECESIZE**
Sub piece size - The size the sub pieces are except for the last sub piece in a piece. It might be a bit smaller depending on whether or not piece divided with sub piece is an even number.
- **PS**
Piece Selection - Gives a reference to the piece selection custom function. Read more about the custom functions in section 3.8.
- **CH**
Choking - Gives a reference to the `choking custom function`. Read more about the custom functions in section 3.8.

- **AS**
Anti-snubbing - Gives a reference to the `anti-snubbing` custom function. Read more about the custom functions in section 3.8.
- **AB**
Availability - Gives a reference to the `availability` custom function. Read more about the custom functions in section 3.8.
- **NC**
Neighborhood check - Gives a reference to the `neighborhood check` custom function. Read more about the custom functions in section 3.8.
- **OU**
Optimistic unchoking - Gives a reference to the `optimistic unchoking` custom function. Read more about the custom functions in section 3.8.
- **MAXPEERS**
Max peers - This is the maximum amount of peers a client should connect to. A peers neighborhood should not contain more neighbors than what this number specifies.
- **OUTIME**
Optimistic unchoking time - Defines how often the custom function for optimistic unchoking should be called. The duration a peer should be optimistically unchoked is also decided by this value. Extra optimistic unchokes may occur if anti-snubbing is used.
- **CHTIME**
Choking time - Defines how often the custom function for choking should be called.
- **ASTIME**
Anti-snubbing time - Defines how often the custom function for anti-snubbing should be called.
- **PSTIME**
Piece selection time - Defines how often the custom function for piece selection should be called.
- **NCTIME**
Neighborhood check time - Defines how often the custom function for neighborhood check should be called.
- **ABTIME**
Availability time - Defines how often the custom function for availability should be called.

- **SD**
Snubbed duration - The time a peer should be choked if anti-snubbing notifies that the peer is snubbing.
- **TUJ**
Time until join - This is the time in milliseconds that the Master Client waits before spawning the client.
- **CSTATE**
Custom State - Before the clients initializes the custom state it's a reference to the init custom function. When it's initialized it depends on what the init custom function return what is stored here. Read more about it in section 3.4.

The macros that work against the Clientstate and what the macros return are the following.

- **SPAWNTIME**
Spawn time - This is a `now()` timestamp when the client was spawned.
- **DIE**
Die is a boolean, if it's true then the client is about to shut down permanently or if it's false the client is not about to shut down.
- **CON**
connected - A boolean that specifies if the client is connected to the network or not.
- **FSIZE**
File size - The size of the file that the client is downloading. Read from the torrent `configuration file`.
- **PSIZE**
Piece size - The size for a piece. Read from the torrent `configuration file`.
- **SPIECES**
Sub pieces - Read section 3.6.
- **HPIECES**
Have pieces - Read section 3.6.
- **DHPIECES**
Don't have pieces - Read section 3.6.
- **SEEDER**
A boolean that is set true if a peer is a seeder, false if the peer is a leecher.

- **SPIECEUP**
Sub piece uploaded - The total size uploaded of the file. Only packages that are completed are used in the calculation.
- **SPIECEDN**
Sub piece downloaded - The total size downloaded of the file. Only packages that are completed are used in the calculation.
- **LCH**
Last choking - A `now()` timestamp when the custom function for choking was last called.
- **LAS**
Last anti-snubbing - A `now()` timestamp when the custom function for anti-snubbing was last called.
- **LAB**
Last availability - A `now()` timestamp when the custom function for availability was last called.
- **LOUASCH**
Last optimistic unchoking anti-snubbing choking - A `now()` timestamp that specifies the minimum wait time before the clients checks if optimistic unchoking, anti-snubbing or choking should be performed again. This option overrides the times given in the `configuration file` and should not be used. It was implemented to be used for debugging only.
- **LPS**
Last piece selection - A `now()` timestamp when the custom function for piece selection was last called.
- **LNC**
Last neighborhood check - A `now()` timestamp when the custom function for neighborhood check was last called.
- **LSENTALIVE**
Last sent alive - A `now()` timestamp when the client last broadcasted alive to its neighbors.
- **LALIVECALL**
Last alive call - A `now()` timestamp when the client last checked if any peers in its neighborhood have pinged out.
- **DNS**
Domain Name System - The name of the tracker the client should connect to.

- **TRACKER**
Contains the IP of the tracker.
- **MAT**
Minimum announce time - The minimum time, specified in milliseconds, the client has to wait before announcing to the tracker again.
- **MRI**
Minimum request interval - The minimum time, specified in milliseconds, the client has to wait before sending another request message to the tracker.
- **LAT**
Last announce time - A `now()` timestamp when the last announce was sent to the tracker.
- **LRI**
Last request interval - A `now()` timestamp when the last request was sent to the tracker.
- **PENDINGHANDSHAKES**
The number of handshakes the client is sending and receiving at the moment. This is used so the client does not exceed the `MAXPEERS` value by contacting new peers when there are pending handshakes.
- **PEERS**
Contains a peer's neighbors and information about them. Read section 3.5 for detailed information.
- **EXOUS**
Extra optimistic unchokes - Contains the IP of the peers that are additionally optimistic unchoked because of occurrences of snubbing.
- **DEFOU**
Default optimistic unchoke - Contains the IP of the default optimistic unchoke. The default optimistic unchoke is changed at regular intervals.
- **UNCHPEERS**
Unchoked peers - A `gb_tree` where the key is the IP for a peer the client is currently unchoking.
- **CHPEERS**
Choked peers - A `gb_tree()` where the key is the IP for a peer the client is currently choking.

- **UNCHINGPEERS**
Unchoking peers - A `gb_tree` where the key is the IP for a peer the client is currently being unchoked by.
- **CHINGPEERS**
Choking peer - A `gb_tree` where the key is the IP for a peer the client is currently being choked by.
- **SEEDERS**
A `gb_tree` where the key is the IP for a peer that is a seeder which the client currently has in its neighborhood.
- **CHOKEDIFFS**
Read section 3.4.
- **SNUBBED**
A `gb_tree` where the key is the IP for a peer the client is currently refusing to upload to because of snubbing.
- **MSGOUT**
Outgoing messages - A state for messages that the client is sending to another `networkclient`. The state is a `gb_tree` which `messagehandler` work against. The key is a handle for a message and the stored value contains information about the message.
- **MSGIN**
Incoming messages - A state for messages that the client is receiving from another `networkclient`. The state is a `gb_tree` which `messagehandler` work against. The key value is a handle for a message and the stored value contains information about the message.
- **AMSGOUT**
Aborted outgoing messages - Total amount of outgoing messages a client has aborted during its life span.
- **AMSGIN**
Aborted incoming messages - Total amount of incoming messages a client has aborted during its life span.
- **FMSGOUT**
Failed outgoing messages - Total amount of messages that failed when client tried to send.

- **FMSGIN**
Failed incoming messages - Total amount of messages that failed when client tried to receive.
- **SMSGOUT**
Successfully sent messages - Total amount of messages that succeeded when client tried to send.
- **SMSGIN**
Successfully received messages - Total amount of messages that succeeded when client tried to receive.
- **LOG**
The handle for the client's log.
- **DEBUGLOG**
Debug log - The handle for the client's debug log.
- **DCMD**
Debug command - Used when a `user_defaults.erl` command that changes the client's state is given in the shell. When this is the case the clients need to make the change in the state and what type of change is saved here.
- **TICK**
Contains the amount of ticks the client has done. Which is the same as how many times a client has recursively called itself.
- **MYIP**
My IP - Contains the client's IP.

3.3 Framework

A client lives in a cycle which is called the client's life cycle. In this life cycle clients work with their State, updating it depending on different messages it receives, what custom functions returns or if ping outs occurs. Before entering the life cycle some initialization is needed. Upon spawning the first task for a client is to find the network and create a link between the network and itself. This is used when the network for some reason exists the clients should exit as well, the clients depend on the network and can not exist without it. After the link is created a sleep-call is executed for a random time between 0 and 5000 milliseconds making sure all clients in a section do not connect to the network at the exact same time. Setting up the client state and reading in the information from the torrent (config) file is the next step. The amount of pieces, sub piece and sizes for the different pieces is calculated here and the initialization function for the custom state is called. The

reference for the initialization function gets replaced by the return value it gives. Now the client is ready to connect to the network.

Connecting to the network is done by the `connectionhandler` module which uses the general API `networkiface.erl` to talk to the network. An Erlang message is sent to the network that the client wants to connect and to what router it wishes to connect to, it then waits for getting a connect success. After getting the connection successful message from the network the clients continues by setting up the log, debug log and the IP it has. A client that is connected to the network is useless until its connected to the tracker so after connecting to the network it sends a message to the network requiring the IP of the given DNS. Once the client has the IP it tries to send a request event started and waits for the message to finish. It does not wait for a reply from the tracker and enters the main loop which is called the life cycle.

3.4 Life cycle

In the life cycle a repeated behavior is performed, except if a disconnect is returned from the availability custom function. All custom functions have wrappers around them to handle when to call them and the return value from them. All custom functions returns a new custom state that all the wrappers need to update the client state with. In the client config file different intervals are given for example how long the client should consider itself snubbed by another client. The wrappers checks these intervals as well and make necessary changes if the time interval has passed. Read in section 3.8 about what the custom functions do and return in detail. The following description will be a walk through in the life cycle step by step describing what happens in each step.

In figure 11 a simplified picture of the life cycle can be seen. Going through the cycle once is called performing a tick and the amount of ticks that have been done is saved in the state.

Updating logs is handled by the module `loghandler`. To update the normal log there are two possible functions to use. The first one is if the network has sent a message ordering the client to update it, this is used in the normal case. Forcing the log to update is another way and is used when a client disconnect from the network. Normal logging has default values it logs from the state and logs all of them each time logging is done.

Debug logging is unlike the normal logging one on the fly when the change happens. It does not log default values but the function calling the debug log needs to specify the string and values it wants to log. Debug logging should only be turned on when actually debugging is needed otherwise it should be turned off since it's a very heavy operation for the disks when large amounts of clients do this simultaneously.

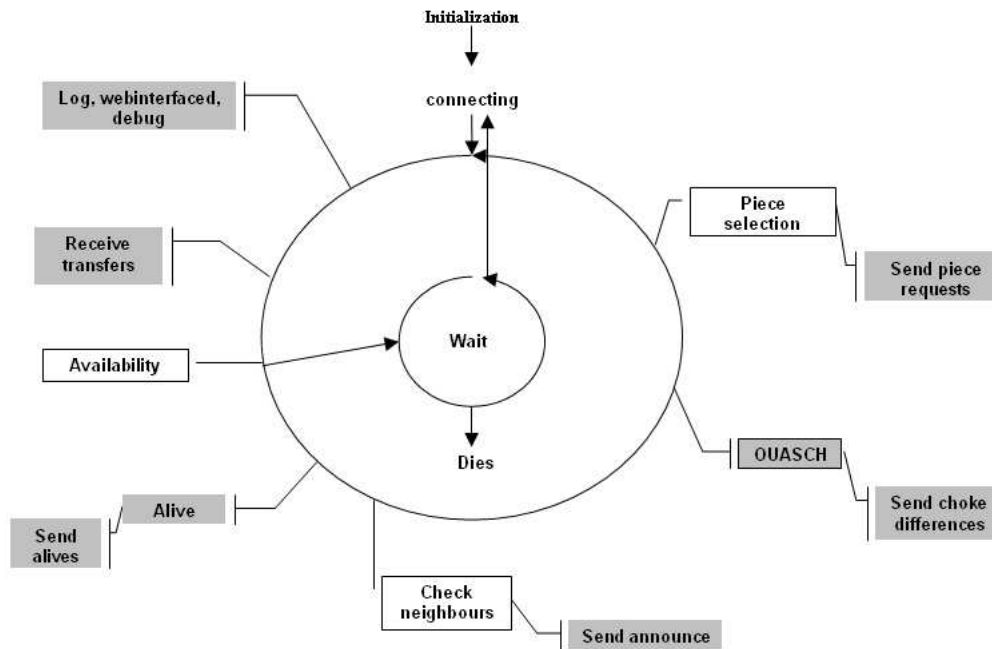


Figure 11: Life cycle for a single client.

Communicating with the web interface is handled by the module `webinterface`. When the web interface needs information it will send a Erlang message to the clients which the clients catches here. To make it more efficient the web interface might ask for different types of messages by sending an `atom()` with the message. The client will then reply with a record containing information that depends on the `atom sent`.

Debug messages can be written in the Erlang shell. More about what commands there are and how they work can be found in appendix B. To check if a debug message has arrived and printing the information the debug command requires the `clientdebug` module is called. Currently there are two debug commands for the clients that actually orders them to perform an operation. These two commands are permanently disconnect and disconnect for a given amount of time. If one of these commands are returned the clients tick function will catch it and run another function to run the given command.

The wrapper for availability is called by the `connectionhandler` module. This module checks that the client does not get a connect when already connected or disconnect when already disconnected from the availability wrapper. Further more it makes the changes needed to the state when disconnecting and the changes needed when reconnecting. It provides a waiting loop when the clients are disconnected waiting for reconnect or

permanently disconnect. Note that all clients need to disconnect gracefully from the tracker and the network, if this rule is not held the simulator will not work. The actual wrapper for the `availability custom function` lies in the `wrappers` module where all custom function wrappers are, except for the `initialization custom function` which has no wrapper. It calls the function if the interval that was given in the `client configuration file` has passed and makes sure the return value is a valid value. Valid return values are:

- `{idle, CState}`
If the client should do nothing about its current connection state. `CState` is the new custom state.
- `{disconnect, CState}`
Disconnect defines that the client should disconnect and enter the wait loop, awaiting further instructions. `CState` is the new custom state.
- `{connect, CState}`
If the client is in the wait loop then it should reconnect when `connect` is returned. `CState` is the new custom state.
- `{die, CState}`
When `die` is returned the client should disconnect and the client process should exit. `CState` is the new custom state.

Checking incoming transfers uses the `pkgcom` module, more information can be found in section 3.7. Basically what the module does is checking if a transfer has been received and acts depending on what type of transfer was received.

Checking alive intervals and sending out alive message to the neighbors is handles by the `alivehandler` module. The alive handler has three major tasks, removing peers from the neighborhood that have pinged out, sending alive messages to all its neighbors so they don't remove the client from their neighborhood and removing peers from all places in the state as they are removed.

The wrapper for neighborhood check is called next. If the interval to call the custom function that was given in the `configuration file` has passed and enough time passed since the last announce or request was sent to the tracker the custom function will be called. The return value should be one of the following:

- `{announce, CState}`
When `announce` is returned the client should announce to the tracker that it wants more peers to its neighborhood. `CState` is the new custom state.

- `{announce, Numwant}, CState`
If an announce is returned with an integer the client should announce to the tracker that it wants the Numwant amount of peers more to its neighborhood. `CState` is the new custom state.
- `{dont_announce, CState}`
If the client should not take any action the custom function should return dont announce. `CState` is the new custom state.

OUASCH (Optimistic Unchkoing, Anti-snubbing, Choking) is a wrapper that calls the optimistic unchoking wrapper, then the anti-snubbing wrapper and finally the choking wrapper. **OUASH** is needed because there is a chance that for example optimistic unchoking and choking to be called in the same tick. If optimistic unchoking chokes peer 1 and choking unchokes peer 1 afterwards we do not need to inform that peer that it got choked and then unchoked right away. What the wrappers for optimistic unchoking, anti-snubbing and choking do is calling the module `chokinghandler` defining what peers they want to choke or unchoke. Choking handler has a state called `CHOKEDIFFS` where it saves if a peer was originally choked or unchoked when a change is made to them. It then changes the value in the peer state to the new value, read more in section 3.5. When the three wrappers for the the custom functions have been called the choking handler is called once more. For each peer that exists in the `CHOKEDIFFS` state it checks what the original choking status was compared to the new choking status. If they are not the same the client sends the new choking status to the peer and stops uploading the file to him. The stop is immediate which means that all uploading message of the file need to be aborted.

Calling the piece selection wrapper can be done by two functions. The first function is a check if the interval that was specified in the `configuration file` has passed. If this is the case then it will call the real piece selection wrapper with the state and the atom interval as argument. Calling the real piece selection wrapper directly with the second argument being a tuple containing IP, piece number, sub piece number and the atom success or fail is the other way. This is done when a client successfully or not successfully downloaded a sub piece from a peer with the given IP. When the real piece selection wrapper is called it will call the piece selection custom function and expect a return value that contains a list of tuples that specifies a piece, piece number and an IP to download it from. The custom function should as usual return the new custom state value as well. Below is how the structure of the return value should be:

```
{[{Piece number, Sub piece number, IP} | Tail], CState}
```

Using the return value from the custom function the piece selection wrapper calls a help function to request the following sub pieces from its neighbors. There is support in this help function for the max down connections value that can be specified in the

configuration file for the clients. However the max down connections setting should not be used since it's not a part of bit torrent and should be removed, the help functions only task is to recursively go through the sub piece to request and request them with help from the `pkgcom` module. Read more about the module in section 3.7.

3.5 Peers

The peer state is a `gb_tree` with the IP for a peer as key and a record of all the information that the client knows about the peer in the form of a record. Module `cpeerhandler` creates the tree, record and performs the edits on it. Furthermore the module also keeps `CHPEERS`, `UNCHPEERS`, `CHINGPEERS`, `UNCHINGPEERS`, `SEEDERS` and `SNUBBED` states updated. Working on the module is tried to be kept as simple as possible by having one setter function and having multiple getters that transforms the data to a useful form for the caller. For example the `get_uploading_pieces(State, IP)` getter takes the part of the record that contains uploading pieces to the given IP and creates a sorted list of them. For each peer the transfers that are uploading and downloading from it are saved which makes the state a little more complex. A couple of new setters are needed to handle it which are named `add_tr_out()`, `add_tr_in()`, `delete_tr_out()` and `delete_tr_in()`. They are used for adding and deleting transfers in and out to a given peer. One of the parameters for the add functions is an message type atom that describes what type of message it is. If the message type is `send` or `request` the module will know that its a message that requests, receives, got a request or sends a sub piece to another peer and will save this extra information in the record as well. The record state contains the following information for each peer:

- `ip`
The IP for the neighbor, this should be the same as the key value in the `gb_tree`.
- `alive`
A `now()` timestamp when the last alive was received from the neighbor.
- `ou`
Optimistic unchoke - If the neighbor is optimistically unchoked or not, if it is there should be a `now()` timestamp saying when it was unchoked.
- `outspeed`
How fast the client is currently uploading to the neighbor.
- `inspeed`
How fast the client is currently downloading from the neighbor.
- `interesting`
If the client is interested or not interested in the neighbor.

- **interested**
If the neighbor is interested or not interested in the client.
- **pieces**
Contains the pieces the neighbor has. The pieces state is on the same form as the clients pieces state, which means that the peer handler module uses the `bitfield` module to work on it.
- **ch_status**
Specifies if the client is choking or unchoking the neighbor.
- **peer_since**
A `now()` timestamp when the neighbor was added to the clients neighborhood.
- **trans_out**
Outgoing transfers - Contains a `gb_tree` that is handled by the `messagehandler` module where the key is a handle to a message currently uploading to the neighbor.
- **trans_in**
Incoming transfers - Contains a `gb_tree` that is handled by the `messagehandler` module where the key is a handle to a message currently downloading from the neighbor.
- **int_bits**
Interesting pieces - Contains all interesting bits the neighbor has where interesting bits are all pieces the neighbor has that the client does not have. Help functions to the peer handler to work on this state can be found in the `interestinghandler` module. The form of the `int_bits` state is the same as the form of the pieces state.
- **ching_status**
Choking status - An atom describing if the neighbor is choking or unchoking the client.
- **seeder**
A boolean that's true if the neighbor is a seeder.
- **snubbed**
An atom that is snubbed if the neighbor is snubbing the client, otherwise the value is `not_snubbed`.
- **last_piece**
If the neighbor unchoked and the client is trying to download from it this will be a `now()` timestamp. The timestamp stands for when the client last downloaded a sub piece from the neighbor. Used for anti-snubbing, if this timestamp is too old then the client can consider itself snubbed.

- **inrequest**
in requests - A `gb_tree` that `messagehandler` uses. The keys are handles to the requests that the neighbor is sending to the client.
- **outrequest**
out requests - A `gb_tree` that `messagehandler` uses. The keys are handles to the requests that the client is sending to the neighbor.
- **insend**
in sends - A `gb_tree` that `messagehandler` uses. The keys are handles to the sub pieces that are transferring from the neighbor to the client.
- **outsend**
out sends - A `gb_tree` that `messagehandler` uses. The keys are handles to the sub pieces that are transferring from the client to the neighbor.

3.6 Pieces

To make changes to the different piece states two modules are used, `bitfield` and `spiecehandler`. In the most common case the `spiecehandler` module uses the `bitfield` module to make changes to the `DHPIECE`, `HPIECES` and `PIECES`. However the `bitfield` module can make changes to these on itself, which is used sometimes for the `PIECES` state.

The states module `bitfield` work on are `gb_trees` where the key is the piece number and the value is a boolean which specified if the client has the given piece or not.

`Spiecehandler` has its own state, `SPIECES`, which keeps track of what sub piece the client already has and what sub piece it is downloading from what neighbors. The structure of the state is shown in figure 12. In the example a `gb_tree()` with three pieces is shown where the piece number is the key value. The value in the node consist of yet another `gb_tree`, this tree is however of the sub pieces that exist for that piece. The key value in this tree is the sub piece number and the value in the node can either be true or a list of IPS. If the value is true then it means that the client already has the sub piece and does not need to download it. In case it's a list then the client does not have the sub piece and might be downloading it. In case the client is trying to download the sub piece there will be a `term()` IP, or more, where IP is expected to be `{ip, status}`. Ip should be the ip to the neighbor the client is requiring the sub piece from and status is an `atom()` value. The value can be:

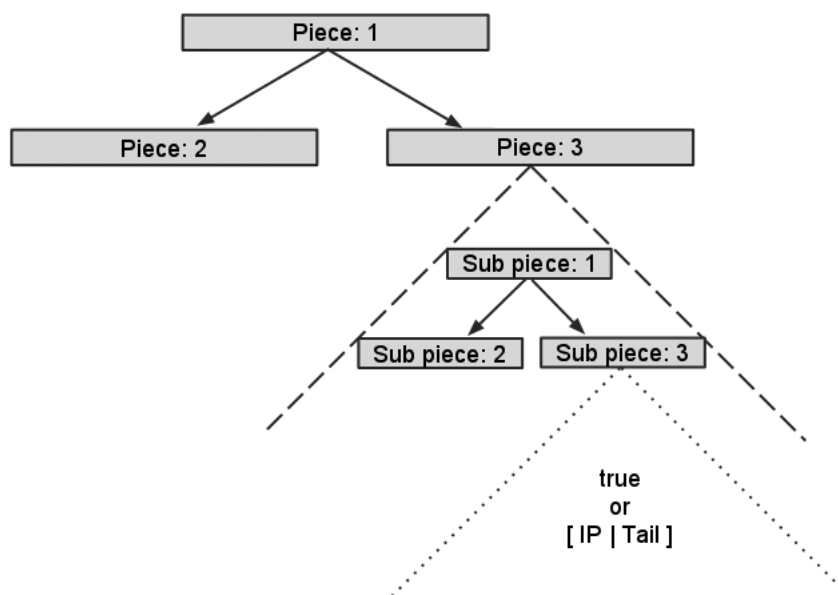


Figure 12: The structure of the state that spiecehandler erl work against.

- **requesting**
When the client is requesting the sub piece from the given neighbor.
- **waiting**
When the request to the neighbor has been completed and the client is awaiting an answer.
- **sending**
When the client is downloading the sub piece from the neighbor.

There are a lot of getters and setters in this module since it's a pretty complex state and might be problematic to get and edit information in it.

3.7 Package communication

Sending packages to each other via the network is done with the `networkiface` module, but its a very general module making it limited. The clients uses the package communication module `pkgcom` as a middle hand to help setting up the messages they want to send and

making changes to the state when receiving a message. Furthermore the module handles the aborting of any messages to a neighbor that involves uploading to it.

There are two send functions in the module, one to send to another client and the other when a client wants to send to the tracker. If a message to another client is to be sent the IP to the client must be specified, the state must be given and what type of message it is. The module will then setup the message depending on what the type was, send the message and change the data in the state that needs to be changed when the message is sent. In most cases this involves saving the message and the handle to it, adding the message to the peer state and if it was a piece message updating the SPIECE state. When sending to a tracker the same rule applies except for an IP is no needed since the client has the IP in the state of tracker it is connected to.

A client can receive a lot of different types of messages, the package communication module work on messages that are sent over the network as transfers. There are three types of messages that can be received:

- **Incoming**
If another client is trying to send the current client a message which needs to be approved before the transfer can take place.
- **pts/2**
`pts(State, Handle)` - If a message the client was sending is completed it will receive the message pts with a handle. The client then knows that the message with the given handle is now completed.
- **pts/3**
`pts(State, Handle, Message)` - When the client has accepted and transfer from another client and the transfer completes it will get a pts message with the handle for the message and the message. It can then use the handle to check what message got completed and look in Message to see what the message was.

Depending on what messages completes a lot of help functions are triggered to make the appropriate changes to the state. In some cases it triggers the client to send a message of its own, for example when the tracker has send new IPS of peers it can contact to get as neighbors the client sends handshakes to them right away.

3.8 Custom Behavior / Algorithms

Custom behavior, custom algorithms and custom functions have all the same meaning. They are a group name for algorithms needed to do perform piece selection, initialize, neighborhood check, optimistic unchoking, choking, anti-snubbing and availability.

4 Process communication

A lot of different messages are sent through out the simulation. Erlang messages are sent between the Erlang processes while the Bittorrent client transfers simulation messages to each others which mainly consists of a body and a header.

This section holds a definition of all Erlang messages that can be sent to the network and the network clients.

4.1 Messages sent to the network

Here is a table of what messages the network can receive. It's a definition of what the `atom` sent to the Erlang process is supposed to look like and also who the messages is supposed to come from.

Any `Pid` in the list below refers to the `Pid` of the sender process.

- `exit`. The network quits instantly. This message is sent by any debug interface.
- `{deny_incoming_package, Pid, Handle}` Reject the incoming package with the given handle. Sent by any connected network client.
- `{accept_incoming_package, Pid, Handle}` Accept the incoming package with the given handle. Sent by any connected network client.
- `{get_client_info, IP, WPid}` The network sends back detailed information of the specific client with using the address `IP` to `WPid`. Sent by the web interface.
- `{get_clients, WPid}` Sent by the web interface. This is a question to the network of what clients that are connected and their general information, and the reply should be sent back to `WPid`.
- `{get_clients_dns, WPid}` Sent by the web interface. This is the same question as the one above, except sent only those clients that have registered a DNS.
- `{get_state, Pid}` Sent by any debug interface. Ask the network to send back it's state to the sender process with pid `Pid`.
- `{get_times, Pid}` Sent by the web interface. Ask the network to send back to pid `Pid`, a tuple with the real time that has taken for the simulations and how many `heartbeats` that the network have run. Reply tuple should look like `{times, Timetaken, Heartbeats}`.
- `{is_ready, Pid}` Sent by any debug interface or control interface when the user is ready to start the simulation. The network will then reply with `{running, Skips, Heartbeats, TimeTaken, SimulationTimeTaken}` given that the network is running. Otherwise it will reply to `Pid` with the atom `ready`.

- {ping, Pid} Sent by any debug interface to see if the network is still there. The network will reply with pong to Pid.
- {connect, Pid, R, BU, BD, EU, ED} Sent by any network client as the initial message when they connects to the network. R is the name of the router to connect to, BU and BD are Bandwidth up and bandwidth down respectively. EU and ED are error rate up and error rate down respectively.
- {package, P, IP, S, M, H} Sent by any connected `networkclient` as the start of a new transfer. P is the pid of the sender, IP is the IP of the receiver, S is the size of the message, M is the contents of the message and H is the header of the message.
- {dns_lookup, Pid, DNS} Sent by any process that wants to lookup a DNS. The network will reply with the IP of the owner of the DNS if there is one.
- {dns_register, Pid, DNS} Sent by any connected network client that wants to register a DNS to it's IP address.
- {get_my_ip, Pid} Sent by any connected network client. The network will send back the IP to the client.
- {disconnect, Pid} Sent by any connected network client. This network client has just disconnected from the network and the network will remove it from it's state.
- {abort_package, Pid, Handle} Sent by a connected network host that is either the sender or receiver of the transfer with the given handle. Aborts the transfer with the given handle.
- {debug_print_network_client, DebugMsg, Host} Sent by any debug interface. The network will redirect the message to Host. Which is either the DNS or the IP.
- heartbeat Sent by any debug interface. Request the network to perform an extra heartbeat. Used mainly for debugging when the interval of heartbeats are long.
- {log, MyPid} Sent by the network itself when it's time to send out log reminders to all connected clients.
- {heartbeat, MyPid} Sent by the network itself when it's time to perform a heartbeat.

4.2 Messages sent to the networkclients

- `connect_success` Successfully connected to the network.
- `connect_fail` Failed connecting to the network.

- {package_success, Handle} Transfer was accepted by the network and is progressing, it was given Handle as id.
- {package_fail} Transfer was not accepted by the network.
- {package_transfer_success, Handle, Message} An incoming transfer with the given Handle was completed. Message contains the information that was sent.
- {package_transfer_success, Handle} An outgoing transfer with the given Handle was completed.
- {package_transfer_fail, Handle} An incoming or outgoing transfer with the given Handle failed to complete.
- {incomming_package, FromIP, Handle, Header} A pending transfer that is sent from FromIP with the given Handle. Header is what type of transfer it is.
- {transfer_progress, Time, Peers} At the given time this is the total upload and download to each and every neighbor.
- {dns_lookup_success, IP} The IP for the given DNS.
- dns_lookup_fail Failed getting IP for the given DNS.
- disconnect_success Disconnected from the network.
- disconnect_fail Failed to disconnected from the network.
- dns_register_success Successfully registered the DNS on the network.
- dns_register_fail Failed to register the DNS on the network.
- {get_my_ip_success, IP} The client's IP.
- get_my_ip_fail Failed to get the client's IP.
- abort_package_success Transfer was successfully aborted.
- abort_package_fail Failed to abort transfer.
- debug_state Print information about the state.
- debug_peers Print information about the neighbors.
- debug_pieces Print information about the pieces.
- debug_subpieces Print information about the sub pieces.

- `debug_chokelist` Print neighbors that are choked.
- `debug_kill` Disconnect permanently.
- `{debug_choke, IP}` Choke the given IP.
- `{debug_dc, Time}` Disconnect from the network for a given amount of time.
- `{get_state, ReplyPid}` Send information about the state to `ReplyPid`.
- `{webi_monitor, ReplyPid}` Send general monitoring information to the `ReplyPid`.
- `{webi_monitor_details, ReplyPid}` Send detailed monitoring information to the `ReplyPid`.
- `{update_log, Time, Heartbeats}` Client should update the log. `Time` and `Heartbeats` is when this transfer was sent.
- `{times, Time, Heartbeats}` What the time is for the network and what tick it is on.

Appendix B: User's Manual

Roland Hybelius

David Viklund

February 18, 2010

Contents

1	Runtime environment	1
1.1	Installation	1
1.2	Running the program	1
2	Configuration files	1
2.1	Network	1
2.2	Client	3
2.3	Tracker	5
2.4	Torrent	6
3	Command line interface	6
3.1	Manual client	6
3.2	Network commands	7
3.3	Client commands	8
4	Web interface	9
4.1	Network log	9
4.1.1	X-values	9
4.1.2	Y-values	10
4.2	Client log	10
4.2.1	X-values	10
4.2.2	Y-values	10
4.3	Monitor	11
4.3.1	Trackers	11
4.3.2	Clients	12
4.4	Dns Monitor	14
4.5	Individual Monitor	14
4.5.1	Statistics	14
4.5.2	Pieces and Piece color index	16
4.5.3	Peers	16

1 Runtime environment

1.1 Installation

There are a few things needed to run the simulator.

First off, you must get the source code. Once you have it you're going to get yaws. The simulator uses yaws embedded mode for hosting a web server to monitor the running simulation and to show logs. To get yaws, you can either download it using your package manager if you have one or download it from the yaws web page.

Thus, you will also need a browser to see the graphs and to monitor. Any browser supporting javascript will do.

To compile the simulator, you can use the Makefile provided in the source.

```
cd <sourceroot>/src/  
make
```

1.2 Running the program

To run the program, use the run-script `sourceroot/src/run`. There are two ways to run this script.

```
./run network tracker client torrent
```

where **network** is the network file to feed the network-module. **tracker** is the file with settings and the modular behavior for the tracker. **client** is the file with the settings and the modular behavior for the clients. **torrent** is the torrent file that both the clients' and the tracker will use.

or

```
./run configfolder
```

which will read in config files from **configfolder**. In that folder it has to be a file called **network**, a file called **tracker**, a file called **client**, a file called **torrent**. The files will be read in like the above case.

There is a few examples of the configuration files in `sourceroot/src/config/`

2 Configuration files

The configuration files are used to set up different network topologies and load different settings and algorithms for the clients. There must be one configuration file for the network, tracker, clients and the torrent or the simulator wont work. Some defensive code is provided if a mistake is done in the configuration files, but most often the simulator will crash and give an error print. The fields are on the form `SettingName=Type`.

2.1 Network

In the network configuration file you can specify how the network should behave and how the network topology is designed. In the file you can input the following.

`HeartbeatInterval=integer`: It is the number on how ofter the heartbeat process will send reminders to the network to progress transfers. Example: `HeartbeatInterval=50` means send every 50 milliseconds.

We would like to have this value as low as possible to increase accuracy of the simulation. But as the value goes down, the amount of CPU time needed goes up.

`SpeedupdateInterval=integer`: This is the time between the speed updates that the network sends out to the networkclients to inform them about how fast their up/down connections to the other networkclients are.

A low value will inform clients often of their download progress keeping them up to date. A high value will not inform the clients that often, but will save some computation power for the network.

Example: `SpeedupdateInterval=1000` means to send out speed updates every 1000 milliseconds.

`HeartbeatsToLive=integer`: Total amount of heartbeats the network should run before shutting down. 0 means infinite. When it's reached, the network process will finish but the web interface will remain running allowing the user to analyze of the graphs in the web interface.

`LogInterval=integer`: How often things are logged in both clients and the network. A higher number will increase work on the hard disk, but log files will be more accurate (and larger). Example `LogInterval=100` means to write to the network log every 100 milliseconds and tell all clients to write to their logs.

`Section=string`: Section **name**: This section should be present one time for every router in the network. the **name** in the title would be the name of the router. Sub values for this section is:

- `Net=string.string.string.string`:
`Net=x.x.x.x`
`x.x.x.x` is the IP-range of the router. The router itself will get the first IP-in the range, then all clients connecting to the router will have the remaining IP-addresses in increasing order. Any x:s will be interpreted as a range from 0 to 254. Example: `Net=192.168.x.x` will give the router IP: `192.168.0.1` and will distribute the other addresses available in the range. First client will get `192.168.0.2` and the last client will get `192.168.254.254`.
- `OtherRouterName=float, float, string, string`:
`OtherRouterName=Capacity, Errorrate, IP-destination, Netmask`
This defines a send connection to `OtherRouterName` with the given `Capacity`. `Errorrate` is not implemented. If an incoming package masks to `IP-destination` with `Netmask`, this is the route to choose. So this is a row in the routing table of the router.

This section is specified one time for each outgoing connection the router has to others (Incoming connection is specified as the other router's routing table).

Note: A Router Must not have multiple entries to redirect to another router in the routing table. If it does, the network will treat it as two different routes to the same host and will have too much bandwidth to that router, but it can only use one of the connections since the first match in the routing table will be the one used.

2.2 Client

Clients have two types of settings, normal settings that specifies for example the upload speed and algorithm settings. For different algorithms to be loaded a module name and function to be called should be specified. The module should be in the `mymodules` folder with `mod_` at the beginning of the name. Algorithms loaded from the configuration file like this are called custom functions and everything else custom settings. How to write your own custom function and how to access the information in a clients state can be found in appendix A. There is a `clientface.erl` to make this easier on the user.

The configuration file is divided into a `Section General` and `Section X` where `X` is any name. In the general section all settings must be specified except the amount setting, it should not exist in the general section. These settings are the default settings and if nothing else is specified in the sections below these are the settings that will be used. After the general section any number can be added of `Section X` with different `X:s`. Here the amount setting should be added, it tells the simulator how many client should be spawned under this section. Any setting specified in the general section can be added here as well, they will overshadow the settings in the general section. An example that does not include all section which in the normal case is needed:

```
Section General
  TimeUntilJoin=50
  BandwidthDown=500
```

```
Section Slow
  BandwidthDown=10
  Amount=1
```

```
Section Generic
  Amount=10
```

Under `Section General` it is specified that a client should spawn after 50 milliseconds and have a bandwidth of 500 to its closes router if nothing else is said. However in the `Section Slow` a new bandwidth is given and an amount of one. The result of this will be one client spawning after 50 milliseconds with 10 as bandwidth down. In `Section Generic` only an amount of 10 is given, this results in 10 clients spawning after 50 milliseconds with a bandwidth of 500.

The following list is the different settings that can be specified in the different sections.

- `MaxAmountOfPeers=integer`: The amount of connections a peer should be limited too. Meaning that the amount of neighbors a peer has in its neighborhood is capped by this number.
- `MaxUpConnections=integer`: Amount of peers that can be simultaneously uploaded to. Could be used by choking to decide how many to choke and keep unchoked. Otherwise it's not used.
- `MaxDownConnections=integer`: Amount of peers that can be simultaneously downloaded from. In Bittorrent this should not be restricted since the choking algorithms should handle it. Use a high value.

- `BandwidthUp=integer`: The peers upload speed to the closest router.
- `BandwidthDown=integer`: The peers download speed to the closest router.
- `TimeUntilJoin=integer`: A number in milliseconds which specified a time which the clients should spawn after.
- `Router=string`: What router the clients should connect to in the network. Look in the network configuration file to see the names of the routers that exists.
- `StartPieces=integer:integer-integer, ...`: What start pieces the clients should start with. The first integer is for the amount of pieces the other two are between what range the pieces should be. For example if 5:1-10 was given, then 5 random pieces between 1 to 10 will be given to the client as it spawns.
- `SubpieceSize=integer`: The size of the sub pieces.
- `ErrorRateUp=float`: The chances of a package getting corrupted while sending to the closest router. This is not implemented.
- `ErrorRateDown=float`: The chances of a package getting corrupted while receiving from the closest router. This is not implemented.
- `InitCustomState=mod_string, string`: `InitCustomState` is the custom function that should initialize the custom state. Read in appendix A what the custom state is. The first string is the name of the module to load and the second should be the function name to call.
- `Choking=mod_string, string`: Give the module name and function name to call. This is the custom function for the choking algorithm.
- `ChokingInterval=integer`: How often the choking algorithm should be called in miliseconds.
- `Antisnubbing=mod_string, string`: Give the module name and function name to call. This is the custom function for the anti-snubbing algorithm.
- `AntisnubbingInterval=integer`: How often the anti-snubbing algorithm should be called in miliseconds.
- `SnubbedDuration=integer`: The amount of time a client should be snubbed when anti-snubbing snubbs it.
- `Availability=mod_string, string`: Give the module name and function name to call. This is the custom function for the availability. Read about it in appendix A.
- `AvailabilityInterval=integer`: How often the availability function should be called in miliseconds.
- `NeighbourhoodCheck=mod_string, string`: Give the module name and function name to call. This is the custom function for the neighborhood balance. Read about it in appendix A.

- `NeighbourhoodCheckInterval=integer`: How often the `neighborhood_check` function should be called in milliseconds.
- `OptimisticUnchoke=mod_string, string`: Give the module name and function name to call. This is the custom function for the optimistic unchoking algorithm.
- `OptimisticUnchokeInterval=integer`: How often the optimistic unchoking algorithm should be called in milliseconds.
- `PieceSelection=mod_string, string`: Give the module name and function name to call. This is the custom function for the piece selection algorithm.
- `PieceSelectionInterval=integer`: How often the piece selection algorithm should be called in milliseconds.
- `Amount=integer`: This should not be specified in the `Section General`. It gives a number of how many clients to spawn with these settings.

Information about the file size and how large the piece size should be is read in from the torrent config file which is a simplified torrent file. Using this the clients will setup pieces and sub pieces.

2.3 Tracker

In the tracker configuration file one `Section X` should exist, where `X` can be any string. In this section the following settings and algorithms should be specified.

- `InitCustomState=mod_string, string`: The custom function that should initialize the custom state. Read in appendix A what the custom state is. The first string is the name of the module to load and the second should be the function name to call.
- `BandwidthUp=integer`: The trackers upload speed to the closest router.
- `BandwidthDown=integer`: The trackers download speed from the closest router.
- `MinAnnounceTime=integer`: The minimum amount of time in milliseconds a peer needs to wait before requesting new neighbors again.
- `MinRequestInterval=integer`: Minimum amount of time in milliseconds a peer needs to wait before sending a request to the tracker again. With an exception for event requests. Read appendix A.
- `AllowNumwant=boolean`: If true then peers are allowed to specify the amount of peers they want when they announce to the tracker.
- `Router=string`: What router the tracker should connect to in the network. Look in the network configuration file to see the names of the routers that exists.
- `Neighbourhood=mod_string, string`: Give the module name and function name to call. This is the custom function for the neighborhood function. Read about it in appendix A.

- `Availability=mod.string, string`: Give the module name and function name to call. This is the custom function for the availability function. Read about it in appendix A.
- `ErrorRateUp=float`: The chances of a package getting corrupted while sending to the closest router. This is not implemented.
- `ErrorRateDown=float`: The chances of a package getting corrupted while receiving from the closest router. This is not implemented.

2.4 Torrent

The torrent configuration file is a simplified torrent file where one `Section X` should exist, where `X` can be any string. In this section the following information should be provided.

- `Size=integer`: Size of the file.
- `PieceSize=integer`: Size of a single piece.
- `DNS=string`: The trackers DNS so the clients can ask the network what the trackers IP is.

Size and piece size will later on be used by the clients to calculate how the file should be divided into pieces. Calculating the last piece size which may differ if size divided with piece size is not an even number is considered as well.

3 Command line interface

When the simulation has started, you'll see a prompt which you can enter commands. All commands available is describe in this section.

3.1 Manual client

The interpreter can also connect to the network as a network client. If so, the user will need to do everything, like sending every single package and decide whether to accept or deny incoming packages. This is mainly for debugging special cases.

- `mail()`
Receives a incoming message from the process message inbox or `no_mail` if there was none.
- `mails()`
Receives all incoming messages from the process message inbox in the inbox.
- `mail_wait()`
Receives a mail and waits a while if there was none.
- `nws(X)`
Sends a Erlang message to the network process. `X` is the message to send.
- `nwsc(IP, Header, Message, Size)`
Sends a transfer with the given `Header` and `Size` to the host `IP`.

- `nwtstart()`
Sends a initial transfers to the tracker to join the swarm.
- `nwt(Header, Message, Size)`
Sends a custom transfer to tracker.
- `nwsd()`
Sends a deterministic undefined transfer to itself.
- `nwsd(IP)`
Sends a deterministic undefined transfer to IP.
- `whoami()`
Prints the IP of the manual client.
- `nwdc()`
Disconnected from the network.
- `nwc()`
Connects to the network.
- `nwc(Router)`
Connects to router on the network.
- `nwdeny(Handle)`
Denies a the incoming transfer Handle.
- `nwaccept(Handle)`
Accepts a the incoming transfer Handle.
- `nwabort(Handle)`
Aborts a transfer, incoming or outgoing.

3.2 Network commands

The following commands are to examine and to interfere with the network-process and to show the networkstate including transfers and routersystem.

- `go()`
Sends the go-signal to start the simulation.
- `commands()`
Lists available commands.
- `hb()`
Instructs the network process to do a extra heartbeat.
- `ping()`
Ping the network.
- `ct()`
Shows all confirmed transfers.

- `ct (IP)`
Shows all confirmed transfers involving IP.
- `ut ()`
Shows all unconfirmed transfers.
- `net ()`
Returns the network state.
- `netl ()`
Shows a summary of the network state.
- `rs ()`
Returns the routersystem of the network state.
- `rsc ()`
Shows compact summary of routersystem.
- `rss (N)`
Shows a description of the host specified by N. N is the number for the host give by rsc.
- `rsl ()`
Shows a list of the networkclients.
- `ipmap ()`
Shows the mapping of IP-addresses → hosts.
- `dnsmap ()`
Shows all registered DNS:s and their owners.
- `ncd (Host)`
Sends debug-message to the client Host.
- `nwu ()`
Shows the clients that has a updated speed.
- `nwstatus (Handle)`
Shows status of a package.

3.3 Client commands

The following commands are for interfering and displaying relevant information about some networkclient. Host is either a host name or an IP.

- `cs (Host)`
Show the state of the networkclient.
- `cpeers (Host)`
Show the peers the networkclient are connected to.
- `cpcs (Host)`
Show the pieces a bittorrent networkclient has.

- `csub(Host)`
Show the sub pieces a bittorrent networkclient has.
- `cchokelist(Host)`
Shows what peers the bittorrent networkclient has choked and unchoked.
- `ckill(Host)`
Instructs the bittorrent networkclient to disconnect from the network and never come back.
- `cdc(Host, Time)`
Instructs the bittorrent networkclient to disconnect from the network and never come back after `Time` milliseconds.

4 Web interface

Once you have fired up the simulator point your browser to `http://localhost:8888/`. You will then be greeted by the web interface of the simulator.

The simulation does not start automatically. It waits for the user to give the signal. This can be done by either pressing the start button on the web interface or to run `go()` in the command line interface.

In the web interface you can view information generated from the log files and you can view the status of the ongoing simulation.

The information about the network health is always shown in the upper-right corner of the page, there you can also stop the simulation or even restart it if it has stopped.

4.1 Network log

In this section you can view graph that are generated from the network log-file.

To generate a graph, you have to specify what values defines the x-axis and what values that defines the y-axis. The simulator will then generate (using Google chart API) a graph showing this information.

4.1.1 X-values

On the X-axis you can have one of the following definitions.

- `Network iterations` is the amount recursive calls by the main-loop in the network. It could also approximate the amount of Erlang-messages received by the network since that is what triggers the recursive call.
- `Real time taken` is the time taken in seconds the network has been running.
- `Heartbeats` is the total amount of heartbeats the network have been running. Skipped heartbeats are included.
- `Simulation time taken` is the time taken in seconds the network has been running calculated from successful heartbeats only.

4.1.2 Y-values

On the Y-axis the following definitions are available.

- `Packages Registered` is the total amount of transfers registered between the network hosts.
- `Skipped Heartbeats` is the amount of skipped heartbeats.
- `Connected Clients` is the amount of clients in the network at the moment.
- `Packages on Network` is the amount of pending transfers currently on the network.

4.2 Client log

In this section you can view a combined graph that are generated using the log files from a subset of the clients.

To generate a graph, you must select what sections of clients you would like to see. Also you have to specify what values defines the x- and y-axis. The simulator will then generate (using Google chart API) a graph showing this information. Also, you have to specify whether or not to show the Average values of the clients in a given section or to show all clients in the section individually. Note, If you have many clients to show and choose to have individual graphs, only a very low amount of coordinates will be reserved for every client. This will make graphs which does not really show much, so a better choice would then be to show the Average values of the sections.

4.2.1 X-values

Available definitions of the X-axis are:

- `Time (s)` is the simulation time taken in seconds the simulation has been running.
- `Network heartbeats` is the total amount of heartbeats done by the network.
- `ClientTicks` is the amount of ticks the clients' have made individually.

4.2.2 Y-values

Available definitions of the Y-axis are:

- `Peers` is the amount of peers the client has.
- `Choked peers` is the amount of choked peers the client has.
- `Unchoked peers` is the amount of unchoked peers the client has.
- `Seeders` is the amount of seeders the client is connected to.
- `Leechers` is the amount of leechers the client is connected to.
- `Pieces I have` is the amount of pieces the client has.

- `Successful received messages` is the total amount of messages the client has received successfully.
- `Failed receiving messages` is the total amount of incoming transfers that this client has gotten failed due to other peers.
- `Aborted receiving messages` is the total amount of incoming transfers that this client has aborted.
- `Successfully sent messages` is the total amount of messages sent to others that has been successful.
- `Failed sent messages` is the total amount of messages sent to others that has failed due to other peers.
- `Aborted send messages` is the total amount of messages sent to others that the client has aborted.
- `Total uploaded` is the total amount of bytes the client has uploaded (Including from those transfers that where aborted or failed).
- `Total downloaded` is the total amount of bytes that the client has downloaded (Including from those transfers that where aborted or failed).
- `Unused subpieces` is the amount of sub pieces the client has that is not part of any completed piece.
- `Snubbed peers` is the amount of snubbed peers the client has.
- `Choking peers` is the amount of peers choking the client.
- `Unchoking peers` is the amount of peers that is not choking the client.

4.3 Monitor

If you enter this section of the web interface you'll be greeted by two tables. The top table shows all trackers and the bottom table shows all bittorrent clients that has joined the network. This page will automatically update every once in a while to keep the information up to date. Notice that if the simulation stops while showing this page, the information on it will disappear.

4.3.1 Trackers

The top-most table on the monitor section shows the statistics for all trackers connected to the simulator. It' will show the IP-address of the tracker, the DNS registered, the router it is connected to and the amount of seeders and leechers on the swarm. Note that the simulator only supports one tracker per swarm. And using multiple trackers should be supported but it is not tested.






4.3.2 Clients

The lower table on the monitor section show statistics on all bittorrent clients that are connected to the network. In our case, that is all bittorrent clients in the swarm since we use only one tracker.

However, in this table there is one row for every client. And in this row a number of different things are visible. In the headers of this table, you can hover with the mouse to see a description of the column.

Example

Trackers											
IP	DNS	Router	Seeders	Leechers							
{10.8.0.2}	tracker.se	R1	3	22							

Clients											
IP	Section	Rou	Pcs	Piece map	Prs	Dl	Ul	Tot Ul	Tot Dl	Ratio	Sd
10.8.0.3	Faster_Pete	R1	1		13	6	4	485.4k	295.9k	1.64	
10.8.0.4	Generic	R1	0		14	10	0	0.00	246.1k	0.00	
10.8.0.5	Generic	R1	0		11	0	0	0.00	0.00	∞	
10.8.0.6	Generic	R1	0		9	8	0	0.00	262.7k	0.00	
10.8.0.7	Initial_seeders	R1	25		10	0	9	757.8k	0.00	∞	x

- IP The IP-address of the client

Example 10.8.0.3

Means that the IP address of this host is 10.8.0.3, in other parts of the program, it will be represented as 10, 8, 0, 3.

- Section The section in the config-file this client comes from.

Example FastDude

Means that the section in the client-configuration file this host is created from is called FastDude.

- Rou The Router this client is connected to.

Example R1

Means that the section in the network-configuration file this host is created from is called R1.

- Pcs

The Amount of pieces this clients has.

Example 4

Means that this client has 4 complete pieces.

- Piece map

The map of the pieces this client has.



The green area represents this client having that piece. We cannot see exactly what piece number it is, but we can approximate it to 10 by just looking at it (If we know

that the total amount of pieces are 24). The cyan shows that the client is downloading or requesting the certain sub pieces of the certain piece. In this picture, there are two different pieces the client is currently working on.

- **Prs**

The amount of peers this client has.

Example 4

This means that the client has a total of 4 peers.

- **Dl**

The amount of pieces this client is downloading at the moment.

Example 2

Means that the client is downloading 2 different sub pieces at the moment.

- **Ul**

The amount of pieces this client is uploading at the moment.

Example 3

Means that the client is uploading parts of 3 different pieces at the moment.

- **Tot Ul**

The total amount this client has uploaded. Note, only successful sent pieces count here.

Example 485.4k

Means that the client has uploaded a total of approximately 485,400 bytes.

- **Tot Dl**

The total amount this client has downloaded. Note, only successful sent pieces count here.

Example 295.9k

Means that the client has uploaded a total of approximately 295,900 bytes.

- **Ratio**

The ratio between upload and download. (total upload divided by total download).

Example 1.64

Means that the total upload / total download = 1.64.

In our example the numbers are: 485,400 / 295,900 = 1.64.

- **Sd**

A x here indicates that the peer is a seeder.

Example x

4.4 Dns Monitor

This is the same as the monitor menu, except it only monitors the peers that have a DNS registered (Typically trackers). The primary use of this is that the web interface does not interfere that much with the running simulation since it shows little data. This menu also updates automatically.

4.5 Individual Monitor

In the Monitor or the Dns Monitor tab of the web interface, all clients are shown. If you click on the link on the IP-address column, that particular client will be monitored, alone with more details. This page will also update automatically. However, this page is divided into 4 parts. Piece color index, Peers, Statistics and Pieces.

4.5.1 Statistics

Here all general things that are relevant in the client's state are shown.

Example

10.8.0.4	
Statistics	
ip	10.8.0.4
section	Generic
bwup	30
bwdn	30
router	R1
amount_pieces	15
total_amount_pieces	25
subpiece_size	17000
tracker_dns	tracker.se
tracker_ip	10.8.0.2
tick	1268
total_upload	1686000
total_download	2434000
up/down ratio	0.692687
seeder	false

- ip

This is the IP of the client.

Example 10.8.0.5

- section

This is the section in the client-configuration file that the client was created from.

Example Generic

- bwup

This is the upload capacity that the client has to the router it is connected to.

Example 30

- bwdn

This is the download capacity that the client has from the router it is connected to.

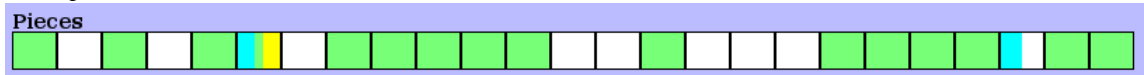
Example 30

- `router`
This is the name of the router that the client is connected to.
Example R1
- `amount_pieces`
This is the amount of pieces that the client has.
Example 0
- `total_amount_pieces`
This is the total amount of pieces that the torrent-file contains.
Example 25
- `subpiece_size`
This is the size in bytes of the sub pieces.
Example 17000
Means that every subpiece is 17,000 bytes large.
- `tracker_dns`
This is the dns to the tracker as specified in the torrent-file.
Example tracker.se
- `tracker_ip`
This is the IP address of the tracker that has been looked up in using the DNS above.
Example 10.8.0.2
- `tick`
The amount of ticks that the client been through.
Example 94
- `total_upload`
The amount of bytes in total that the client has uploaded.
Example 0
- `total_download`
The amount of bytes in total that the client has downloaded.
Example 17000
- `up/down_ratio`
The ratio between upload and download defined as Upload / Download
Example 0.000000
- `seeder`
The truth of the fact that the client is a seeder.
Example false

4.5.2 Pieces and Piece color index

Pieces is the map that the client has of all pieces and sub pieces. Different colors indicates different states. Yellow, for instance, show that the client is about to start downloading the subpiece. Other color and their description can be seen in the Piece color index.

Example



In this example we see that the client has quite many pieces (Lots of green boxes) and two pieces querying/downloading.

4.5.3 Peers

In this part of the page, we can see the information about all peers that the client has.

As the other tables, you can hover over the headers in each column to see a description of it, also note that the columns are grouped together with similar columns for readability.

Example

Peers												
se	IP	Choking			Speed		Interest		Pieces	Transfers		
		cg	cd	ou	sn	in	out	ig		id	pin	pout
	10.8.0.3			x		9.160	0.000	x			23 ¹ 23 ² 23 ³ 23 ⁴ 23 ⁵	
x	10.8.0.7					4.260	0.000	x			6 ¹ 6 ² 6 ³ 6 ⁴	
	10.8.0.9		x			2.625	1.545		x			25
	10.8.0.11					0.590	6.140	x	x		6 ⁷ 6 ⁸ 6 ⁹ 6 ¹⁰	25

A brief description of the columns follows.

- se
Seeder. Checked if this peer is a seeder.
- IP
IP. The IP-address of the peer
- cg
Choking. Checked if this peer is choking this client
- cd
Choked. Checked if this peer is being choked by this client
- ou
Optimistically unchoked. Checked if this peer is optimistically unchoked by the
- sn
Snubbed. Checked if this peer is snubbed
- in
Speed in. The speed this peer is sending to the client
- out
Speed out. The speed the client is sending to the peer

- `ig`
Interesting. True if this peer has some interesting pieces
- `id`
Interested. True if this peer think the client has interesting pieces
- `Pieces`
Pieces. The pieces this peer has.
- `pin`
Incoming pieces. The pieces and sub pieces the client is downloading from this peer. Note that the colors and numbers are the same as in the previous Pieces section.
- `pout`
Outgoing pieces. The pieces this peer is uploading to the client. Note that we cannot see what sub pieces are being downloaded. This is because the sub pieces is irrelevant since other peers may have different sizes of their sub pieces.

Appendix C: Bittorrent Simulator in Erlang

March 21, 2009

In this exjobb the students will write a simulator for bit-torrent like protocols in Erlang. The exjobb has four main phases

- Reading up on Erlang and the Bit-torrent protocols.
- Implementation of the simulator
- Experiments
- Writing up.

Specification of the simulator

The goal of the simulator is not to simulate exactly the bit-torrent protocol but to simulate bit-torrent like protocols, so it should be flexible and programmable.

The simulator should include features for:

- Specification framework and simulator of bit-torrent like protocols
- A network simulation, you should be able to simulate different network topologies with different connection speeds.
- A test harness to be able run many large scale tests from shell scripts

Experimentals should include:

- the ability to change the protocols to see the effect;
- the ability to be able simulate different topologies and connection speeds.
- Variations on the Bittorrent protocol should include:

- Different packed selection strategies;
- Variations on the Tit-for-tat peer selection algorithm.

A very rough estimate 60% simulator and 50% experiments.
10 weeks total each.

- 1 week - reading
- 1 week - code design
- 4 weeks coding
- 2 weeks experiments
- 2 weeks writing up.