Tribler Download Core Improvement

BSc Thesis IN3700

Rick van Hattem (1297295)
R.D.T.vanHattem@student.tudelft.nl

Raynor Vliegendhart (1174827)
R.Vliegendhart@student.tudelft.nl

Delft University of Technology

October 20, 2007

Committee

Dr. Ir. J.A. Pouwelse
Ir. B.R. Sodoyer
Delft University of Technology

Preface

Computer Science students of the Technical University of Delft have to undertake a – preferably external – project as part of their Bachelor program. These projects are done in groups.

This thesis is the result of the BSc project of Rick van Hattem and Raynor Vliegendhart. The project took place at the Technical University of Delft from early May to the end of September 2007.

We want to thank Johan Pouwelse for giving us the opportunity to participate in the Tribler research project and supervising us. Meetings during the project allowed us to exchange ideas and come up with new things to investigate. We also want to thank both Johan Pouwelse and Bernard Sodoyer for their comments on the draft version of this thesis.

Rick van Hattem
Raynor Vliegendhart

Delft
October 16, 2007

Summary

BitTorrent is a popular peer-to-peer file sharing protocol. From a user's perspective, it is important that a BitTorrent client downloads the files as fast as possible. The socially enhanced BitTorrent client called Tribler – from the Delft University of Technology and the Vrije Universiteit – was presumed to be lacking in speed. This presumption started this thesis' research project. The research focused on two questions. How poor is Tribler's download performance and what determines the performance of a BitTorrent client in general?

To understand the problem, we need to be aware that the limited available upload capacity of the peers in the swarm is a speed bottleneck and that one own's upload capacity is a valuable, but limited currency that can be exchanged for speed from others.

However, it is not known which behaviour is best. In order to solve this mystery, we have to compare different behaviours through measurement. Important aspects to measure include:

- Peer discovery and try out, measured by connection attempts and optimistic unchokes.

- Implementation of the tit-for-tat algorihtm, measured by chokes and unchokes.

To measure and analyse these aspects, two approaches were taken. Our first design involved modifying existing BitTorrent clients to log internal events, but its applicability was limited to open source clients. As a result, we came with a design that analysed network traffic and was therefore applicable to all clients. However, a wrongly taken design decision limited its analysis power.

The knowledge gained from our analysis is that the initial phase and the end phase of a download are important. For the initial phase, a good discovery of peers required. For the end phase, a different piece selection algorithm is required.

There are also some questions left that need to be answered and work to be done. Questions left are:

- Do different peer discovery algorithms benefit in different situations?
- Does the Mainline BitTorrent client benefit from its new DNA service or is it malware?

Future work involves improving our measuring tool and answering these questions.

Contents

Chapter 1

Introduction

Nowadays, high speed internet connections are common[22] and users are sharing content to their heart's content (Figure 1.1). Music, videos and other files are distributed through various means on the internet. BitTorrent, a *peer-to-peer file sharing protocol* (P2P), is one popular way to do so.

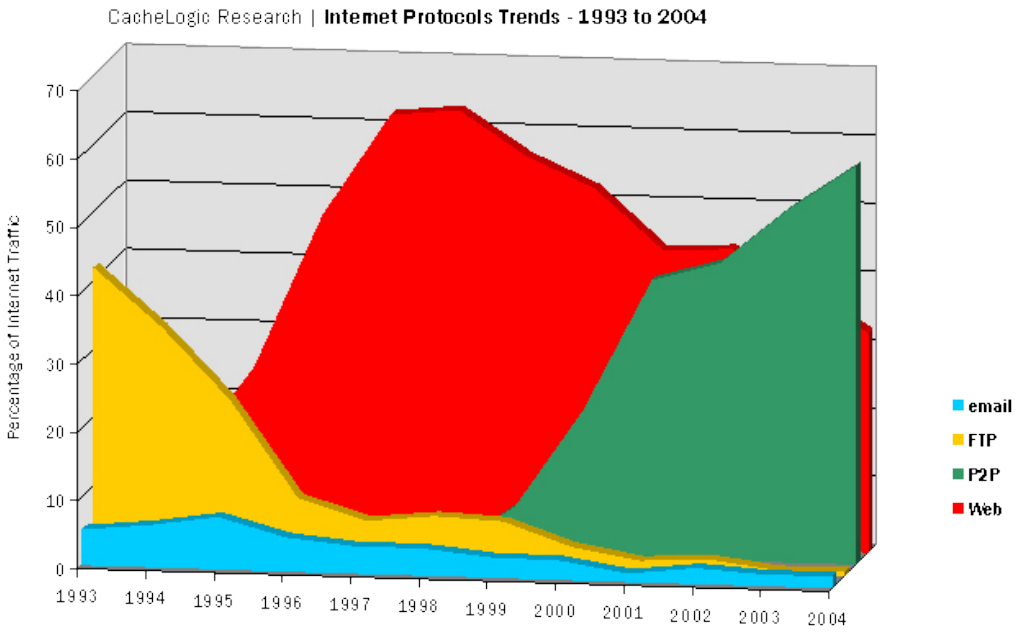


Figure 1.1: Internet trends show an increasing popularity of peer-to-peer systems.

Tribler is a socially enhanced file sharing application created by researchers at the Delft University of Technology and the Vrije Universiteit (Figure 1.2). It is built on top of the existing BitTorrent file sharing network. By adding social aspects to file sharing, users can improve their overall experience by moderating available content and stimulate each other to be cooperative [15].

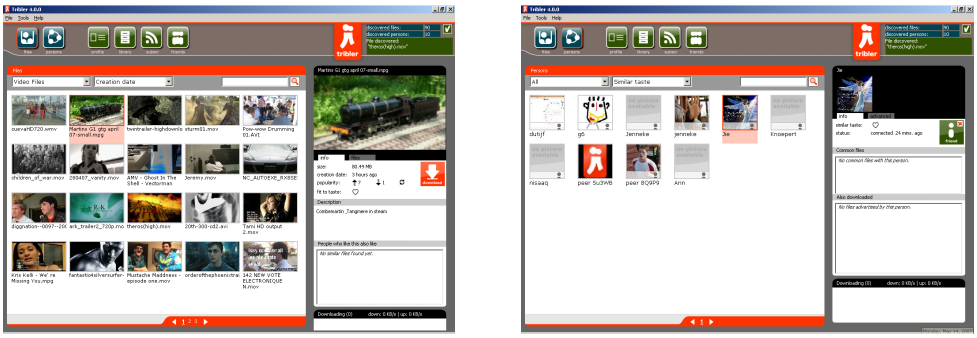


Figure 1.2: In Tribler, users users can share files using the BitTorrent protocol and download cooperatively with friends.

Unfortunately, the researchers of Tribler expressed their concern of their client's download performance. Even if a client has all kind of neat features, a user will not use it if it performs poorly. This concern motivated the research to Tribler's download performance. In this thesis, we will answer the two main questions of this research.
The first question is, is the performance of the Tribler client really as bad as it is thought to be? We will answer this question by comparing its performance with the performance of other clients.
Strongly linked with the first question is the second question. What determines the download performance of a BitTorrent client in general? This question will also be answered through comparison. By performing tests on different BitTorrent clients, we will analyse the differences in their behaviour. For this analysis, we have written a software tool.

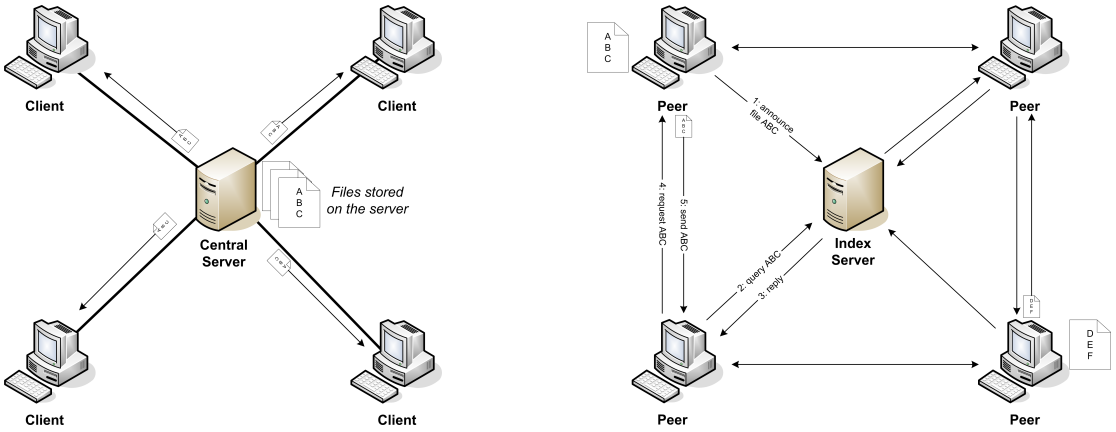
The remainder of this introductory chapter contains a brief description of early (file sharing) peer-to-peer systems in section 1.1. In this section, we explain how these systems worked and show their advantages and disadvantages. We also explain how the newer peer-to-peer system BitTorrent works and its main advantage over the earlier peer-to-peer networks in section 1.2. Finally, we conclude the chapter with the outline of this thesis in section 1.3.

1.1 Early peer-to-peer systems

In traditional computer architectures, files are stored centrally on a server (Figure 1.3a). The central server provides services to its clients. A client can connect and request a file it is interested in. The server will then send the client the file he requested. This architecture is called the *client-server* architecture.

While this architecture is fairly simple, it has its disadvantages. The main disadvantage is the *cost of scalability*. The client-server architecture can scale fairly

well,[1] but the owner or maintainer of the server has to incur the costs. Another disadvantage is the *single point of failure*. If the central server will stop functioning or is taken down, clients can no longer make use of its services.



(a) A client-server architecture.

(b) A hybrid peer-to-peer system.

Figure 1.3: In a traditional client-server architecture (a), files are stored centrally on a server. In a hybrid peer-to-peer system (b), files are stored on the peers. An index server keeps track of the location of files in the network.

In a *peer-to-peer* architecture, files are stored and distributed amongst the computers that are part of the network. All computers in this network are equal and can act both as a client requesting services and as a server providing services. Hence, they are called *peers*.

Hybrid peer-to-peer systems

Unlike the client-server architecture, a peer-to-peer system is a bit more complex. Peers in a network are faced with the problem of locating files they want. One solution is to use a *hybrid* peer-to-peer architecture (Figure 1.3b). In this architecture, there is an index server fulfilling the role of a directory. This server keeps track of available files in the network and where they are stored at. When a peer wants to download a file, it asks the server if anyone has it. The server responds and tells the peer where it can find the file. The peer then establishes a connection with the peer that has the file and the file can be transferred. A popular, but now defunct network that used this architecture was Napster [1].

The advantage of this architecture over the traditional client-server architecture is that the maintainer of the index server has lower costs. The server does not need

[1]For example, scalability can be obtained through caching, server replication and content delivery networks [16].

to provide the actual files and thus requires less bandwidth. Files are replicated amongst the peers. Popular files are higher in demand and get replicated faster, increasing their availability. Thus, scalability is achieved automatically.

The problem of the single point of failure still remains, but it is less severe. When the index server is out of order, peers cannot locate any files in the network. However, this does not disrupt any file transfers that are already being done.

Decentralized peer-to-peer systems

To eliminate the single point of failure problem, one could eliminate the existence of index servers. The resulting architecture would be a decentralized peer-to-peer network (Figure 1.4). Now, the responsibility of locating files is placed on the peers. When a peer wants to download a file, it sends out a query to all the other nodes[2] it is connected to. These nodes forward the request to all nodes they are connected and the process repeats itself until the request reaches a node that has the file. This node will then reply and the response travels the same path back till it reaches the requesting peer. The requesting peer can then establish a connection with the peer that has the file and the file transfer can begin [1].

This level of indirection does make look-ups slower as the query may have to travel through a lot of nodes until it reaches the node it has the file. The look-up can be sped up, though, by having each peer cache responses from other nodes. Each peer is then basically an index server.

In this new architecture, the peer is also responsible of finding a peer of the existing network to connect to. In the hybrid architecture, peers have to connect to a central server to be part of the network, but there is no central point of access in a decentralized network. To join a decentralized network, a new peer must have an initial list of a few existing nodes in the network. In this so called *bootstrapping* phase, the peer tries to connect to a few peers in its list. Once connected and part of the network, the peer can find new peers through the network and update its list.

The problem that remains is that a new peer has to acquire a nodes list the first time it wants to connect to the network. To solve this, peer-to-peer applications usually download a peer list from a webserver.

Leeching and performance

There are a few disadvantages of the two described peer-to-peer systems. One is that in early networks, participants are not incented to share. This gives birth to

[2]Peers and nodes are used interchangedly throughout the remainder of the text.

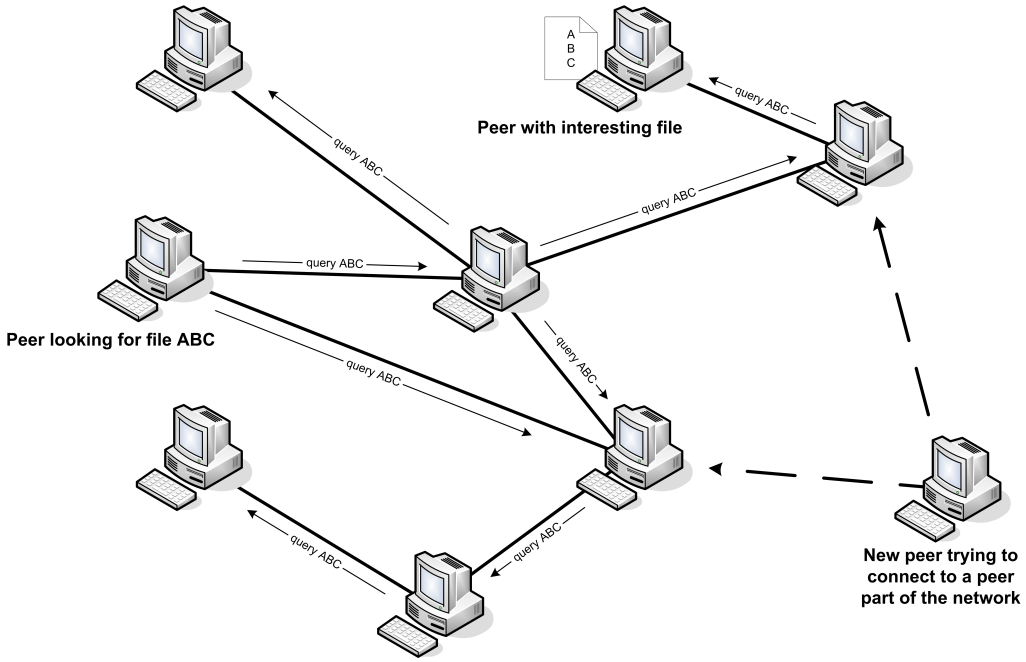


Figure 1.4: In a decentralized P2P system, files are discovered by flooding queries through the network. New peers join the network by trying to connect to existing peers.

leechers, who consume resources of the network, but do not make their own resources available. While this was a non-existing problem in a client-server architecture, it is in peer-to-peer systems.

Another disadvantage is that the download speed is heavily depending on the peer you are downloading from. Even worse, in early file sharing applications, you are often restricted from downloading a file from a single peer. Compared to the traditional client-server architecture, the central server usually has dedicated fast lines in contrast to peers who are often using normal consumer connections.

1.2 BitTorrent, the incentive to share

Bram Cohen released his first version of BitTorrent in 2001[10]. It was developed to create a distribution mechanism in which the cost of upload is shared by the downloaders[11]. This is achieved by allowing downloaders to trade smaller pieces of the file with each other. The following sections describe BitTorrent and a brief graphical overview is depicted in Figure 1.5.

Contributing files

Distribution in BitTorrent works as follows. To offer a file for distribution, the contributor first creates a *`.torrent`* file. This file contains *metadata* about the file to be offered. It contains both the *hash* of the whole file to be offered and the *hashes* of the *smaller pieces*. Evidently, it also contains the size of a single piece and the amount of pieces the whole is divided into. The `.torrent` file also contains the address of a so called *tracker*. After the metadata is created, the contributor registers the `.torrent` file at the tracker and connects to it.

Tracking the swarm

A BitTorrent tracker is a server that keeps track of the *peers* in the *swarm*, their completion rate and other information. The peers regularly keep in touch with the tracker. A swarm is the total group of downloaders and *seeders*. A seed is a peer that has the full file.

Joining the swarm

When other peers want to download the file that is being offered, they have to download the *.torrent* file and connect to the tracker specified in the metadata. Then, the peers receive a list of peers and seeds that recently have connected to the tracker. They can then try and connect to the peers and seeds listed in the tracker response. This is how new peers can connect to the swarm.

Downloading files and fair sharing

When a peer is part of a swarm, it can receive pieces from seeds and from other peers. Pieces are exchanged between peers on a *tit-for-tat* basis. This means that if peers are willing to share pieces they have with other peers, they can receive other pieces from those other peers and achieve higher download speeds and thus they are incented to do so.

The tit-for-tat algorithm is simple and effective. Each peer is connected to several other peers. Peers can request pieces from each other, but these requests are not always honored. Each peer keeps track of the amount of pieces it received from others. This information is then used as a selection criterion. Requests from peers who have been uploading the most are more likely to be honored than requests from leechers.

Improving the architecture

The original BitTorrent protocol relies on a central server to track the swarm. Solutions for this single point of failure have been developed over the years. Nowadays,

BitTorrent has support for multiple trackers in `.torrent` files and it supports decentralized tracking.

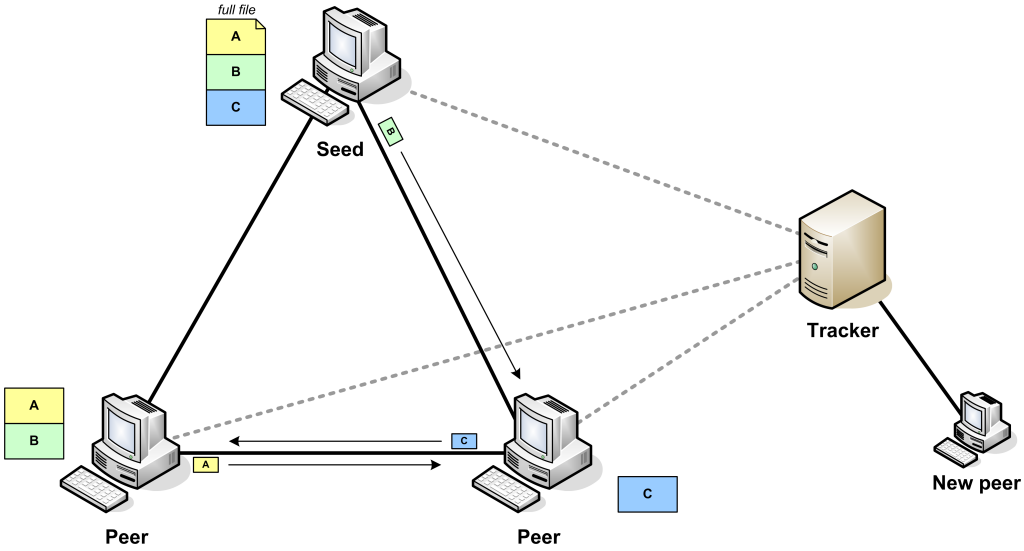


Figure 1.5: In a BitTorrent swarm, one or more seeds are distributing file pieces to other peers. Amongst peers, missing pieces are traded. New peers join the swarm by contacting a tracker, which has a list of peers in the swarm.

1.3 Thesis outline

The remainder of the thesis will focus on the two questions posed earlier in this chapter. How poor is Tribler's download performance and what determines the download performance of a BitTorrent client in general? In order to answer these questions, we will first need to get a better understanding of the problem. Chapter 2 analyses this problem. In chapter 3 we present the evolution of the design of our measurement testbed that is required to find the answers to our questions. Implementing the final design was not without trouble. The problems we encountered are discussed in chapter 4. The insights we acquired through measurements are discussed in chapter 5 and an overall reflection on how the project went is given in chapter 6. In chapter 7 we present our conclusions and we show that there is still work to do.
In appendix A the requirements and design document can be found. Our test plan is appended as appendix B. How to deploy our created software tool can be found in appendix C and how to use it can be found in appendix D.

Chapter 2

Problem Analysis

From a user's perspective, the prime goal of peer-to-peer software is to download files as fast as possible. If a client fails to perform well, the user will use a different client or it will use alternative means to obtain the files he is interested in. In order to make Tribler a faster BitTorrent client, we need to have a greater understanding of this problem. In section 2.1 we discuss the bottleneck in attaining a high download performance and in section 2.2 we show that this bottleneck has consequences for the behaviour of BitTorrent clients. In the final section of this chapter (2.3), we discuss that measuring behaviour of these clients is necessary to determine which behaviour is best.

2.1 Speed bottleneck

When a client is performing a download transfer, it is limited by resources. For one, the peer itself has only a limited amount of bandwidth. It cannot download and upload faster than the maximum download and upload bandwidth. This means the client must perform as best as it can under these conditions. In the eyes of the user, performing best means fully using the available download bandwidth. In order words, the download channel of the user's connection should be saturated.

But how does the user's client achieve this? The following example will illustrate. A BitTorrent swarm consists of a diversity of peers. Assume there are three peers named A, B and C in a certain BitTorrent swarm (Figure 2.1). Peer A has an maximum upload capacity of 50 KB/s, peer B has a capacity of 20 KB/s and peer C has a capacity of 80 KB/s. The user's connection is limited to a maximum download speed of 100 KB/s. It is obvious that the bottom peer needs at least to be connected to two of the three other peers in order to be able saturate its connection. However, if the bottom peer is connected to only peer A and B, it will not be able to download at the maximum obtainable speed at all.

From the previous example, it should be clear that *the available upload capacity in a*

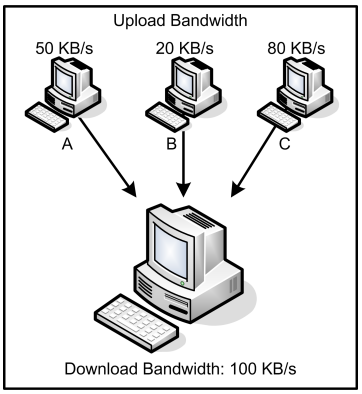


Figure 2.1: The bottom peer can saturate his download connection in several ways.

swarm is *a determining factor* in the maximum speed a client can obtain in theory. In practice, a client might not always achieve that speed. It is either limited by its own download bandwidth or *by the peers it sees*.

2.2 Fighting for scarce upload bandwidth

In the previous section it became clear that it is important for the client to connect to enough other peers in order to be able to saturate its download connection. However, just connecting to the right amount of peers does not guarantee high speed downloads.

Let us look at the following example. In Figure 2.2, the two bottom peers have a connection with the top peer. The top peer can upload at a maximum of 100 KB/s and the two bottom peers each can download at the same speed. For simplicity's sake, there are no other connections, so the bottom two peers cannoth exchange file pieces with each other. In this situation, each of the bottom peer would like to download at 100 KB/s, but the available bandwidth is scarce and the top peer has to divide its upload over the two bottom peers. How does the top peer determine how to distribute its upload bandwidth? Well, the top peer judges the other two peers by their upload behaviour. When a bottom peer is sharing interesting file pieces with the top peer, the top peer will like him and is more likely to *reciprocate* by sending interesting file pieces back.

So, the upload behaviour of a peer is important to get scarce upload bandwidth from other peers. However, the upload bandwidth of oneself is also scarce. This essentially means that upload bandwidth is a *rare currency* that needs to be cleverly spent. Each foreign peer might be running a different client and each might have slightly different reciprocation rules. A good performing client must thus be able to figure out which peer has the best exchange rate, so it can get the most bang for its upload.

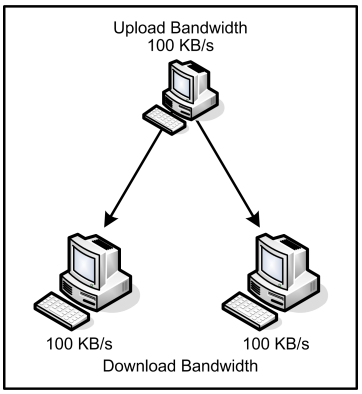


Figure 2.2: Two peers fighting for scarce upload bandwidth.

2.3 Measuring behaviour

As shown in the previous two sections, BitTorrent clients need to find and connect to peers with a reasonable amount of bandwidth and then test those peers for reciprocation. However, it is currently not known which *peer discovery* strategy and which *peer try out* strategy is best. In order to determine this, we have to test different strategies and evaluate them. Evaluation is based on their impact on the download performance. So how do we measure this?

We can measure the impact on the download performance by simply starting a BitTorrent download and measure how much time it takes for a certain client to complete the download. While this is certainly a valid way of measuring, it would not give us any information how exactly the download performance is affected, but only that it is affected in a positive or negative way.

Therefore, it is a better idea to measure the progress over time. From the progress curve's incline, we can derive the download speed at any point in time. From this, we can also see the maximum speed a client obtained in a test and how long it took to obtain that speed.

However, just having a picture of the progress made over time is still lacking information. We can only observe changes in the download speed, but not explain them. If we want to explain changes in speed, we will have to be able to link them to other changes. For example, the client discovered a few new peers recently and tried them out. This is why it helps to have information of the following:

- *Discovered peers.* Knowing more peers means a client can more likely saturate its connection, as discussed in the first section.
- *Chokes and chokes.* Regular chokes and unchokes in BitTorrent are used to tell the other party if the client is willing to send data. Only unchoked foreign peers will receive file pieces.

- *Optimistic unchokes.* An optimistic unchoke is a mechanism in BitTorrent to try out a new peer. Normally, a client prefers sharing pieces with other peers who have been nice in the past, but every now and then a new peer gets unchoked.

If we can measure this information, we can analyse the differences between existing clients and we can evaluate them. To measure this, we have developed a measurement testbed. This design is discussed in the next chapter.

Chapter 3

Measurement testbed design and evolution

This chapter describes our process of designing the measurement testbed for BitTorrent applications. Initially, the project's goal was to analyse the performance of Tribler compared to other clients and, based on that analysis, improve Tribler's download core (section 3.1. However, during the project the task of actually implementing a better download core was scrapped. Instead, the project shifted its focus on analysis only (section 3.2). As a consequence of this change, we started to redesign our way of measuring. The final design of our measurement testbed –coined BitSMART– is discussed in section 3.3 and its design flaw is discussed in section 3.4. Section 3.5 gives a comparison between our initial and our final design.

3.1 Initial design, internal logging

At the start of the project, we had meetings with our supervisor, J.A. Pouwelse. In these meetings, our approach to the problem started to take shape. To compare Tribler's download performance with other BitTorrent clients, our supervisor suggested the following list of open source clients:

- Azureus[3].
- BitTorrent, the official client also referred to as Mainline[6].
- Boudewijn's fork of Tribler[17].
- libtorrent[13].

We decided to measure the performance by tracing the internal state changes of each client. This was done by inserting our own code into the clients. After modifying a client, tests could be performed with that client. By logging each change in progress and logging every discovered peer, choke and unchoke to a file, we were able to parse

these logs later on and generate graphs from it. This design is depicted in Figure 3.1.

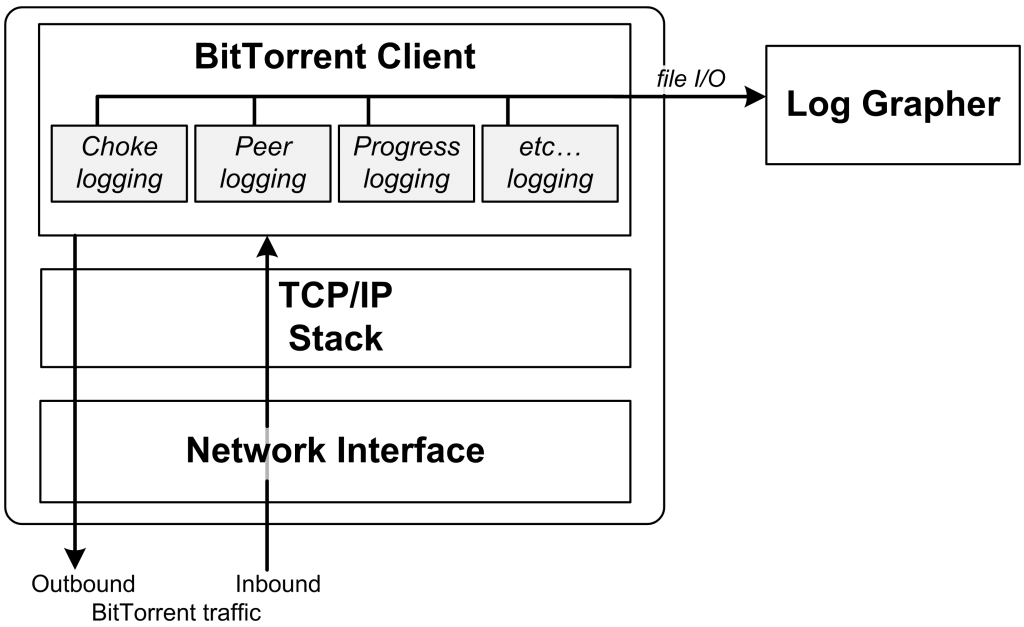


Figure 3.1: The initial design of our measurement testbed.

Unfortunately, we were not able to make use of libtorrent's own logging features and we were not able to get Boudewijn's client to work. The logs of libtorrent were large and difficult to parse. The problems of Boudewijn's client were related to the fact that it was not a stable release. In the interest of time, we decided to ignore these two clients and focus on the analysis of the remaining clients.

From the analysis phase, we came to understand the importance of several performance factors[1] and we were ready to design a new download core for Tribler.

3.2 Project's turning point

Shortly after presenting our acquired insights, our supervisor suggested a change in the project. Instead of implementing an improved download core for Tribler, we would have to focus on analysis only and leave the actual download core implementation to the Tribler team.

The reasoning behind this suggestions was that our analysis only covered a few clients. This was caused by our measuring method that was only applicable to open source clients. So our new task was to come up with a new design that allowed us

[1]The acquired insights from the measurements can be found in Chapter 5.

to analyse more clients. Being able to analyse more clients would certainly mean we could give a better evaluation of Tribler's performance.

While we were pondering the suggestion, we came across a tool called Wireshark[23]. After trying it out, we thought it could be pretty useful. It could do network packet capturing and it could recognize BitTorrent packets. With this information, we accepted the suggestion and started the redesign.

3.3 Final design, the BitSMART tool

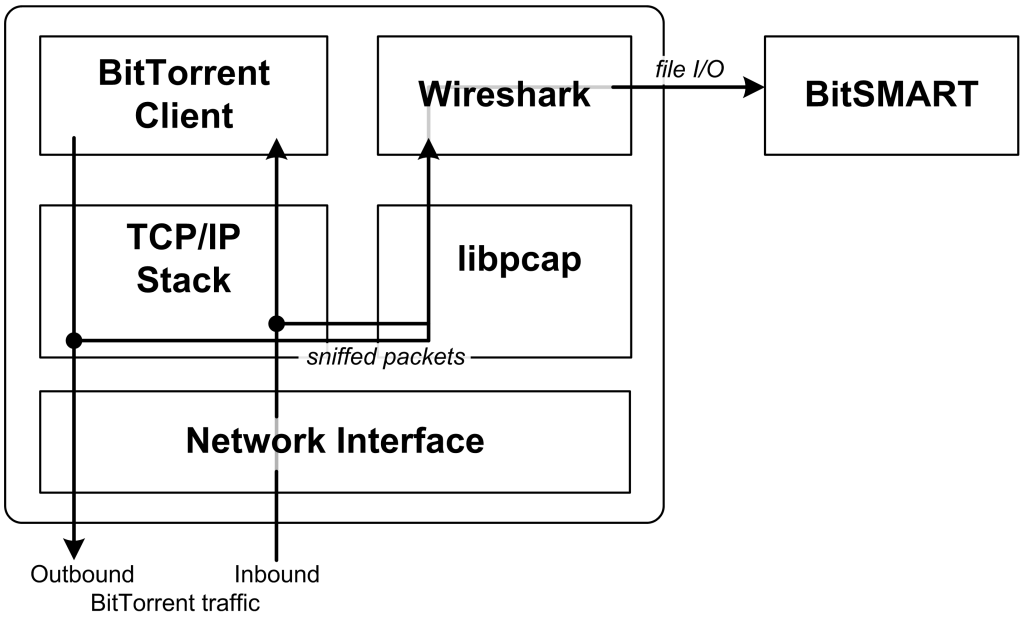


Figure 3.2: The final design of our measurement testbed.

For the redesign, we reformed the measurement testbed, so the measurement took place completely outside the BitTorrent client. With this, we no longer had to modify each client that had to be tested and we were no longer limited to open source clients. The Wireshark component would perform the network packet capturing during the test by using the libpcap library[12]. The captured BitTorrent traffic would then be saved to a file in some format (discussed in the next section).

The next step in the chain would be our new software component, which we named BitSMART.[2] This tool is made out of three parts:

- The front-end component, responsible for parsing the output of Wireshark and performing an analysis on the BitTorrent traffic.

[2]**Bit**Torrent **S**peed **M**easurement and **A**nalysis for **R**esearch **T**ool.

- The back-end component, responsible for generating graphs.
- The main component, responsible for the command line interface and driving the two front-end and back-end components.

Of these three components, the main component would require the least amount of effort to implement, as its role is only to be a mediator between the user and the two other components. The back-end component would also require little effort, as we had already developed a graphing tool for our first design. Most effort would have to be spent in writing the front-end.

3.4 Fatal flaw, the PDML format

For the interface between the Wireshark component and BitSMART (Figure 3.1), we had several options. Wireshark is capable of writing captured network traffic in several file formats:

- pcap format, containing Ethernet frames.
- Plain text, containing short summarized information about protocols.
- PDML (XML format), containing detailed protocol information.

Of these three formats, the plain text format was not found to be suitable as it did not contain detailed information. PDML is easy to read – it is an XML document – and it contains specific BitTorrent information. The pcap format on the other hand contains all network traffic in its raw form, but reading it is not quite as simple. On top of that, choosing this format would mean we have to do a lot of the low-level analysis (like re-assembling TCP packets) and protocol recognition ourselves.

Based on this information we had at the time we had to make a decision, we decided to go with the PDML format. Wireshark would do most of the heavy lifting of analysing the network traffic and presenting it in a suitable format. This would save us a lot of time, so we could spend our time on focusing on analysing the BitTorrent traffic.

Unfortunately, while Wireshark can recognize all kind of protocols, we found out later on that its PDML output is limiting, as it does not always contain full details of TCP packets. This happens when Wireshark was unable to recognize the protocol. A concrete example would be the tracker traffic. While the tracker protocol is just HTTP, Wireshark does not often recognize it. This is the case when the traffic flows over a non-standard HTTP port. But even when it does recognize the traffic, the details of tracker responses, i.e. the full content, are not extractable from the XML file.

And so, the time we thought that we had saved and could have been spent in in-depth BitTorrent analysis became meaningless, as the design decision limited us. By the time we found out, there was not enough time to switch to the pcap format. As a consequence, we decided to continue with the PDML format and create a limited tool, so we could at least gather experience for future work.

3.5 Comparison of the two designs

Ignoring the flawed design decision described in the previous section, there are a few differences between the two designs in their applicability and analysis power. These comparisons are summarized in the following table.

	Logging	Network traffic analysis
Applicability	−−	++
Scalability	−−	++
Development time	+/−	−−
Accuracy	++	+/−

Table 3.1: Comparison between our initial and our final design.

The applicability of our first design is very limited. As this design requires source code modification, it can only be applied to open source BitTorrent clients. As discussed earlier in this chapter, this is the reason why we switched to the network traffic analysis method. This method can be applied to any BitTorrent application.

Our first design also does not scale well. For each new client that has to be tested, new logging code has to be inserted. In order to do this, we first have to identifiy all the locations in the source code where the logging activity should take place. This requires a reasonable understanding of the BitTorrent client's code and it would even require reverse engineering if the documentation is lacking. On the other hand, our BitSMART tool only has to be designed and implemented once and it can be used for any new client without any effort added.

However, implementing logging functionality in a client does cost considerable less time than writing a full fledged network analysis tool. While it still takes some time to understand a client's source code, doing all the low-level network analysis on the other hand requires lots of time. Trying to save time by using existing software might backfire, as discussed in the previous section.

While our first design was limited in its applicability, our final design is limited in its accuracy. This limitation comes from the fact that we are considering the BitTorrent client under test as a black box and only analyse the input and output.

As a consequence, encrypted messages sent between clients can be captured by the network packet capturer, but they can not be deciphered. As a result, we cannot analyse these encrypted messages and this might give skewed results. Our first design does not have this limitation, as it captures all relevant information inside the client where all BitTorrent messages are already deciphered.

When all these advantages and disadvantages of both designs are taken into account, which design is best? Well, it depends on the purpose of measuring a client's download performance. The network traffic analysis approach is best suited for comparing different BitTorrent clients as it can be applied to any client. Implementing a tool that does this analysis would cost a lot of time however and it is less accurate than adding logging code to each client.
Adding logging code is preferable when the measurement only has to be done on a single client. In this case, the developers of a BitTorrent client could add extensive and modular logging features. This will allow them to evaluate the performance of different versions of their client.

Chapter 4

Implementation trouble

In this chapter, we will present the main problems we encountered during development and how we solved them. In section 4.1, we discuss the problems we had with parsing XML. While it might seem to be a trivial task at first sight, it is not in certain situations and with certain requirements. In section 4.2, we discuss the problems we had with converting pcap files to XML files correctly, without losing BitTorrent messages. Finally, we confirm the rule of thumb that one should never trust external input in section 4.3.

4.1 Parsing XML

The task of our final design's front-end component was supposed to be fairly simple. Read the XML output of Wireshark and extract the information we are interested in. Our first naive attempt, however, used a DOM[1] XML parser. A DOM XML parser tries to load the whole XML document into memory as a tree-like datastructure. Normally, this would not be a problem, were it not for the very large files[2] to be parsed. The XML parser simply ran out of memory.

Since storing the whole file in memory was not an option, we had to read the XML file sequentially and only storing pieces of information we really needed in memory. We first looked at the option of using a SAX[3] parser, but given the push nature[4] of SAX and our limited experience with it, we decided not to use it.

The parser we were still looking for had to be able to read the XML file sequentially to be memory efficient. However, as we were debating how to do the XML parsing, we also found out we had another requirement for the parser. We had to be able

[1]Document Object Model.

[2]Raw *pcap* files of 200 MB corresponded with XML files larger than 1 GB.

[3]Simple API for XML, a serial parser.

[4]SAX parsers call your code when a new XML node is encountered, instead of your code calling the parser.

to scan a part of the XML file twice. This seemed necessary for the algorithm to extract the local IP from the XML file.[5] Simply reopening the file was deemed to be unfavorable. Files like STDIN[6] are not seekable and rewindable.

After some pondering we decided we would have to write our own relatively simple XML parser. This parser would have to read files sequentially, but also had to be able to buffer read XML nodes when instructed, so it was possible to rewind to the last buffer starting point. Also, to deal with very large inputs, we added support to the parser to handle input that was spanning several files.

4.2 From pcap to XML, lost in translation

Near the end of the development cycle, we found out that Wireshark showed more captured BitTorrent messages when we loaded a pcap file in its graphical application than when we asked Wireshark's command line tool to convert a pcap file to XML. We were puzzled. How come that the XML output was missing BitTorrent messages?

Shortly after this discovery, we stumbled on something interesting. Opening a pcap file in Wireshark and then filtering on BitTorrent messages had a different behaviour than specifying the same filter in the open dialog box. In the latter case, less BitTorrent messages were recognized and shown than in the former case (Figure 4.1). We also found out that splitting pcap files into several pcap files and then convert them to XML also caused certain message to become lost.

After studying the manual of Wireshark, we found a way to solve the afore mentioned problems. By specifying a *correct* filter that selected only BitTorrent traffic, we were able to convert pcap files to XML seamlessly. Apparently, our initial command flags were incorrenct. Also, the XML output became several times smaller with these flags than our initial approach. As a consequence, the XML file output only spanned a single file, solving the problem of multiple XML files.

4.3 Erroneous piece indices

One of the odd bugs we encountered in our code was related to erroneous input. Our BitSMART tool was working fine for a few test pcap files, so we thought it would work on real measurements. It worked for most measurements, but there were two exceptional cases that crashed our tool. The two cases were measurements from Tribler and μTorrent.

[5]Details of this algorithm can be found in the design document appendix.

[6]STDIN stands for Standard Input, a special file on most common operating systems.

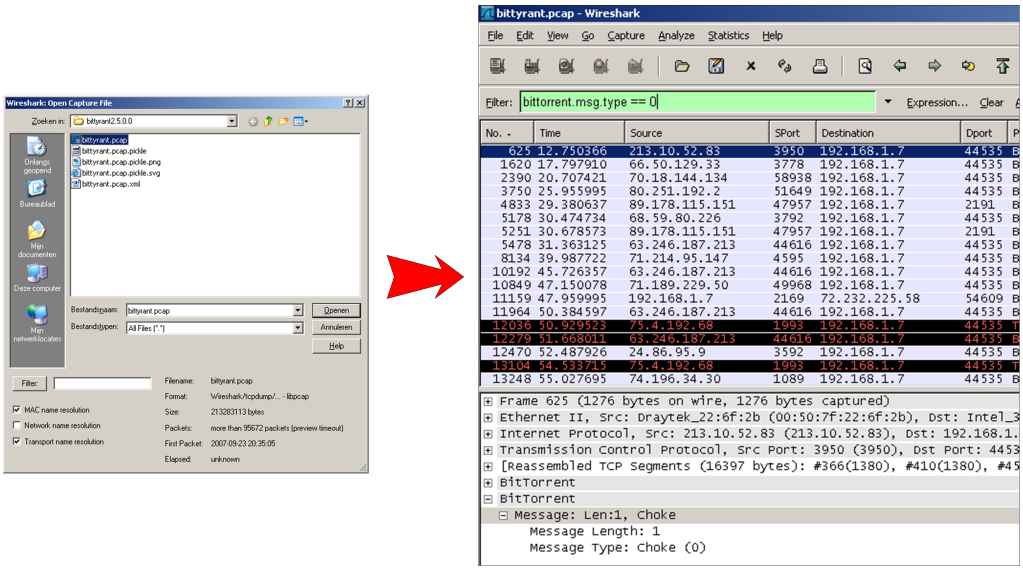


Figure 4.1: Different behaviour in Wireshark when specifying the filter after a file is loaded (a) and when specifying the filter before loading (b).

After debugging, we saw that a few *have messages*[7] in the XML output contained a too large piece index. This caused our code to crash and we fixed it by checking if the index is within bounds (Figure 4.2).

After fixing the bug, we investigated if there was more to this bug. We first suspected that maybe there was a bug in Wireshark's pcap to XML conversion. After comparing the original pcap file and the XML output, this seemed not to be the case. The only other explanations we could came up with were:

- Wireshark's BitTorrent packet dissector[8] contains bugs.
- Certain BitTorrent clients deliberately send erroneous have messages.
- The original pcap files got slightly damaged.

Unfortunately, all three seem unlikely and we are not sure what is causing the erroneous piece indices.

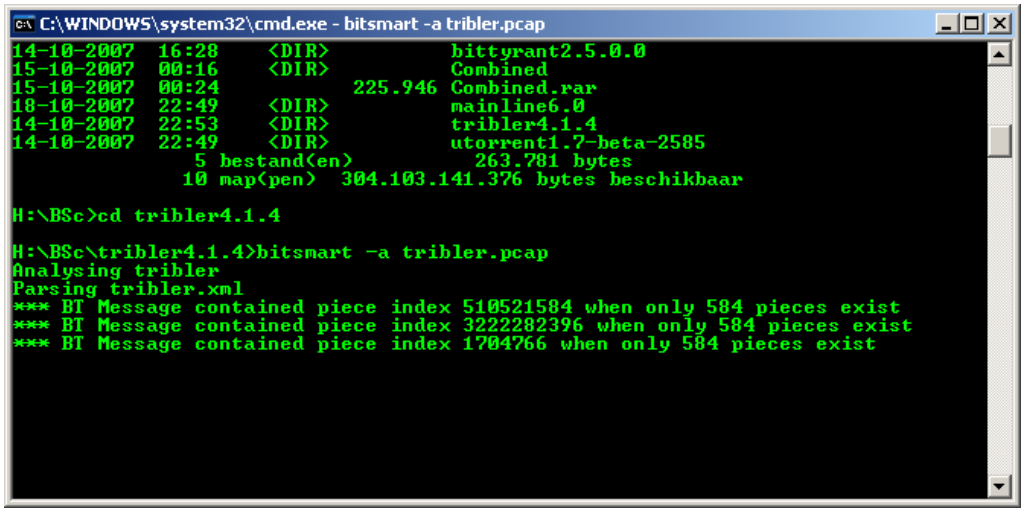


Figure 4.2: BitSMART encountering erroneous piece indices in the XML file.

[7] A have message in the BitTorrent protocol is used to tell other peers that you have a certain piece. It is used to convey one's progress.

[8] A Wireshark packet dissector is a part of Wireshark that recognizes a certain network protocol.

Chapter 5

Measurements and acquired insights

This chapter presents the acquired insights through our measurements. Section 5.1 will focus on the critical download phases of a BitTorrent download. We show that there are two important aspects in reaching high download speeds. We also present interesting and unexpected results of a measurement, leading us to believe that there might not be a single, true strategy for BitTorrent clients in section 5.2. Additionally, we reveal a shocking discovery that came to light during one of our tests in section 5.3. We discovered that Mainline is using a piece of software of which its purpose is unknown.

5.1 Critical download phases

Initially, when the project's goal was to find the areas where Tribler's performance was lacking *and* improve them, we did several measurements using a logging method as described in chapter 3. From these measurements, we quickly identified the two most criticial phases of a BitTorrent download. In the following paragraphs, we will use the data of measurements of Tribler 3.6, Azureus 2.5.0.4[1] and Mainline 5.0.7.

Figure 5.1[2] shows that Tribler needs a bit more time than Azureus to reach maximum speed. For Tribler, this initial phase of the download however does not seem to have much impact on the download performance, as Tribler is only about 15 to 20 seconds behind Azureus.

Instead, the most critical phase is the end phase of the download. The progress graph clearly shows that near the end Tribler's download speed collapses, while Azureus almost retains maximum speed. The inspection and the comparison of the

[1] Azureus was tested in two different configurations, with and without DHT.

[2] While this is a diagram from a specific measurement, other measurements shared the same characteristics.

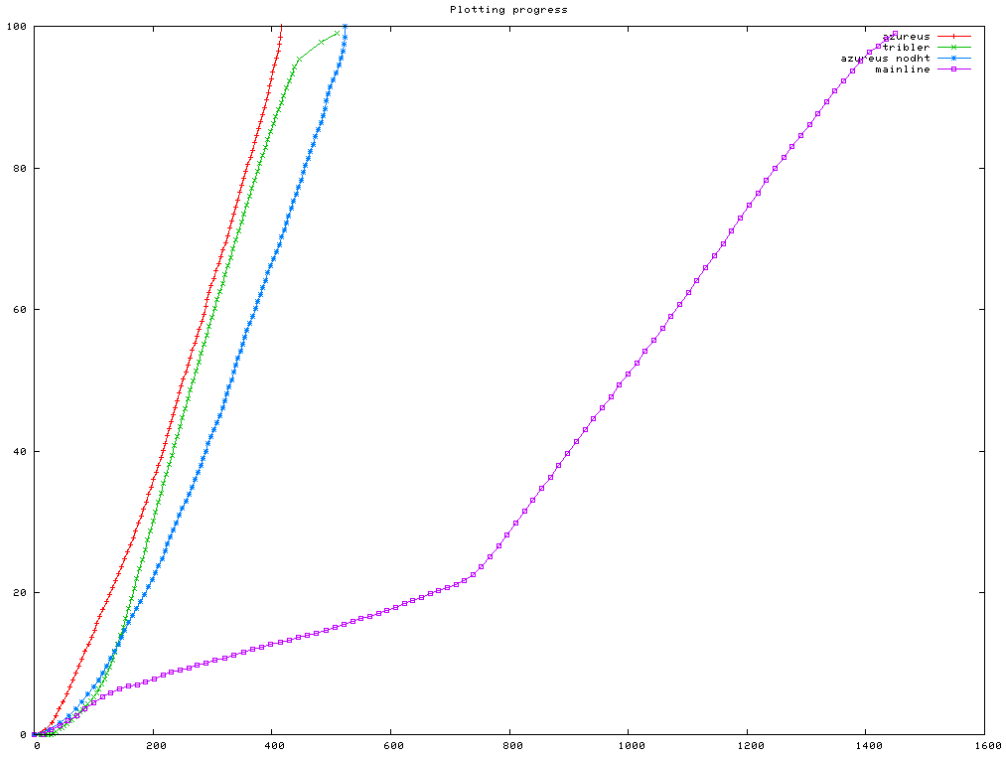


Figure 5.1: Progress in percentage over time. Time unit is in seconds.

two client's source code showed us that both clients behave differently in the end phase of the download, the so called endgame mode. This special mode is described in [11] and it specifies how to request file pieces. Looking at the results, it obviously seems that Azureus' endgame strategy performs better than Tribler's.

While inspecting the source of Azureus, we also discovered that Azureus does optimistic unchokes between choke/unchoke rounds (which take place every ten seconds) when not all upload slots are in use. This means that when an upload slot becomes unused, Azureus does not wait till the next choke/unchoke round to fill the slots. Instead, it will check every second if it can perform any so called *immediate unchokes*. Adding immediate unchokes to Tribler might improve the initial phase of the download.

Focusing on the rather terrible performance of the Mainline client in Figure 5.1, one might wonder why the Mainline client does not even manage to obtain the same highest speed as the other clients. To come up with a possible explanation, we would need more information. Taking the information from Figure 5.2 into account, we can see a very plausible cause. The Mainline client does not simply try hard enough to find more peers. Other clients search more aggressively for new peers. Knowing more peers also means you can try more of them, and thus having a higher chance

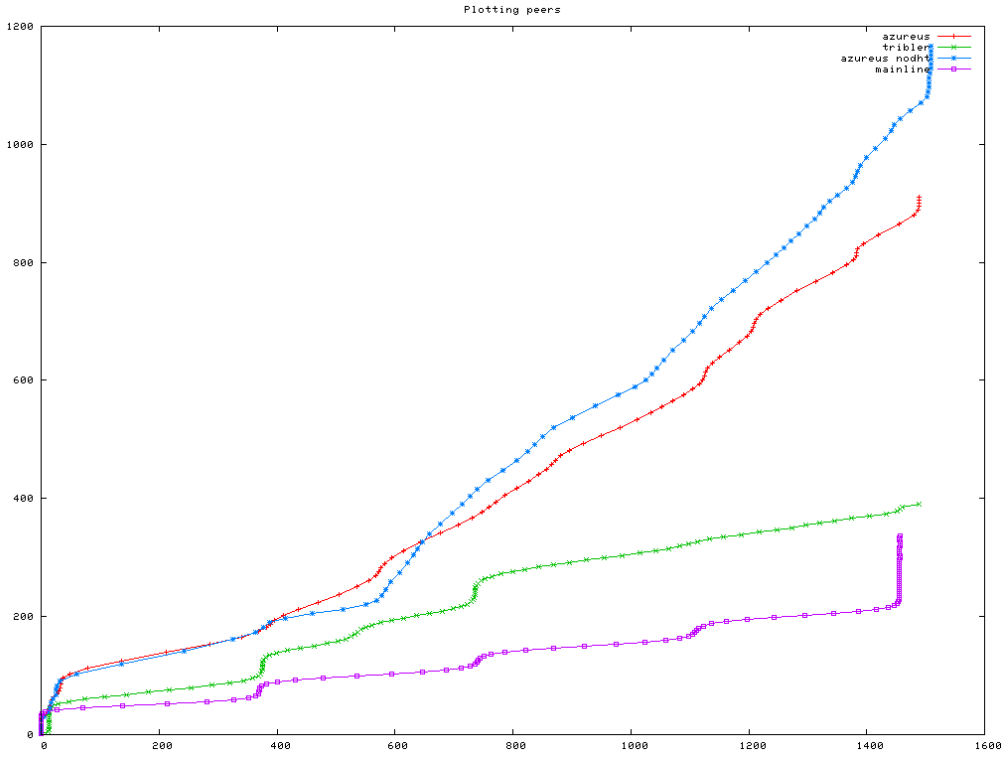


Figure 5.2: Discovered peers over time. Time unit is in seconds.

of finding faster peers.

In other words, to perform well, a BitTorrent client should try to find good peers quickly in the initial phase of the download. The time it costs to reach top speed is time that cannot be won in the middle phase of the download. That means if a BitTorrent client is slow at the start, it will lag behind its faster competitors. Fortunately for Tribler, it is not too bad at getting that maximum speed quickly.

A BitTorrent client should also try to maintain that maximum speed. In a middle of the download, this does not seem to be a problem for the three clients. The difficulty seems to lie in finishing the download. Unfortunately for Tribler, it is not very good at it. The obstacle of getting the last few file pieces during the endgame requires a different piece request algorithm.

5.2 Adaptive strategies

Using our measuring tool, we unfortunately did not quite gain new insights, but remarkably, one of the measurements featured an unexpected performance. In the battle between uTorrent 1.7, BitTyrant 1.1, Azureus 3.0.2.2, Tribler 4.1.4, BitComet

0.89 and Mainline 6.0,[3] the winner was Mainline. Performing rather poorly in older tests,[4] this time it passed the finishing line at a blazing speed. The times are listed in Table 5.1 and the progress is shown graphically in Figure 5.3. Figure 5.4 shows the peer discovery over time. Graphs of chokes and unchokes are left out as they did not show any interesting features.

#	BitTorrent client	Time	Difference	
1	Mainline	07:56	—	
2	μTorrent	09:07	01:11	13%
3	Azureus	09:16	01:20	14%
4	BitTyrant	10:02	02:06	21%
5	BitComet	10:20	02:24	23%
6	Tribler	10:25	02:29	24%

Table 5.1: Download times in a swarm of approximately 7000 seeds and 400 peers.

These times might or might not be significant, but it is highly remarkable that the Mainline client managed to be the fastest with a significant lesser amount of discovered and connected peers. Previously we argumented that knowing too few peers would be disadvantageous for a BitTorrent client, but now it seems to be a good strategy. Is there a possible explanation?

Well, looking at the swarm one might notice the enormous amount of seeds. We assume that the swarm was really healthy and that were relatively a lot of fast seeds. This meant that it would only require a handful of these fast seeders to saturate an ADSL connection. In this case, Mainline's strategy would be extremely lucky. While others are trying out a lot of peers and most likely suffering a bit from unstable or low download speeds, the Mainline client takes it easy and enjoys the high speed transfers of the few peers it found. The progress graph shows clearly that the Mainline client needs a very short time to obtain maximum speed.

So our conclusion would be that a client should have several levels of aggressiveness in discovering peers, based on the current situation. In order to be able to judge the current situation, however, would require some awareness about the capacity of the network connection. Still, having this information means the client will not waste time trying to find even better peers aggressively when it is not needed. On the other hand, not having this information and thus capable of only using one strategy means that the client will only perform optimal in environments that are favorable to its strategy.

[3]BitThief was also participating, but consumed too much CPU power, rendering the measurement data invalid.

[4]See Figure 5.1 in the previous section.

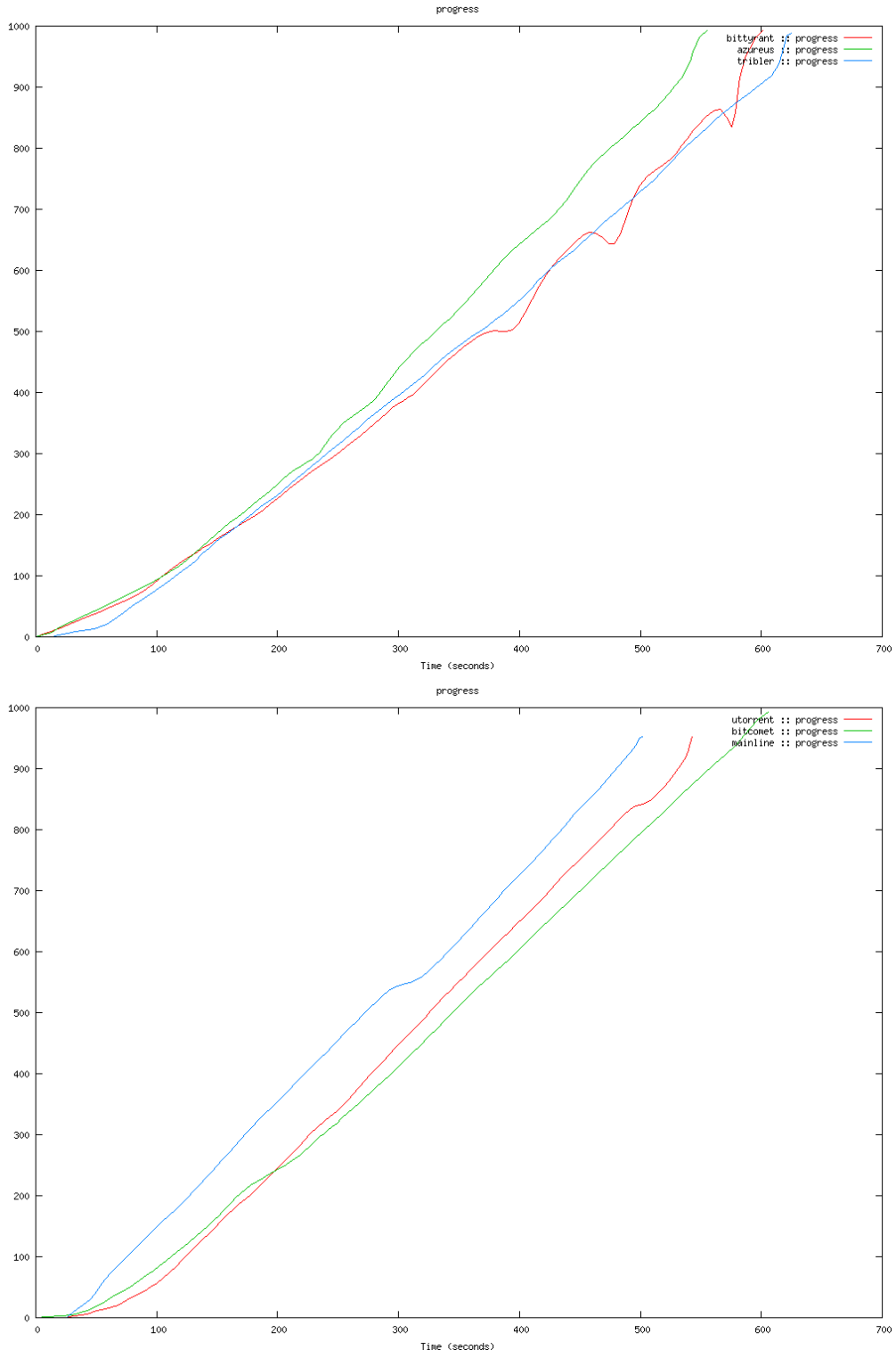


Figure 5.3: Progress in permillage over time.

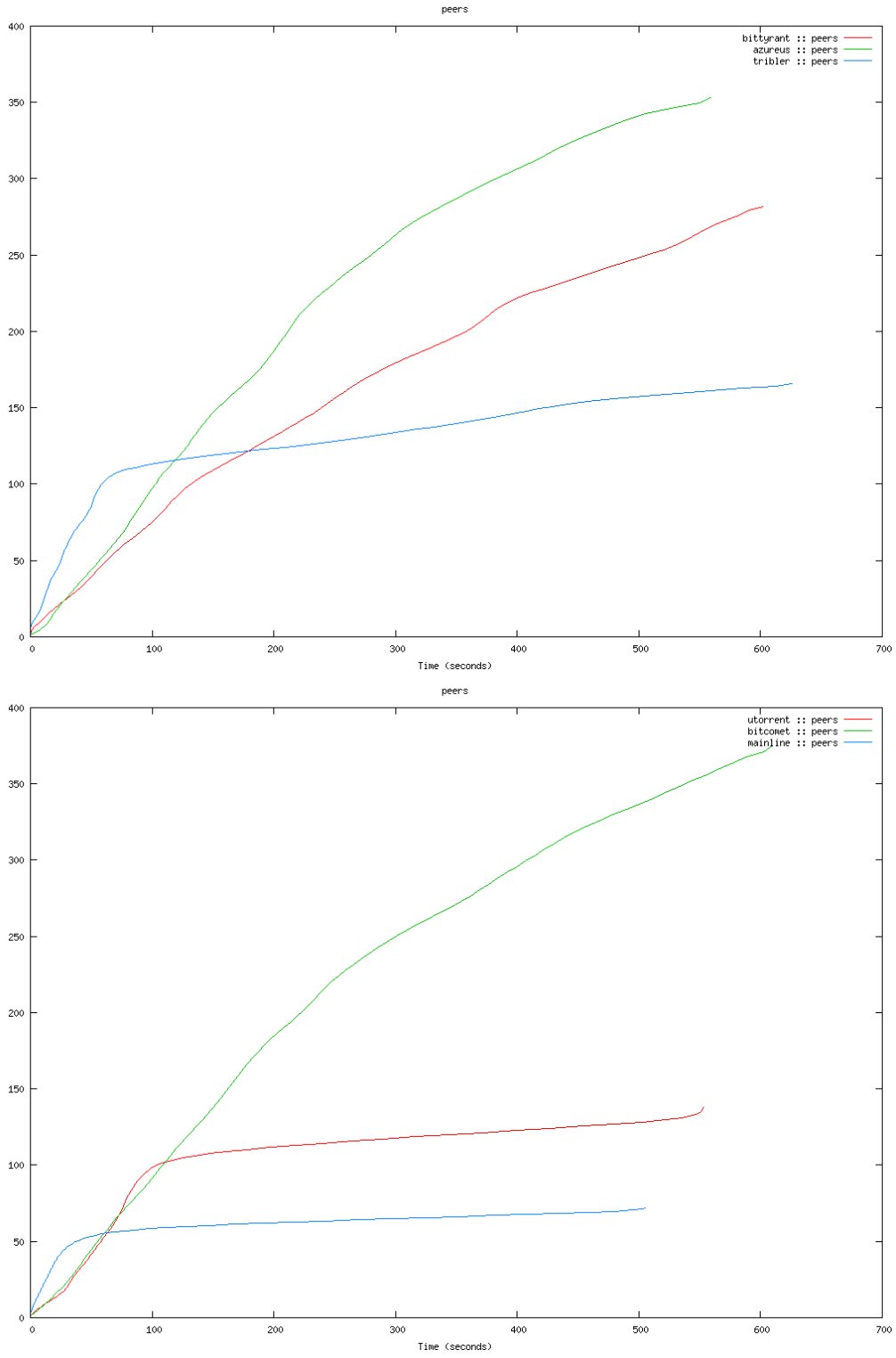


Figure 5.4: Peers discovered over time.

5.3 The unknown factor, DNA

While the adaptive strategy sounds like a plausible reason why Mainline was the fastest client in a test discussed in the previous section, we recently made a shocking discovery. Mainline 6 seems to unadvertisingly install a background task, called Delivery Network Acceleration, or DNA for short. It is not clearly known what it does and it is hard to find some detailed information about it. The Mainline's website does have a page about it and it suggest it accelerates your downloads[7]. Unfortunately, we cannot easily examine the inner workings of DNA and confirm if it is true. Since the release of version 6, the Mainline client is closed source. To make things even more suspicious, uninstalling the Mainline client *does not* uninstall the DNA background task. The only way to detect DNA is to inspect the process list or come across the DNA icon in the control panel.

So why does the Mainline client install this DNA service? Does it really speed up downloads and would this explain the remarkable performances in one of our tests? From an interview with one of the developers[2],[5] we get the suspicion that BitTorrent.com might be selling your bandwidth to other companies, but we cannot say for sure. It would be an interesting case to investigate though, but it is out of our project's scope.

[5]This video interview can also be found through Tribler. Keywords: "navin dna".

Chapter 6

Reflection

In this chapter we will reflect on the planning and delays of the project in section 6.1. We will also list the tools we used for development and discuss their effectiveness. This is done in section 6.2. In section 6.3 our mutual collaboration is discussed.

6.1 Planning and delay

Initially, the project was estimated to take about ten weeks. This was when the project's goal was to perform an analysis on a few open source clients and then implement a better download core for Tribler. For this planning, we had taken into account that certain planned tasks might not be able to be completed in the allocated time, so we reserved two extra weeks.

In the first phase of the project we already encountered a very slight delay. This delay was caused by the difficulties we had with reverse engineering the open source BitTorrent clients. Often, the documentation of these clients were lacking, so we had to manually walk through the code and try to find the right location to log certain events. This usually took a few tries.

The rest of the project went well until the change of the project's goal. When we created a revised planning, we really underestimated the time needed to implement the analysis tool. This was caused by trying to squeeze all the required tasks in what was left of the originally planned ten to twelve weeks. On the other hand, we cannot say for sure that we would have been able to complete the tool in twelve weeks, if we worked on a general analysis tool straight from the start. This is because in the first few weeks of the project, we already gained some knowledge and insights that we would not have had if we started the project with designing a general analysis tool right away.

The delays on the finalization of the project were caused by a large chain of planned and unplanned events. Because of the unexpected events that delayed the project, the project entered a time frame of conflicting, previously planned events.

In order to prevent these delays in the future, we would have to assess the impact of a project change more carefully. If we had done this for this project, we would have discovered the change would require more time than previously allocated and we had to inform our supervisor about it. We would also had to make clear, that the extra required time would mean that the project would take place in a period, where the project members would not be able to work full time on the project.

6.2 Tools used for development

To support our development, we have used a variety of tools for development and communication. Most of the tools were chosen because of our familiarity with them and their proven effectiveness. We will now discuss them.

SVN

At the start of the project, we created a SVN repository. SVN[1] is a version control system, which allows users to add and retrieve files from a repository, while the versioning system keeps track of all the changes. It is a great tool for projects, as all project files are stored in a central place and every change made is visible to all project members. No one is ever working with old files. It is because of these advantages and because of our past experiences that we chose to create a SVN repository.

Eclipse

Eclipse[2] is an extendable Integrated Development Environment, or IDE for short. An IDE bundles all related tasks into a single application. This means a developer does not have to switch applications and can do most of the development tasks in his IDE of choice. The reason why we chose Eclipse is because of its wide applicability and our experience. In our opinion, it really increases productivity. We used Eclipse for the following tasks:

- Python development using the PyDev plugin.[3]
- Modifying the Java BitTorrent client Azureus, using the JDT plugin that is included with most Eclipse distributions.
- Writing LaTeX documents using the TeXlipse plugin.[4]

[1] `http://subversion.tigris.org/`
[2] `http://www.eclipse.org/`
[3] `http://pydev.sourceforge.net/`
[4] `http://texlipse.sourceforge.net/`

In addition, we used the Eclipse plugin called Subclipse[5], so we could commit to and update files from our repositry within Eclipse.

Tribler Wiki

Our supervisor created a wiki entry on the Tribler site.[6] This entry was meant to contain all import insights, graphs and other important documents. In the early phases of the project, we kept the wiki page reasonably up to date, but in the later phases we kept spending more time on development and personal communication than updating the wiki.

In hindsight, this is a shame as we believe that a wiki is helpful and can serve as documentation and thus as external communication of the project's progress. However, we also think that it is a hassle of navigating to the wiki page, log in and update it. In our eyes, the process of documenting is still too much separated from the design and implementation cycles. It would be great if an IDE contained support for updating wiki pages. This would allow the developer to easily update the wiki every time he or she has finished a small part of the implementation. Perhaps committing files to the SVN repository should be linked with updating the wiki?

6.3 Collaboration

Throughout all the phases of the project, we worked well as a team. We achieved this by a great amount of interaction. We kept in contact through instant messaging using chat applications like ICQ and IRC and we often had meetings in person. We also kept each other informed or shared ideas when we were en route by using text messaging on our phones.

As a consequence, we were also able to distribute the workload evenly. By keeping each other informed of the task we were doing at the moment, we could easily give each other directions. We could also easily assign each other a task that needed to be done in the near future. This one-on-one interaction worked well because we were a two-man team, which in our experience are easier to coordinate.

[5] `http://subclipse.tigris.org/`
[6] https://www.tribler.org/DownloadPerformance

Chapter 7

Conclusions and future work

We conclude this thesis with our conclusions based on our acquired insights. The answers to our two performance questions are answered in section 7.1. Although we hoped to acquire more insights into the download performance with our measurement tool, we did not learn much. Still, we gained experience and based on that, we present the work that still has to be done in section 7.2.

7.1 Conclusions

From the latest measurements, we have to conclude that Tribler is unfortunately not the best performing client. If we had to rank the six tested BitTorrent clients, this would be the outcome:

1. Mainline
2. μTorrent
3. Azureus
4. BitTyrant
5. BitComent
6. Tribler

Tribler was 24% slower than the number one client. In order to improve the performance of Tribler – or any other BitTorrent client in general – one has to make sure of the following:

- The client should find fast peers quickly in the beginning phase of a download in order to reach a maximum speed as soon as possible. This is done by trying out a lot of peers.

- Nearing the end phase of a download (called endgame), the client should change its piece request behaviour to prevent any slowdowns.

Additionally, a client might benefit from an adaptive strategies, but we were not able to confirm this. Still, in certain situations a different peer discovery strategy appeared to be more effective.

7.2 Future work

So far, we have only been exploring the surface of the BitTorrent download performance problem. In other to delve deeper and get a greater understanding, we need a better measurement analysis tool. In order to improve our tool, the following has to be done:

- The front-end module of our tool should be replaced. The used PDML format is too restrictive. To do more indepth analysis, we need more control and do more work ourselves. That is, we need to work with raw ethernet frames. Unfortunately, we estimate that this approach will take a lot of time.

- After the replacement, more analysis features can be added. Several examples include storing information about a peer's upload and the detection and limited analysis of encrypted traffic.

Finally, worth investigating is the unexpected performance boost of Mainline version 6. Was the client just lucky with its relaxed peer discovery algorithm or can this algorithm actually benefit from certain situations? If the latter is the case, adaptive strategies are worth implementing in Tribler.
However, it is also possible that Mainline's DNA service is improving the download speeds, although it is not documented. It is also secretly installed, making the process suspicious. Is BitTorrent.com perhaps selling its users' upload bandwidth? Further research should give us a clear answer.

Appendices

Appendix A

Requirements and Design

This document presents the requirements and the design of BitSMART, which stands for **Bit**Torrent **S**peed **M**easurement and **A**nalysis for **R**esearch **T**ool. STATUS: Being modified!

A.1 Requirements

A Problem

The BitSMART software will analyze network traffic generated by a BitTorrent client and present the information in a understandable graphical format.

B Background information

See `http://wiki.theory.org/BitTorrentSpecification`.

C Environment and system models

The BitSMART software is to run on Linux and Windows systems with root or administrator privileges. The measurement itself should take place on the same machine as the running BitTorrent client, which is downloading a single torrent. During the measurement, the IP address of the machine should remain fixed.

D Functional Requirements

R.1. The BitSMART application must be able to read the output of a *pcap*[1] application, created during a single measurement. This requirement can be split into two subrequirements:

[1] Application programming interface for Packet Capturing.

R.1.1. The application should be able to preprocess raw *pcap files*[2] and convert them to an intermediate format in which BitTorrent protocol information is available.

R.1.2. The application should be able to parse this intermediate format. This seems to be a trivial requirement, but it is less so when R.1.1 is performed by an external or foreign software component.

R.2. While reading the output of a pcap application, the BitSMART application should perform an analysis on the BitTorrent traffic. The analysis should be stored in an internal data structure. For this analysis, the *torrent metadata*[3] is needed.

R.3. The analysis of R.2 should provide insights in:

R.3.1. Download speed (progress in percentage over time).

R.3.2. Choking (number of chokes, unchokes and unique unchokes over time).

R.3.3. Peer discovery (amount of peers discovered over time).

This means the created datastructure should contain sufficient information to provide these insights.

R.4. The BitSMART application should be able to store and load the datastructure of R.2 created during analysis to and from a file. Each analysis is automatically stored to a file.

R.5. After analysis of R.2 has been done, the BitSMART application should be able to output graphs of R.3.1, R.3.2 and R.3.3.

R.6. The BitSMART application should be able to load several previously stored analyses and produce combined graphs of R.3.1, R.3.2 and R.3.3.

E Non-functional Requirements

User interface and human factors

The application should be simple in use and runnable from the command line.

Documentation

The process of installing the software tool and the steps required to perform the measurement of a BitTorrent client should be clearly documented.

[2] A pcap file contains captured ethernet frames.

[3] This information is stored in a `.torrent` file.

Performance

There are no performance constraints imposed on the actual analysis and graphing features of the BitSMART application.

System modifications

The system should be designed to be modular, so it can easily be modified and extended in the future.

F Constraints

The system should be written in Python as it is the primary language used by the Tribler team.

A.2 Design

This part of the document outlines the most important aspects of BitSMART's design.

A.2.1 System Overview

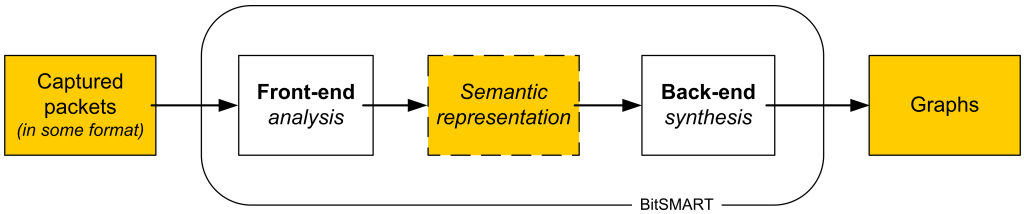


Figure A.1: Input and output behaviour.

The BitSMART system takes captured packets as input and produces several graphs as output. Internally, it contains two main components. The front-end analyses the captured IP traffic and stores it in some datastructure. This semantic representation – which is discussed in section A.2.3 – is then passed on to the back-end. The back-end's task is to create a visual representation of it.

The main advantage of this design is that it is quite easy to change or extend. As long as the internal semantic representation is unchanged and well documented, the front-end can be replaced, either statically or dynamically.

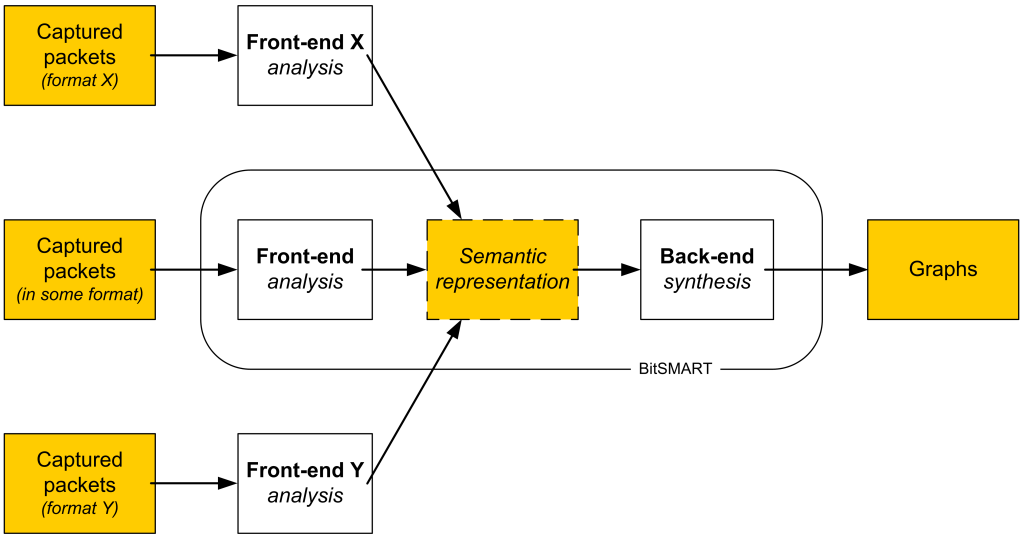


Figure A.2: Change and extension through different front-end modules.

A.2.2 Component View

Figure A.3 is a high-level component overview, showing the connections between the three top level components. Using the network card, the TCP/UDP packet capturer component captures all data that the BitTorrent client sends and receives. Wireshark will fulfill this role in our design, but any packet capturer component can be used.[4] The packets captured by this component will be read by the BitSMART component's front-end through file I/O.

The BitSMART component can be split into the following sub-components. The main application component is the mediator between the front-end components (Figure A.4, right hand side) and the back-end component. This purpose of the main component is to select the right front-end component, give it a file containing captured packets, and pass the results of the front-end component to the back-end.

Front-end Component

We had three options for the front-component, as Wireshark can output the captured packets in several formats:

- Raw (pcap format)
- Plain text
- PDML (XML format)

Of these three formats, the plain text format was not found to be suitable as it did not contain detailed information. So two options were left. PDML is easy to read –

[4]Depending on the output format of the packet capturer, a different front-end might be required.

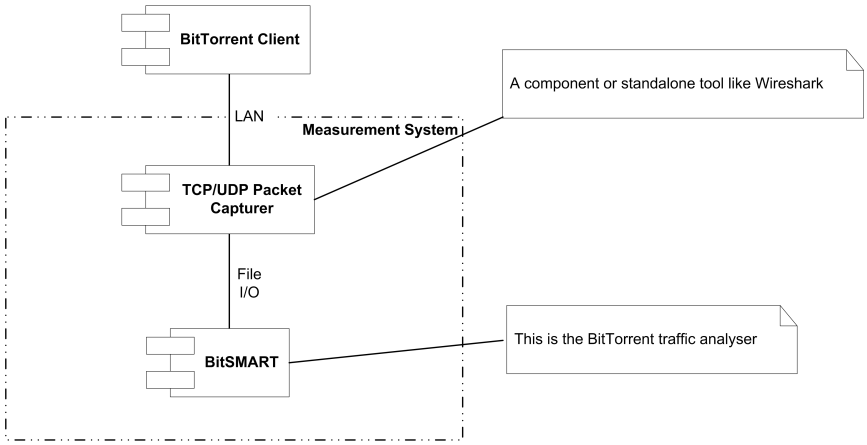


Figure A.3: High-level component overview.

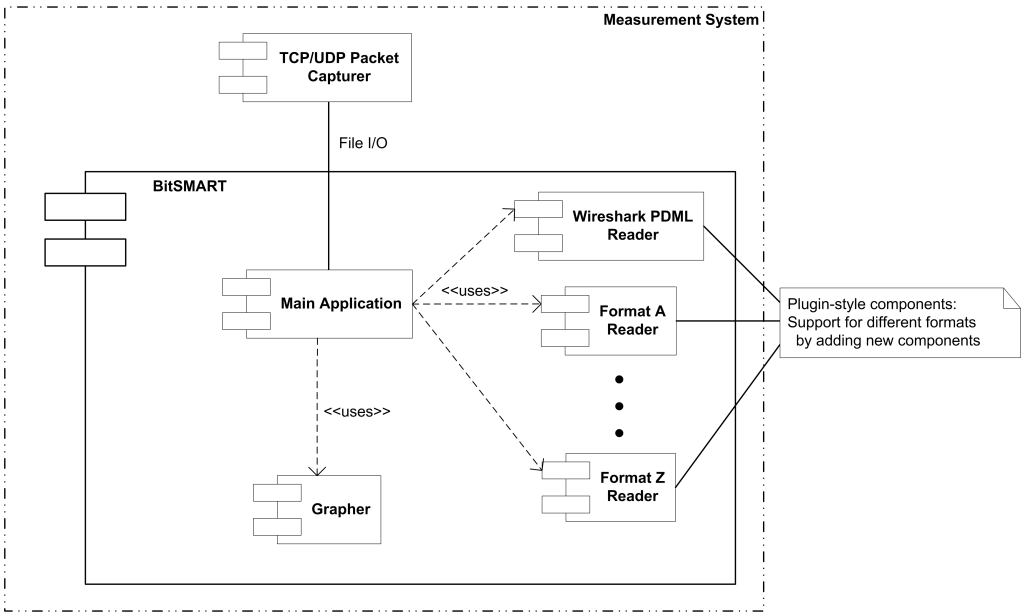


Figure A.4: Internals of the BitSMART component.

it is an XML document – and it contains specific BitTorrent information. The raw format on the other hand contains all network packets, but reading it is not quite as simple. Creating a reader for this format would take a lot of time. The reader would have to be able to re-assemble TCP packets to construct TCP conversations and perform analysis on these conversation to recognize BitTorrent traffic. So, the decision was made to create a reader for the PDML format.

Back-end Component

The back-end component will be based on the graph code we wrote previously.

A.2.3 Class Diagrams

This section describes the datastructure of the internal semantic representation, the interface of the front-end and the interface of the back-end.

Semantic Representation

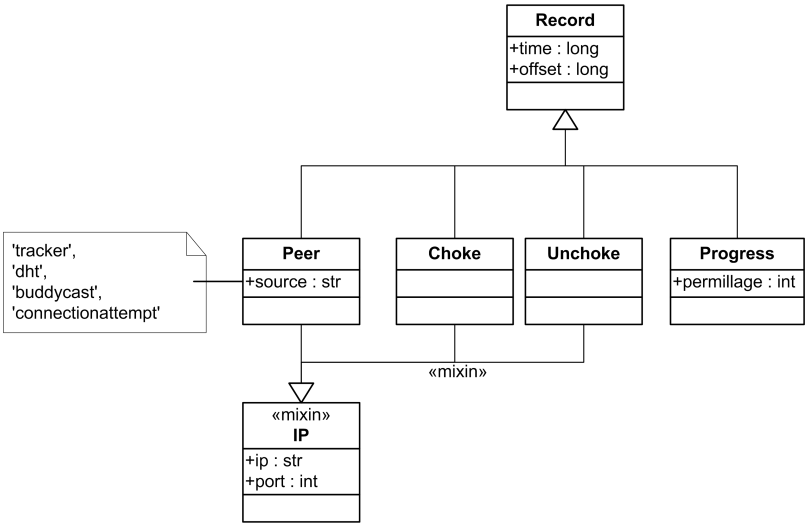


Figure A.5: Semantic representation.

The structure of the semantic representation is based on the previously written code. Each change or event is modeled as a `Record`. Each record contains a time field and an offset field. The offset is the time difference between the current record and the first record.

Front-end Interface

The interface of a front-end module consists of a constructor accepting a file handle or its file name containing captured packets and the dictionary read from a torrent

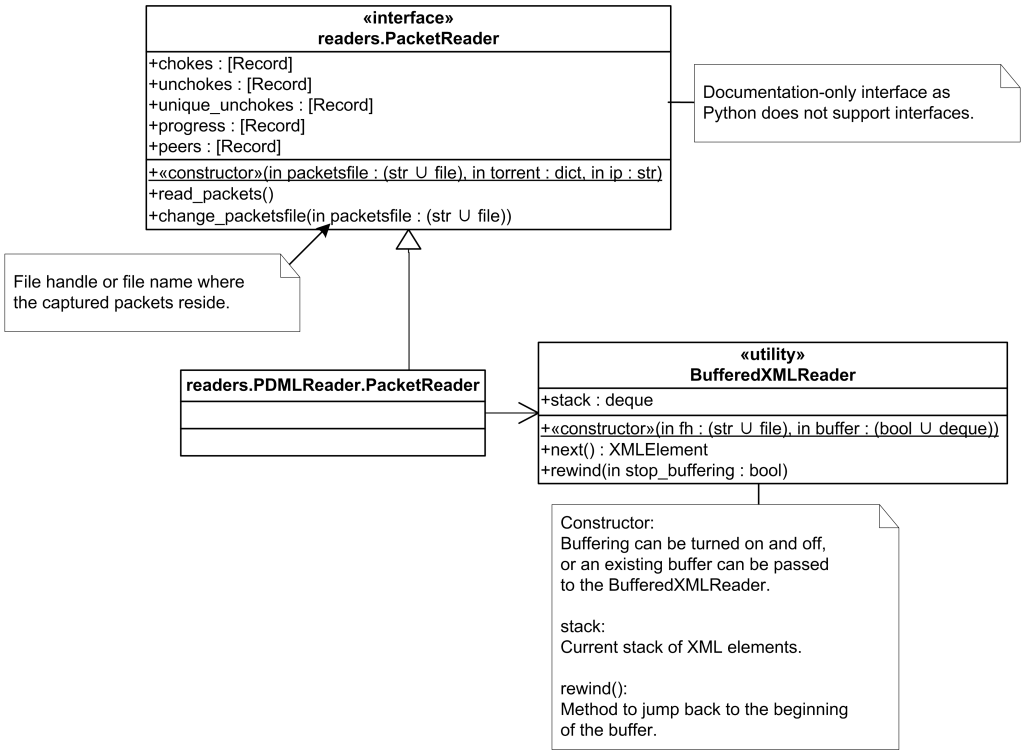


Figure A.6: PacketReader interface and structure.

meta file. By invoking `read_packets()`, the reader will read the file and the results from the analysis will be available in public properties. After each read operation, the packets file can be changed with `change_packets()`. This is used for input spanning multiple files.

The `PDMLReader` class implements all methods and public fields of the front-end interface. It uses the `BufferedXMLReader` to read the PDML format. By iterating over all the XML nodes, the `PDMLReader` extracts the information required.

Back-end Interface

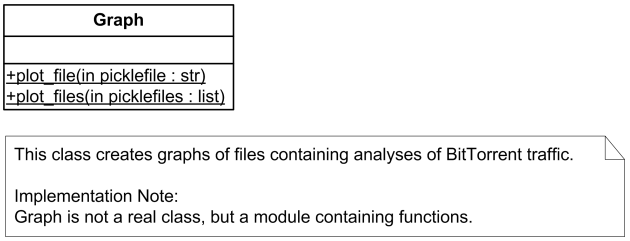


Figure A.7: Back-end interface.

The interface of the back-end module is straightforward. It can create graphs from a single measurement by using `plot_file()` and it can create graphs from several measurements by using `plot_files()`.

A.2.4 Sequence Diagrams

The following diagrams show the sequence of method calls when a user wants to preprocess, analyse and plot a single or several measurements. The general overview is depicted in Figure A.8. A detailed view of reading PDML files is shown in Figure A.9.

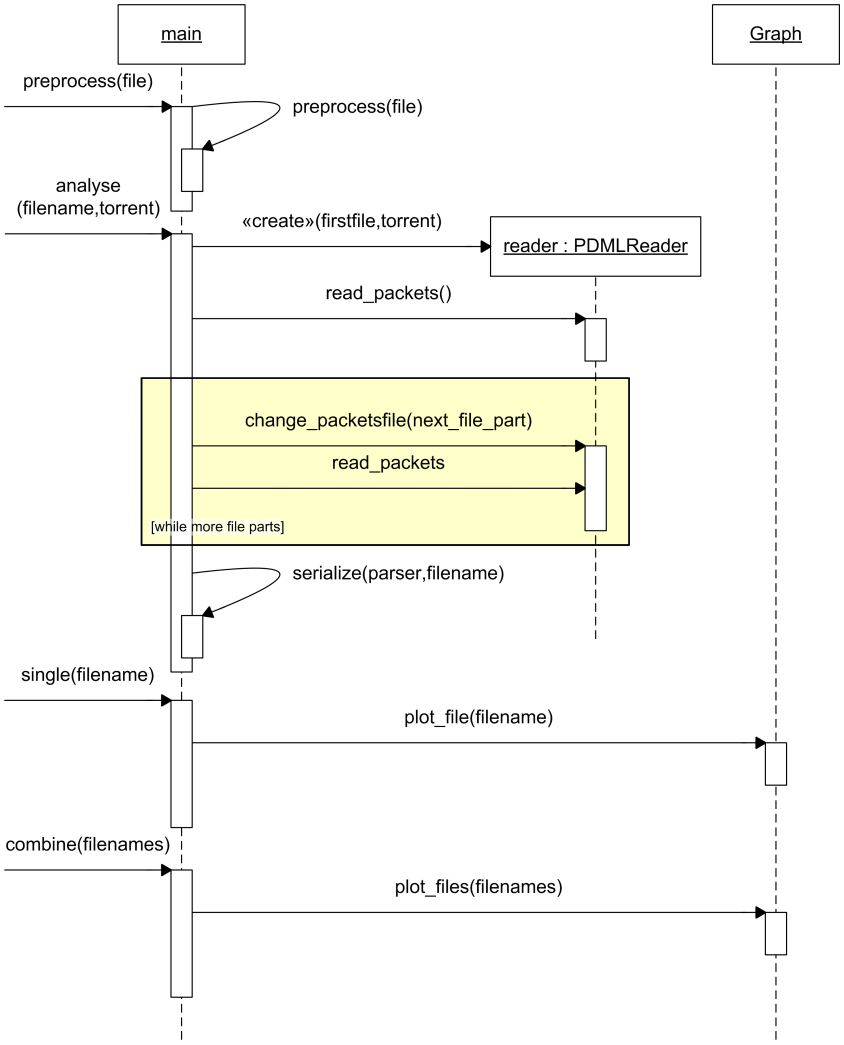


Figure A.8: Sequence diagram of preprocessing, analysing and plotting a single and multiple measurements.

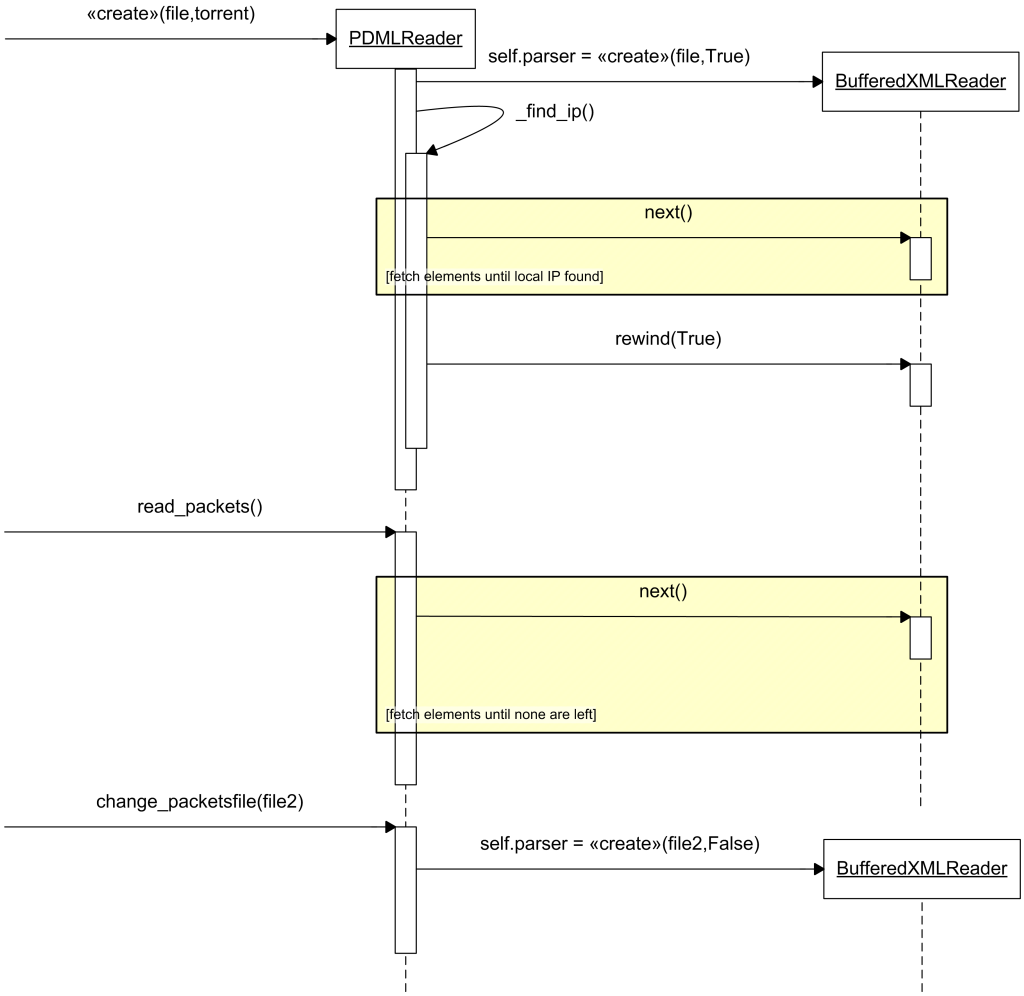


Figure A.9: Sequence diagram of parsing the PDML file.

A.2.5 Algorithms

In this section we will discuss general algorithms that are required for components of the BitSMART system.

Local IP Discovery

A file containing captured packets does not explicitly contain the IP address of the machine on which the packets were captured. We need to analyse the TCP packets to figure out the local IP address, so we know which BitTorrent messages are relevant.

To figure out the local address, we need to iterate through the active TCP connections. A TCP connection consists of two endpoints. One of these endpoints is our local node; the other is some foreign node. Now, if we consider all pairs of

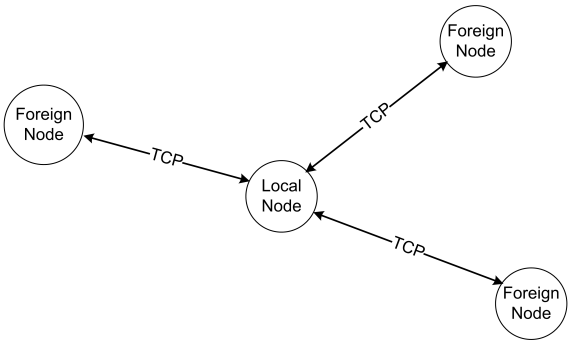


Figure A.10: An example of connections between several nodes.

endpoints, one endpoint will occur in multiple pairs. This endpoint is the local node. So if we store each pair in a dictionary of sets, the entry whose set contains more than one endpoint will be the local endpoint we're looking for. The algorithm based on this idea is shown in Figure A.11.

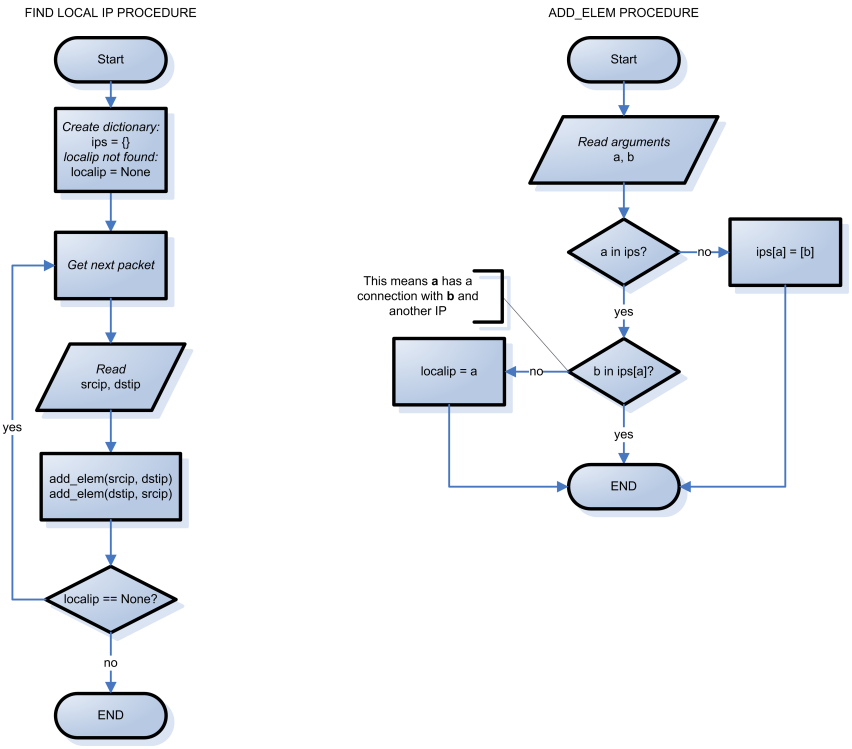


Figure A.11: Local IP Discovery Algorithm.

Appendix B

Test Plan

Overview

Because of the dynamic nature of a language like Python, testing is harder and prone to runtime-errors. Because of this we have chosen to do unit tests and a lot of manual testing.

Tools used for testing

For the unit testing, we have used two of the build-in unit testing frameworks in Python, the *unittest* module and the *doctest* module. The following code snippets are two examples of both the unittest and the doctest module:

unittest

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):
    '''Test sequence functions

    The class inherits unittest.TestCase to enable unit testing'''

    def setUp(self):
        '''Prepare the variables for the tests'''
        self.seq = range(10)

    def testchoice(self):
        '''Test if the random.choice function is working properly'''
        element = random.choice(self.seq)
```

```
        self.assert_(element in self.seq)

if __name__ == ' __main__ ':
    unittest.main()
```

If everything is working correctly, the output of "python file.py -v" should be something like this:

```
Test if the random.choice function is working properly ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

doctest

```
def pow(x, y=2):
    '''Return x raised to the power y

    The lines prepended with >>> will be executed, the lines
    below that, without the >>> are the expected results, if
    the output differs an error is given

    >>> pow(0, 0)
    1
    >>> pow(4, -1)
    0.25

    Create an error and say that we're expecting that error
    >>> pow('test')
    Traceback (most recent call last):
        ...
    TypeError: unsupported operand type(s) for ** or pow():
      'str' and 'int'

    In Python ** is equivalent to x^y (latex style)
      or pow(x, y) (c style)'''
    return x ** y

' Check if we are running as a standalone file or are imported
  as a module '
```

```
if __name__ == ' __main__ ':
    ' import the doctest module and test the module '
    import doctest
    doctest.testmod(exclude_empty=True)
```

If everything is working correctly the output of "python file.py -v" should be something like this:

```
Trying:
    pow(0, 0)
Expecting:
    1
ok
Trying:
    pow(4, -1)
Expecting:
    0.25
ok
Trying:
    pow('test')
Expecting:
    Traceback (most recent call last):
        ...
    TypeError: unsupported operand type(s) for ** or pow():
      'str' and 'int'
ok
1 items passed all tests:
   3 tests in __main__.pow
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

Testing BitSMART

Besides the unit tests discussed in the previous section, we have tested the program thoroughly by giving the program varying input. While doing this, we used Wireshark to check if the output was correct.

For the automated tests for BitSMART, one has to enter the following command:

```
python test.py
```

This command has an optional "-i" or "–interactive" flag to enable the interactive tests for Gnuplot.py. Unfortunately, because of limitations in the Gnuplot.py module, this is the only way to test it currently (Figure B.1).

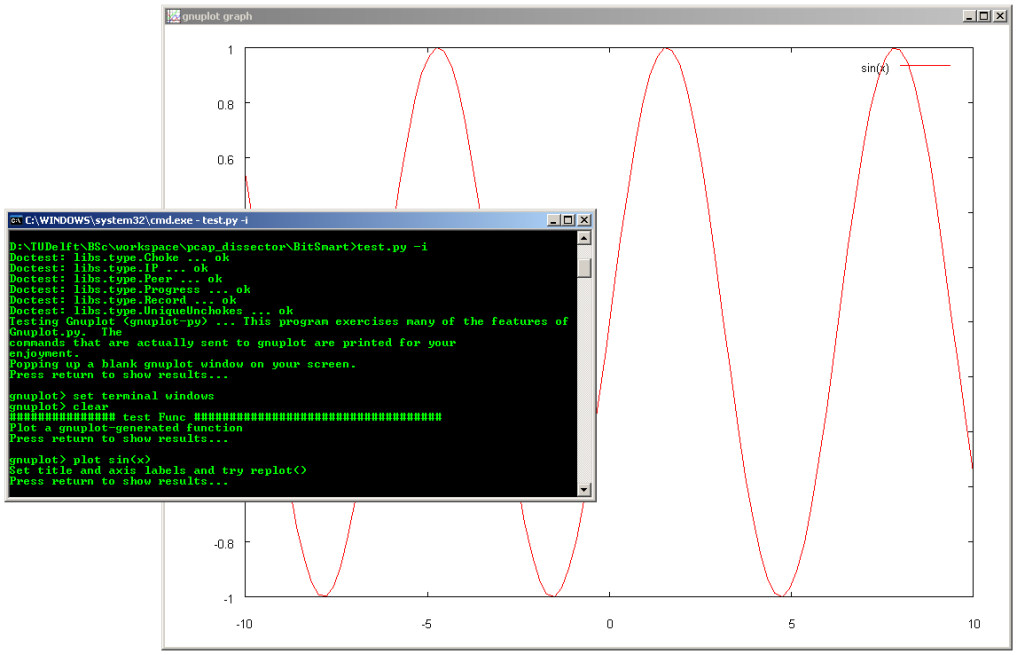


Figure B.1: Testing the Gnuplot.py module.

Appendix C

Deployment

Overview

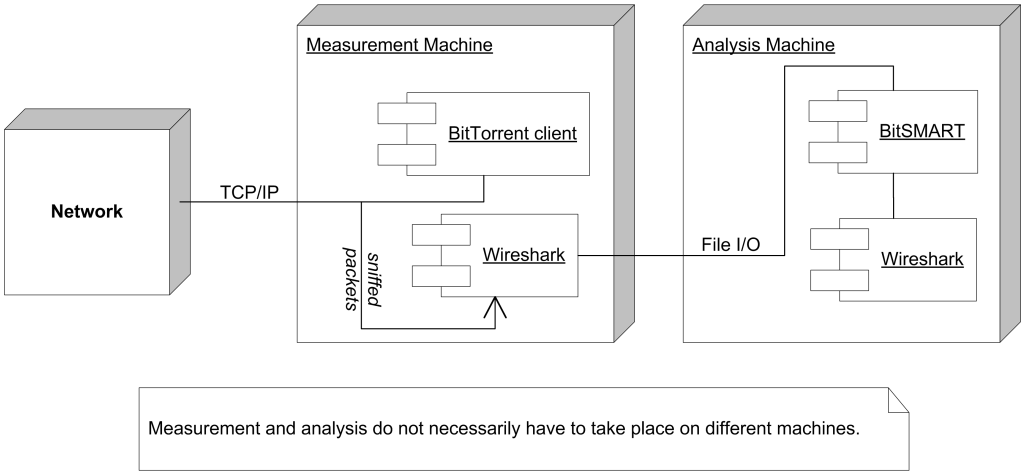


Figure C.1: Deployment diagram.

This document describes the deployment of the BitSMART application.

Measurement Machine

For the measurement machine, it is required that it has the following components installed:

- The BitTorrent client to be tested.
- The Wireshark network protocol analyser tool. `http://www.wireshark.org/`.

To perform the actual measurement, please see the BitSMART manual.

Analysis Machine

For the analysis machine, it is required that it has the following components installed:

- The BitSMART application.
  `https://www.tribler.org/DownloadPerformance`.
- The Wireshark network protocol analyser tool.
  `http://www.wireshark.org/`.
- Python 2.4 or 2.5.
  `http://www.python.org/`.
- gnuplot 4.2 patch level 2.
  `http://www.gnuplot.info/`.
- Gnuplot.py 1.7.
  `http://gnuplot-py.sourceforge.net/`.
- NumPy 24.2.
  `numpy.scipy.org/`

It is recommended that Python, Wireshark and gnuplot binaries are added to the path environment variable.

Deploying BitSMART

BitSMART can be simply deployed by unpacking its archive.

File Descriptions

```
/
├── bitsmart.py ..... Application entry point
├── test.py ..... Test cases
├── libs ..... Libraries directory
│   ├── argparse.py ..... Command line arguments parser
│   ├── bencode.py ..... Bencode encoding and decoding
│   └── type.py ..... Record class hierarchy
├── readers ..... Front-end modules directory
│   └── PDMLReader ..... PDML format reader module
│       ├── packetreader.py ..... PacketReader class
│       └── xmlreader.py ..... BufferedXMLReader class
├── tests ..... Test data directory for test.py
└── tools ..... Tools directory
    ├── graph.py ..... Back-end graphing module
    └── plugins.py ..... Dynamic module loading
```

Appendix D

BitSMART Manual

This manual presents the system requirements and the usage of the BitSMART tool.

System Requirements

The following software packages are required to run BitSMART:

- Python 2.4 or 2.5
  `http://www.python.org/`
  Tested with both Python 2.4 and Python 2.5
- Wireshark
  `http://www.wireshark.org/`
  Tested with Wireshark 0.99.6
- gnuplot
  `http://www.gnuplot.info`
  Tested with gnuplot 4.2 patchlevel 2 (must support both svg and png)
- Gnuplot.py
  `http://gnuplot-py.sourceforge.net/`
  Tested with Gnuplot.py 1.7
- NumPy
  `http://numpy.scipy.org/`

Tested with Numeric 24.2 (note, Gnuplot.py is **not** compatible with recent versions that use "import numpy" instead of "import Numeric")

It is recommended that the Python, Wireshark and gnuplot binaries are added to the path environment variable.

Usage

To use BitSMART to perform a measurement, please follow the following steps.

Preparations

Before performing the measurement, configure the BitTorrent client under test. After configuring the client, please look up the correct network interface with:

```
tshark -D
```

Performing a Test Capture

Before actually starting a measurement, do a test capture:

```
tshark -i network_interface
```

Check if packets are captured when there is network activity. If not, try turning off promiscuous mode:

```
tshark -i network_interface -p
```

If all fails, please read the Wireshark manual.

Measurement

To perform a measurement, execute the following steps:

1. Run tshark:
   `tshark -i network_interface [-p] filename.pcap`
2. Start the BitTorrent client under test and start the download.
3. After the download finished, stop tshark by pressing Ctrl+C.

General information about BitSMART

Path

If Python is not in your PATH, it can be called like this in Linux:

/usr/bin/env python `bitsmart.py -h`

In Windows, one can add the Python folder to the PATH on the Advanced tab of the System applet in the Control Panel.

Output directory

The output directory (where all the files will be placed) can be specified like this:

`bitsmart.py -o` *directory*
or:
`bitsmart.py --output` *directory*

Extra info

If more info is needed, `bitsmart.py -h` *and* `bitsmart.py --help` give an overview of all flags and there descriptions. BitSMART is capable of processing multiple files at once. *filename* means one or more files and the flags affect all of the given files.

Preprocessing

Captured pcap files must be processed before they can be analysed:

`bitsmart.py -p` *filename*
or:
`bitsmart.py --preprocess` *filename*

This reads *filename*.pcap and produces a *filename*.xml file.

Analysing

Preprocessed files can be analysed to convert the xml file to a pickle file:

`bitsmart.py -a` *`filename`*
or:
`bitsmart.py --analyse` *`filename`*

This will generate *filename*.pickle. The latter file is used for combining graphs. If there is no *filename*.xml available BitSMART will try to find a *filename*.pcap and process this file.

Adding a torrent file

The analyse mode has the possibility of adding a torrent file for more acuraccy, by doing this the amount of pieces can be retrieved directly instead of being guessed from the bitfield. The torrent file must be given as an extra flag:

`bitsmart.py -a` *`filename`* `-t` *`torrentfile`*
or:
`bitsmart.py --analyse` *`filename`* `--torrent` *`torrentfile`*

Creating a single graph

A previous measurement can be graphed as follows:

`bitsmart.py -s` *`filename`*
or:
`bitsmart.py --single` *`filename`*

This produces the graphs *filename*.png and *filename*.svg.
If there is no *filename*.pickle available, BitSMART will try to find a *filename*.xml and analyse this file.

Combining Several Measurements Graphs

Previous measurements can be combined and graphed as follows:

`bitsmart.py -c` *`filename1 [filename2 ...]`*

or:

`bitsmart.py --combine` *`filename1 [filename2 ...]`*

This produces several graphs:

- peers.png/svg
- progress.png/svg
- chokes.png/svg
- unchokes.png/svg
- unique_unchokes.png/svg

If there is no *filename*.pickle available, BitSMART will try to find a *filename*.xml and analyse this file.

Bibliography

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, Dec 2004.

[2] Ashwin Navin of BitTorrent, discussing BitTorrent DNA. `http://youtube.com/watch?v=uEF_giyvRaA`.

[3] Azureus - Java BitTorrent Client. `http://azureus.sourceforge.net/`.

[4] BitComet - A C++ BitTorrent client. `http://www.bitcomet.com/`.

[5] BitThief - A free riding BitTorrent client. `http://dcg.ethz.ch/projects/bitthief/`.

[6] BitTorrent - The Official BitTorrent client (mainline). `http://www.bittorrent.com/`.

[7] BitTorrent DNA - Delivery Network Acceleration. `http://www.bittorrent.com/about/dna`.

[8] BitTyrant - A strategic BitTorrent client based on Azureus. `http://bittyrant.cs.washington.edu/`.

[9] CacheLogic. `http://www.cachelogic.com/`.

[10] B. Cohen. Personal announcement: BitTorrent - a new P2P app. `http://finance.groups.yahoo.com/group/decentralization/message/3160`.

[11] B. Cohen. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, 6, 2003.

[12] libpcap - A low-level network monitoring framework. `http://sourceforge.net/projects/libpcap/`.

[13] libtorrent - Open source C++ client. `http://sourceforge.net/projects/libtorrent/`.

[14] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent?, 2006.

[15] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M. van Steen, and H.J. Sips. Tribler: A social-based peer-to-peer system. *Concurrency and Computation (to appear)*, 2007. Accepted for publication.

[16] A.S. Tanenbaum. Computer Networks Fourth Edition. *Person Education International*, 2003.

[17] Tribler, Boudewijn Client.
`https://www.tribler.org/browser/abc/branches/boudewijn/tribler-frayja`.

[18] The Tribler Client.
`https://www.tribler.org/Download`.

[19] Tribler - The Tribler Vision.
`https://www.tribler.org/triblerVision`.

[20] Unofficial BitTorrent Protocol Specification 1.0.
`http://wiki.theory.org/BitTorrentSpecification`.

[21] μTorrent - A lightweight BitTorrent client.
`http://www.utorrent.com/`.

[22] WebSiteOptimization.com. Europe Passes US in Broadband Penetration.
`http://www.websiteoptimization.com/bw/0709/`.

[23] Wireshark - Network Protocol Analyzer.
`http://www.wireshark.org/`.