IT 10 051
Examensarbete 15 hp
Oktober 2010

A Bittorrent simulator in Erlang - continued

Niclas Stensbäck

Institutionen för informationsteknologi
Department of Information Technology

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

A Bittorrent simulator in Erlang - continued

Niclas Stensbäck

In an effort to improve the work of a previous Master thesis, in which a bittorrent simulator was constructed in Erlang, a new, modular and distributed network module was implemented, documented and incorporated into the application which was written by two students.

Although no new simulations were run, and some matters concerning the timing measurement and the presentation of data remain, a robust and transparent new network implementation is in place to make the running of this Bittorrent simulator more efficient and more realistic.

Included in this thesis are some explanations of the structure of the thesis ``Bittorrent simulator in Erlang'', in order to highlight some of the difficulties and considerations that were made in this thesis work.

Handledare: Justin Pearson
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 10 051
Tryckt av: Reprocentralen ITC

Contents

Abstract

In an effort to improve the work of a previous Master thesis, in which a bittorrent simulator was constructed in Erlang, a new, modular and distributed network module was implemented, documented and incorporated into the application which was written by two students.

Although no new simulations were run, and some matters concerning the timing measurement and the presentation of data remain, a robust and transparent new network implementation is in place to make the running of this Bittorrent simulator more efficient and more realistic.

Included in this thesis are some explanations of the structure of the thesis "Bittorrent simulator in Erlang", in order to highlight some of the difficulties and considerations that were made in this thesis work.

1 Introduction

The Bittorrent protocol is one that has been subject to much discussion, not only for its merits and legal implications. Many theorize about the viability and indeed possibility for game theory applications to hold their ground in a setting where a tit-for-tat behaviour is wanted but not (necessarily) strictly enforced.

The Bittorrent protocol is very well documented online and in articles and so this thesis is not going to go into any in-depth exploration of it. Simply put, a bittorrent file distribution will start with a tracker centrally organizing different hosts, connecting them to each other and giving them addresses to people who already possess whatever parts of a file you want to download. A seeder is someone who possess all parts of the "torrent", the file or files being distributed, and a leecher is one who does not. Different strategies and tools are used to control the flow of data and to maximize general throughput and individual throughput.

Due to the distributed nature and largely voluntary sharing mechanism of the Bittorrent protocol, it lends itself well to discussions on game theory. As such, a simulator which is able to simulate many different scenarios, behaviours and approaches would be quite useful to be able to experiment with this protocol and test hypotheses.

The distributed way in which Bittorrent works also lends itself well to an implementation that functions well in a distributed manner, and Erlang is a natural choice for implementations of these kinds of applications. Erlang makes spawning new processes easy, processes are lightweight and communication between them is very quick (and simple) as well. [1] Since Erlang is a functional language, it is quite a joy to code in as well so it is very understandable that the first thesis chose it to implement the bittorrent simulator and a part of the motivation of this thesis as well.

A thesis that was originally intended to be two computer science bachelor theses at UU was started in early 2009. The two master students programmed an application which had as its lofty goals to be not only a modular and useful simulator of the behaviour of the bittorrent protocol, but also to make use of the popular distributed nature of the programming language Erlang. Though parallelism was their goal, the result at the end of their thesis had some issues:

- The entire network structure is contained inside a single process that serves as the routing system and as the "heart" of the network, making time pass and handling all meta calls from the nodes to get various meta-information as well as calculating the speeds of all transfers in the network.

- The notion of time passing for the network is divided into discrete time-steps which makes it hard to assess the actual speeds of things.
- The time controllers are actually timers running against the system clock, making the simulation run in real-time at best, and any lag in the "heart" makes the clients behave strangely since time passed in the network simulation does not match up with their timers.
- The bandwidth of the nodes in the simulation are quite arbitrary, with the actual bandwidth of the nodes in the network corresponding rather strangely with the perceived performance of the network.
- The code in and of itself is hard to read, with a giant client state file which contains nothing but global def:s to access elements in the 100-elements or so large state tuple and many examples of function call chains. These quickly become obtuse and makes following the code rather hard.
- The contact surface between different parts of the code is quite large. Many different functions are used to send packages to the network and to receive them. This is not such a big problem for the application itself but it makes it unnecessarily complex to understand and modify.

2 Problem Description

The main goals of this thesis were, as stated in the original submission of the problem description, to:

- Rewrite the network core according to a distributed model so that the network is not just made up inside the network process but actually consists of distinct processes and modules.
- Give the simulation some reasonable notion of time, which the heartbeats really did not permit.
- Rewrite things in the implementation that do not make much sense
- Do some simulations of the Bittorrent protocol to test the network if time permits.

Rewriting the network core was the main goal and from the start this was foreseen to be the main time sink of the thesis as well.

Early on in the planning stage of this thesis, it was clear that the structure of the network should be as streamlined as possible. However, it was also important to still be able to facilitate the nodes that were already implemented in the network. To do this, a few modules and key structures were described in the outlining of the work of the thesis.

An understanding yielded by analysing the source code of the original thesis for some time has given rise to the following description describing the way the network used to work, which is given mostly for historical reasons if someone wants to understand the original implementation.

- The start of the simulation consists of the main module spawning the network process and the master client.
- The network process (which actually spawns the master-client and tracker process) simply sets up the variables set in the configuration file, then goes into $tick/1$, waiting for messages from the network nodes, whilst the *masterclient* module spawns all the clients in the network, with some random delay between different types of clients. Clients are spawned in groups, where each group follow a generic order with some alterations from the standard client to make them different from other groups. These groups are then spawned together sequentially by the *masterclient*.
- A client, when started, tries to connect with the network by sending a connect-message to the network process, where it specifies what router it wants to connect to as well as it is bandwidth and error rates. Assuming the router specified has not allocated all of the IP-addresses in it is specified range yet, the network sends back a connect_success message.The client will then attempt to locate the tracker via a dns lookup before initiating the bittorrent protocol.
- The tracker starts to communicate with the network much in the way that a normal client does, first by setting up a state given the configurations from the files, attempting to connect with the network process then requesting an IP from it. The tracker then asks the router to register a dns name corresponding to the URL of the tracker given by the config file. After this is done the tracker simply listens for messages from clients and handles them appropriately, adding clients to the swarm, sending out neighbour request responses and MRI (Minimum Request Interval) and MAT-values (Minimum Announce Time) to nodes connecting to it. After clients receive these values they will periodically request new peers to add to their neighbourhood.

This is quite a large piece of software, and runtime documentation is covered rather sparsely in the original thesis, so a part of this thesis goes into describing this system, mainly the network nodes and not so much the network itself since it is bypassed by this thesis work. The bulk of the thesis, however is the description of the new modules that make up the new network structure and their interactions with the old client implementation.

2.1 Nodes in the network

2.1.1 Clients - Leechers and Seeders

Every client is their own process, and they run differently depending on how the client configuration file is written, with some randomness in the code to simulate client latency. As such, analysing the runtime behaviour of the clients, and especially why some parts of the code are written the way that they are is quite hard. Apart from this, the general design of the client is not too complex. The client will, sequentially, try to request pieces from other nodes in the network, decide upon choking their peers, look for new neighbours, send out alive messages and check their incoming transfers (see picture included from thesis). Although this behaviour is rather apparent when viewing the thesis paper as well as the main code itself, finding out what goes on within each part of this loop is not so obvious. This is a partial, inferred understanding of the clients' communication with and over the network.

- When the client is started, the first things that is transferred via the network are some meta calls to signal to the network that the client has connected.
- After this, the client will connect to the tracker, getting the minimum request and announce times (how often the client can request or announce things to the tracker).
- Every so often after this, the client will request new neighbours from the tracker.
- Whenever new peers are added to the clients neighbourhood, it will initiate a handshake with these clients.
- Whenever a handshake is initiated, the two clients will exchange interested lists, the pieces of the torrent that they are interested in downloading, and send requests for packets to each other, if applicable.
- Every so often, the client will send out an alive-signal to the peers in its neighbourhood, to let them know that the client is still alive.
- A client will send a [$finished$] packet to the tracker when it has gotten hold of all pieces of the torrent, and declare itself to be uninterested in all connections that it has, becoming a seeder. Until it gets to this point it will periodically send requests to its neighbours for packets and act as its client functions dictate, which depend on it is custom functions.

The way that a Bittorrent client behaves may differ quite a bit depending on the implementation of the client and user selected options. To accommodate this, the implementation of the old thesis provided for developers to construct functions to implement client behaviour. For more info on these, see the documentation for the original thesis.

2.1.2 Packages, or Packets

Whenever a client in the network wants to communicate with other nodes (or the network itself) it uses the so called $networkiface$ module. In the original implementation, this module uses various fields of the clients state to send information to the network process (registered in the main module to the atom *network*). It then recieves either a confirmation message along with a handle corresponding to the message or a message letting the client know that the sending failed along with a cause, probably that the packet was adressed to an incorrect (or non-existing) IP. The handle is saved in a queue inside the client state and maintained in the state to keep track of messages that it is sending (and receiving). Apart from receiving a confirmation when the transfer is sent (along with which the handle is provided), a client will also receive a confirmation when the transfer is either dropped by the recipient, sent successfully or lost in the network. It will also receive updates regarding the transfer progress of the package and any packages that other clients want to send to it. The client needs to either accept or drop these packages before they are sent.

3 Methods

Although this understanding of the system did arrive gradually as part of the new network grew to fruition it became clear that simply gluing a new network core in the place of the old one would not be that simple. Many changes were made to the clients as well as the network interface module of the system. The design choices of these should be apparent under each heading below.

The new network implementation was envisioned to be rather modular, much because of the fact that the division of labour in an application should be clear. An expansion of the network or changing its behaviour therefore should not be hard to implement. Hopefully the documentation below should give a good idea of how things are structured and why.

3.1 The packet record

Early on the decision was made to have a shared record that the different modules would use to make sure that the data followed the same pattern and that the required information would be available when needed at any given time. This record grew as time went on and the packet record contains information about the IP:s of the sender and destination as well as the current sender of the packet. The original format of having header, handles and content was also included in the record in order to better comply with the expectations of the tracker and client implementations already in place. The records are constructed and deconstructed in the networkiface module and handled by the link and router modules seamlessly.

The life of a packet consists of:

1. Construction, the sender constructs the packet, setting the relevant values to their wanted values (*sender* and *original_sender*, importantly, should be $self()$).

2. The packet is sent to the *link* of the client and will be forwarded to a recipient (via any number of routers).

3. When the packet is routed through the first router, a handle is generated by the router (consisting of the PID of the sender and a counter which is the amount of received transmissions from the client).

4. When the packet is routed through the last router (the one the recipient is connected to), $accept/denyincomingtransfer$ packets are sent to the recipient and the packet is sent through the last link. If the recipient wants to drop the packet it can do so when the packet is received.

5. The packet is deconstructed in the networkiface, and a message is extracted that the client is familiar with and returned to the client implementation.

3.2 The Link module

This module was originally envisioned to be the main bottleneck of the system where delays were put into place to simulate the lag of internet traffic. The class as written is a very simple "pass it on" module that maintains a state consisting of the links two ends and the information needed to calculate waiting times. As the thesis work went on, additional behaviour would occasionally be implemented in the link module and subsequently moved out into the router module, since the idea is that the link should be as "stupid" (and therefore general) as possible. This is not only to make the code prettier and easier to navigate, but also because a design choice was to try to make this network modular in the sense that the client side should be possible to alter without any fuss with the network side.

The module maintains state through a simple record which (at the moment) consists of the PID:s of the two processes the link connects and the bandwidth of the link.

3.2.1 General functionality

The module is initialised by spawning a process running the init function. The two arguments are the processes that the link connects. These nodes can now send packets to the Link process and expect them to be delivered to the other node in a timely manner.

3.2.2 Sleep function

The sleep function is a simple implementation where the link-process will sleep for $(Size/Bandwidth) * 1000$ (rounded) miliseconds before sending the message.

3.2.3 Lossiness

Lossiness is at the time of writing disabled in the Link module. Initially this choice was made because of uncertainty of how the nodes would be notified of sending failure. As lossiness is implemented, the clients and tracker should be able to handle this as well as they would in the original implementation. This is one of the reasons that the *original_sender* field of the packet needs to be set when it is created, because messages are sent to that PID when the packet is dropped by a link.

Note, however, that it is unclear if lossiness is actually implemented in the old network, so if lossiness is handled by the clients or works at all in the current network client implementation is not certain.

3.2.4 packet_noreply

An additional field was created to allow routers and other nodes to communicate with each other without having to deal with the bandwidth issues and packet loss. Meta-packets that the nodes send are the main users of this functionality, and this is used by the *send*/1 function in the networkiface module.

3.2.5 Using the Link module

To use the link module, spawn a process running the *init*/3 function in the *link* module. The first two arguments are the PIDs of the two nodes you want the link to connect. The third argument is the bandwidth of the link. Then, you simply send messages to the link looking like: *packet*, *#packet*, a tuple consisting of the atom *packet* and a record of the packet type, and make sure to set *sender* and *original_sender* to *self*(). If the packet is transmitted successfully (it is, since packet loss is not implemented at the moment) a message looking like:

```
{packet, #packet{header=package_transfer_success,
handle=Packet#packet.handle}}
```

will be sent to the original process, (the handle will be sent to us when the packet reaches a router).

Here is a simple example which shows how to set up and use the Link module. Simply spawn the processes which want to be connected and have them wait for the Link Pid to be sent to them and then have them send their messages to that Pid.

```
X = spawn(fun() ->
      receive X ->
      Packet = #packet{sender=self(), size=100, content=ping},
      X ! {packet, Packet},
      receive {packet, #packet{content=pong}} ->
      io:format("X got pong~n")
      end
      end
         end),
Link = spawn(link, init, [X, self(), 1000]),
X ! Link,
receive {packet, #packet{content=ping}} ->
    io:format("self() got ping~n"),
    Link ! {packet, #packet{sender=self(), size=100, content=pong}}
end,
done.
```

This will cause X to send a ping message through the link to *self*() who will then send a response pong to X, also through the link. The link is fairly invisible, you simply consider the Pid of the link to be the Pid of the process on the other end and wrap messages in a tuple like specified.

3.3 The Router module

The module that turned out to be the beefiest of the ones coded during the work of this thesis is the router module. Though simple in its idea, many pieces of code were needed to come together for the routing system to work properly.

The general idea of the routing module is fairly straightforward: receive packets from links, look up the IP of the recipient, and forward it to the relevant host. In practice, things turned out to be a bit more complicated. In order to generate routing tables, a variant of Dijkstra's algorithm was implemented (see below). This is used to make sure that routers can forward packets properly between nodes, despite the fact that a route may not be apparent just from looking at the destination (since a routing a packet may require multiple intermediate hops to routers). Apart from this many things that were expected by the clients to be performed by the network are now performed by the router process that they are connected to, stuff like IP assignment as well as some functionalities which perhaps are not entirely obvious that they should be handled by the network. The router contains a function called *handle_metacalls*/2 which handles all this meta-data that the clients want to send to and receive from the network. The idea is that if you wish to alter the clients in some way you only need to look into and alter this function to make the network respond and send the required data accordingly.

3.3.1 Forwarding packets

The routers' main job is to forward packets for the clients. To do this successfully there are some considerations that need to be taken with regard to the expected behaviour from the network.

First of all, when a packet is first received by the router (i.e. when the *original_sender*-field of the packet is the same as the *sender*-field), the handle of the packet is generated for and a *packet_success* message is sent to the node, containing the the handle. This is expected behaviour from the network by the clients and is implemented in the *check_first*/2 function in the router module.

Whenever a packet is forwarded, the sender field is changed to the routers PID. The main reason for this is because of the link module, since it uses that field to check who to send the package to, but it also has implications on the design of the packet-record. The need for an *original_sender*-field becomes apparent when you consider this for example, something which was not realised in the beginning of the thesis work.

If the router finds that a packet sent to it was a normal packet (i.e. not a meta-packet), it attempts to route it to it is destination. This is done by two functions, *route* and *route*2, which search through the clients and router connection collections respectively to find the right host node. The *ets_map* collection of Clients use the IP as the key value of the connection tuple, so it is a simple matter of looking up the IP-value in the table to find the right connection. Routing for a node which is not immediately connected to the router is a bit harder, since the destination IP needs to be checked against the range of all the other routers connected to it. The router simply does a linear search of the router connections to find the one with the right range, using the method *within_range*/3 (which is actually reused from the old thesis).

3.3.2 Bits and pieces in the router module

The router process maintains a state via their *tick*/1-function, using a record that contains some information about the router itself, the connections it has to other nodes. Right now the only book keeping the router does is a counter keeping track of the number of sent packets. This is done by in the function *sent_packet*/1 which increases the value of this field.

The router uses a record called *connection* in order to manage the data saved for routing purposes and what it knows about the nodes connected to it. This record is used both for routers and 'normal' nodes but the way that the router uses them differs between the two. Notably, the $rangefloor$ and $rangeroof$ fields are the size of the network that a connected router can contain nodes within, $rangefloor$ is the first IP-address and $rangeroof$ is the last IP address which is assignable. These values are used in the routing function to route packets to hosts that are not directly connected to the router (that is, routing packets to other routers), while the *ip* field is used to search for clients that are directly connected to the router.

3.3.3 Using the Router module

Use the *routing* module to spawn the routers according to the network config file given to the simulation. The router-processes will be spawned and go into the hold/1 function where they will wait for a message looking like so: $\{packet, \#packet\{content = go\}\}$ where *#packet* indicates a packet record. You can send *start_go* to any of the routers to initiate all of the routers that this node is connected to (including itself). Until this is done clients can be added by sending messages to the routers looking like so: {*add_client*, *Link_pid*, *Client_pid*}, and routers via a corresponding *add_router* message. Note also that any messages sent by clients will not be handled by a router which is in the *hold*/1-loop, the router needs to be started first by a *start_go* message.

```
Ip = {128,0,0,1},
Floor = {128,0,0,2},
Roof = {128,0,0,254},
Rid = spawn(router, init, ["R1", Ip, {Floor, Roof}]),
Link = spawn(link, init, [self(), Rid, 100]),
Rid ! {add_client, Link, self()},
Rid ! start_go,
Link ! {packet, #packet{sender=self(), content={get_my_ip, self()}}}},
receive
  {packet, #packet{content={get_my_ip_success, Ip}}}->
     io:format("Got Ip:~w from Router:~w! ~n", [Ip, Rid])
end.
```

The above code will set up a router process with the internal network 128.0.0.X, meaning it can hold 252 hosts within it. A link is spawned between $self()$ and the router and is the connection is added to the router. A message is sent requesting the IP registered with the router and is summarily returned.

Note that the router does not send the client any information about it like the client's IP unless asked the client asks for it. For a list of meta-calls that the router module handles, see table 1.

3.4 The Dijkstra module

The Dijkstra module implements the spawning of the network links as specified in the configuration files. The router processes are spawned and connections between them are generated along with other necessary data. Data is supplied from the network configuration file in the same way as the old simulation (although packet loss is not taken into consideration). Then, an implementation of Dijkstra's algorithm is performed with each router being the source node of a run of the algorithm, generating the routing tables and the preferred route of the networks. This notion of routing has no consideration for route congestion but works well for the networks that have been simulated so far. Dijkstra's algorithm (citera AD-boken) is optimal when run from a single source and works well for the number of routers that can reasonably be expected to be simulated in the network. The effectiveness of this module is not optimal, for sure, in part

Table 1: Router meta package handling

Keyword	router module meaning
connect	The client wants to connect to the router. This was implemented mainly for checking that the clients were communication correctly in the.
get_my_ip	The client wants to know it IP. Router finds the clients entry in its clients table and returns the corresponding IP.
register_dns	This is used only by the tracker when it wants to register it's IP to a URL. This is saved in the router state and all routers connected to it.
dns_lookup	A client wants to look up the adress of the router. It is extracted from the routers state and returned.
add_dns	Another router that this router is connected to wants the router to add the tracker IP to it's state.

because no information about routes is kept in between the different runs for different routers. This means that a lot of the results are being recalculated for each router. Again, the size of the expected network and the amount of time it would take to figure out a more clever algorithm was seen as reasons enough to be satisfied with the current implementation.

3.4.1 Using the Dijkstra module

Simply run the *dijkstra*/1-function in the module and, if supplied with a configuration file in the required format, the function will spawn the router processes and construct the connections between them.

3.5 Alterations made to Clients

The main changes that were made to the Client side of the code (i.e. the nodes which are not the tracker) were done simply to add the link-field from the client-state to function calls to the network interface. The client state had to be modified slightly as well to add the link to the client state. Other than that, only a few changes to the actual functionality were made.

One of these alterations is the *init*/1 which now waits for a link PID to be sent to the client process before it starts its life cycle. This is because the link needs to have the PIDs of the two processes connected to it in order to be spawned correctly, so the client process, when spawned, simply waits for the link to be sent to it.

3.5.1 pkgcom

The package handling module of the client side is a bit messy with very large methods consisting mostly of case switches, so alterations here were kept to a minimum so as to not break anything. Most of the changes that were made were adding the links from the client state to all calls to the *send* methods in the network interface module. An interesting issue that occured during the development was the handshaking mechanism of the clients. Whenever a client gets new peers from the tracker to add to its neighbourhood it will attempt to send a handshake packet to them in order to initiate transmissions between them. What used to happen was that this would result in a livelock in testing when the network contained only two nodes. Each node would send a handshake request to the other, and upon receiving the other's request would drop the packet since the client had already sent a message to that node. This message is implied to be a handshake and this resulted in the two nodes continually dropping each others handshakes, waiting for the other to accept one. This was 'fixed' (with no known side-effects) by simply removing the requirement that no messages must be destined to the initiator of a handshake. It is unclear why this bug was did not show in testing with the old network, since the issue would seem to have still been there.

Other than that few things were changed in the *pkgcom* module, since (thankfully) the layering between it and the $networkiface$ module is pretty robust.

3.6 Alterations made to networkiface

The $networkiface$ module is of great importance to this thesis and many changes were made to it, although none of them too great so as to not disturb the functionality of the clients interfacing with it.

- The *connect*/7 method was altered to be a dummy call, since it is assumed that no client can be started without receiving a link to the network, so connecting to the network is superflous in this new implementation. Since we do not trust the clients to not do things before sending a connect message to the network the clients instead wait for the connection to be set up in their *init* functions.

- The two *send* functions were altered to take the link PIDs of clients as arguments in order to have a destination to send data to. The functions were also altered in order to send correct packets that the Link would recognise. *send*/1 deservers special mention since this function is only used in the original thesis to send meta-packets to the network, so a special flag, *packet_noreply* is used when sending it to make sure that the packet is not lost and the Link is not bogged down by handling it. This may or may not be considered preferable but would be easy to rever, simply remove the _noreply part from the tuple in *send*/1.

- Initial attempts were made to try to move the reception of all messages into a separate function, in order to make the code easier to manage and alter, but were discontinued, since it would make the code quite no easier to read and understand, and because of problems when implementing the pattern matching needed for this. All of the receive-statements were instead altered to handle and deconstruct packets sent from the network, converting them to the format that the client implementations would expect.

3.7 Alterations made to masterclient

An important part of the joining together of the new network and the clients was the masterclient module. It is responsible for spawning the clients according to the specifications in the configuration files and does so, starting the part of the simulation where the actual work goes on. The modifications made were not very complex however. The *init*-function was altered to take a collection of the routers as an argument and send the message to the routers to start them running, since the cliens will not (currently) wait after being sent their link before starting their execution.

The master client uses the client configuration entries when constructing the clients, so the bandwidth field and the router structure is used to construct a link between each client and the correct router. A message is sent to the router to add the client to its routing table. The master client is the one adding the client to the router and constructing a link between them because it's the process which is "highest" in the hierarchy of the modules that has the information required to do the job, rather than sending more data to the client and have it do more things which are not its main objective. Rather than passing data downwards or upwards, the masteclient handles this part of the code.

3.8 Alterations made to Tracker

Although the tracker is critically important to the simulation, it's code base is a lot smaller and more manageable than the bittorrent clients'. The same sort of changes were made to the tracker as to the clients, mainly the insertion of the link into the state of the tracker and inclusion of it in function calls to the network interface.

4 Result

A rather big part of the thesis was spent searching through the codebase and trying to figure out where things actually got done in the network, and subsequently modifying already written code to cope with the behaviour of the software already in place. This may well have been avoided by planning the software better, however, the architecture of the network was hard to understand without actually running the clients. The focus of this thesis was to recode the network so alterations of the client-side of the application was kept to a minimum. Although some documentation was provided via the report of the thesis, understanding and altering the structure of the clients would take a lot of time, so focus was kept to the network side, with only minor changes made to make the network side possible with the structure chosen. At the same time, the possibility to alter the client side without having to look at the network code was also kept in mind, so the contact surface between the network and the client has been kept to a minimum.

As a result of this, the clients function much the same as they did in the old network, the only real change was to accommodate the packet record, adding the link to their state and handling the *packet_success*-message.

4.1 packet_success

Whenever a client sends a message over the network, it immediately receives a response from the network which contains a handle for the message. This handle is then saved in the client and used to refer to the message in future transactions. The network then sends a message to the recipient asking if the node wants to receive the message (given the header and sender IP of the message). If the recipient wants to receive the message it's added the network and, when delivered, another message is sent to the sender informing it that the message was delivered.

This part of the network was mostly cut out of the new implementation. This traffic does not make much sense on a low-level view of the network (though it's still supported) so it's kept in on a minimal level. These packet successes are still sent and people do reject packets, but at least parts of it are sent directly to the processes themselves, alleviating the network. Since these message did not take any bandwidth in the old representation, it only makes sense to do it this way. To remove them from the clients themselves would be a massive job that this thesis was not intended to cover so the network has been made to deal with it instead.

Most of this handling of traffic that, in retrospect seems unnecessary is instead sent directly to and from the clients themselves in order to not bog the network with the unnecessary data flow.

4.2 The Link module

The fact that links are bidirectional as implemented today is a bit problematic, since bandwidth (in the old simulation and in reality) does not behave that way. The link module only contains one bandwidth field, which is used for messages going in both directions. An alteration to this would not be too hard to implement, you can either change the link to have two bandwidths (one for each direction) or make them one-directional and spawn two links for each connection. Incidentally, this is already done for the routers because of the nature of the dijkstra implementation.

4.3 Comparison with the old network

Since the timing mechanism of the old network was so different from this one they are quite hard to compare to each other. The old network ran partially in real time, with the clients book-keeping and timing being run by simple erlang timers. The network process itself, however, ran by heartbeats that ran at some interval and processed data at some (unclear) speed. Apart from making progress hard to measure, this also made the network slow down when under stress since clients would not wait for the network to finish frames before sending out new requests to it. Inevitably the transfer queue will be bogged down and new frames will be calculated slower and slower.

The new network, on the other hand, simply runs all processes in real time and delays are added only in the client behaviour themselves and in the link modules. This also makes simulations over a longer span of time more reasonable since it would be rather simple to implement a clock that ticks forward and tells all links that time passes.

4.4 Transfers

The speed of transferring in the old network is not really translatable to real speeds. The standard speed of the clients as specified in the client specification file is '30', but it's unclear how fast this is supposed to be in real life. In the new implementation the time taken to send a packet of size $Size$ over a link with bandwidth $Bandwidth$ is simply defined as $(Size/Bandwidth) * 1000$ miliseconds. This is a simple implementation and does not work too badly, but the amount of messages that the clients send to each other in order to transfer data makes this implementation cumbersome. A good change would be to make multiple simultaneous transfers possible over one link, for example by slowing the link for each transfer added to it and speeding it up for each transfer leaving it. This will simulate the behaviour of a real network connection more realistically.

However, the notion of a 'package' in the old implementation is also different from what you might expect, since the old network would transfer these packages in streams over a prolonged period of time and send transfer updates periodically based on this. This gives the clients the impressions that these transfers are not so much packets, but more a large sum of packets that are transfered over the network. This is not reflected well in the new implementation since all the data is sent along the network together at the same time, being delayed by all intermediate links as one clump of data. This is also reflected by the fact that the client could deny incoming data as soon as it was to be sent, reflecting that some handshaking was going on "behind the scene" between the nodes. This is not implemented in the new network either because it does not make sense on a packet level of the network, but if you consider the data to be more abstract, larger chunks of data it seems more reasonable. Again, this could be changed by making the first router send these messages before trying to route them. Since the clients themselves do not know the PIDs of the recipients of their messages this approach would require some modifications to the client code.

4.5 Known problems / bugs

- The package sizes in the old network is assumed to contrived of many smaller tcp/ip packets (presumably), so the fact that links only perform one transfer at a time is something which will probably need to be adressed in some way sooner or later. "Packages" in the old implementation do not really translate that well to the packets being sent by the new network (and would be rather hard to do well) so some rework should be seriously considered. Simply changing the links to allow them to send multiple transfers at once is a fix which should work well but problems may still arise, especially with regards to the calculating of speed and since the package accepting and dropping is only done when the package is at the last router.

- A bug that is rather strange is found in a function called *delete_tr_out*/3 in the module *cpeerhandler*. The function itself is not documented but the code tries to delete/abort a transfer to a node which the client no longer wants to be performed (possibly because one of the nodes has become a seeder). What happened was that the client would attempt to delete a transfer which was no longer present in the datastructures that it was to be deleted from. This is probably due to the fact that the message had been delivered before then, since this and package failures are the two reasons this would happen and package failure does not occur for any reason other than the recipient dropping it. The code was written in such a manner that this caused the entire simulation to crash. This, along with the fact that the bug does not appear until the 'end-game'phase of the simulation, it was very hard to debug. The code was changed so that the client simply ignores the transfer not existing and goes on with business, but print-outs from the running of the clients indicate that this bug grows over and ends up affecting a lot of the clients to different degrees. It's also unclear whether or not the clients actually recover from this and continue to send data (more testing is clearly needed here) and whether or not this affects the behaviour of the client. In any case the bug is quite strange and only appears to occur pseudo-randomly.

- Speed updates do not exist in the network. This is not very strange since a network (nor a router) keeps track of the traffic speeds of specific clients, but it's still something that the clients themselves expect the network to do for them. A fix for this would be to edit the responsible interface at the package handling level (*pkgcom*). Something which would have to be accounted for is the fact that clients do not, at the moment, have any way to choose at what time to check for speeds. Although a timer started when a packet is sent and compared to when the *package_success* message is received would do the job well it is unclear how this sporadic data regarding speed would work with regards to the way user algorithms are implemented, since the network would calculate all the speeds of transfers in bursts. The refresh rate on these speeds would be rather limited as well since they would occur very infrequently once transfers are started.
- The generating of handles for packets sent over the network is something which could be expanded to be much more useful than it is right now. As it is, handles are only used for book keeping purposes in the client. Potentially, they could be used as a form of time measurement, for example via the vector time stamps or lamport time stamps schemes.

5 Analysis / Discussion

The implementation of the new network side of this simulation has gone well, and although testing needs to be done it should work very well in a simulated environment of many nodes. The network should not have the problems that its predeccesor had with “dropped” frames and lost calculations because there are not any frames, all the routers and links run in parallel, in real time. Since processes and message passing is so light weight in Erlang, the simulation should run rather well. However, the speed of the simulation is rather unclear, not only because extensive testing was not possible to fit into this thesis but also because comparisons with the old network is nigh impossible.

Although for the most part the work of the thesis went very well, some aspects of the implementation of the new network is not very well suited considering the client side.

Specifically, the sending of packages as a singular batch of data is something which the links do not handle too well, since only one packet can be sent at one time through any given link. This is the result mainly of different levels of abstraction in the implementations, but it should be possible to remedy by altering the links to work in a more complex, high-level manner. The problem is essentially that the old transfers that were requested by clients were implicitly including many things which would be handled by seperately sending packets back and forth. The idea that a client can drop packages before they are sent to them are a rather clear indication of this, but there really is no reason why this traffic should not be sent over the network as well. With the previous implementation there might have been a reasonable rationale to this (not bogging down the network process) but it still creates a sort of meta-channel through which clients can communicate which should really be handled by the network as well in order to simulate communication latency. The fact that clients were expected to drop packets as soon as they were sent into the network seemed a bit superflous and unrealistic at the time of implementation and was basically removed, since the routers actually send the packets the clients are deciding on in advance of the request to receive the packet. This would probably be possible to reimplement in the network as it works now, most probably being placed in the router module. The reason that this approach was taken is because data transfers were understood to be abstracted differently between the two theses.

One aspect which proved to be time-consuming to handle was the fact that the client side code does a lot of things which are not meant to be part of the network simulation proper. These meta-calls is right now forced to be handled by the network, but without the delaying by the link bandwidth (verktyg?). Ideally, the configuration files and setup time of the network (before the actual simulation starts) would take care of all these things, but a lot of intelligence (as well as data) would have to be placed into the module taking care of this.

Overall, some things in the network implementation were solved in an ad-hoc manner, and although it would have taken quite a bit of time, a solid overview, design and planning of the entire network module would probably have helped with this.

It is also unfortunate that there was not enough time to perform any real testing of the bittorrent protocol, nor to implement any real way to interface with the network in order to extract statistics. However, this should not be too hard to add because of the implementations structure. However, care must be taken to not add too much strain to the network during the fetching of data since it would slow down the simulation unrealistically. Slowing down one process will not cause the other nodes in the network to wait for it and would produce unrealistic lag. One way to mitigate this would be implementing some sort of global clock that all nodes rely on to measure time. Any maintenance or data-collection from the nodes could be performed 'in between' the ticks of the clock and no node in the network would be able to tell the difference. It would also allow the simulation to run as fast as possible, rather than just (optimally) in real-time.

6 Future Work

Something which would be a good thing to add in any future expansion of the network would be some book-keeping stats and interfacing to the *link*- and *router* modules so that an outside module could get some data showing the flow of packets and present it. This is not something that was prioritised in the thesis work, since focus was kept to making the network work at all. The data that the simulation calculates is not much good if it's not presentable however, so some interface seems like a natural progression now that the simulation (seemingly) works well. Altering the *sent_packet*-functions in these modules to add functionality and to add more book keeping functionality seems like a natural step.

The sleep function in the Link module could quite simply be altered to achieve a more complex (or intelligent) behaviour of the network. One possibility is that the link should be able to handle several messages at once, since the way it works currrently is more correct for a ”lower” simulation of the network than what the current clients expect. One possible problem with this is that the waiting time needs to be altered as packets enter and leave the link, which could be quite tricky, considering for example the overhead that heavy calculations would add, possibly impeding the performance of the link.

Another crucial addition is the addition of some sort of speed mechanism. In the old thesis, clients relied on the network to perform a heavy calculation of all transfer speeds and sending these to the nodes, bogging down the system if the simulation was large enough. With this new distributed network not only does it become hard to calculate these things (since you do not have an overview of the network in one router) but also rather illogical. A good modification would be to add some measuring tool to the client to calculate the speed at which it is sending data to another client. This could be achieved via message sending “above” the network, but something that is done entirely client-side would probably be preferrable.

Adding congestion control for the network, or making the network more “clever” overall is also something that would be very useful. Right now there is nothing in the router implementations that is very clever or similar to the real algorithms that routers use on the internet. This should be relatively easy to add by simply altering the *route* functions, although the algorithms themselves may be rather complex.

Another possible extension is rewriting the client side to be more modular in design. Modifying the client modules right now is quite ardous and perhaps a complete rewrite may well be preferable to maintaning and altering the code now in place.

A crucial addition is some sort of user interface. The simulation needs some GUI to be useful, and there is no real way to extract data from the simulation that is going on. The Google charts work but are also rather limited, a more powerful tool to generate statistics and making the clients and routers output data in a clever form would make the simulation a lot more useful.

In general, this thesis has not gone to too many lengths to actually guarantee the correctness of the clients behaviour (much because of the structure of the clients themselves), so a thorough debugging of the clients would probably be a good idea, though again it may well be easier to rewrite the clients from scratch.

References

[1] *http://www.erlang.org*

[2] Roland Hybelius, David Viklund, *A Bittorrent simulator in Erlang*, Department of Information Technology, Uppsala University, February 2010

[3] Arnaud Legout, G. Urvoy-Keller, P. Michiardi *Rarest First and Choke Algorithms Are Enough*,P. 2006. Rarest first and choke algorithms are enough. In Proceedings of the 6th ACM SIGCOMM Conference on internet Measurement

[4] Bram Cohen, *The BitTorrent Protocol Specification*, fetched from http://bittorrent.org/beps/bep_0003.html on Oct 7 2010.

[5] Bram Cohen, *Incentives build Robustness in BitTorrent*, May 22, 2003