Anonymous HD video streaming
Peer-to-peer onion router

Risto J.H. Tanaskoski
Anonymous HD video streaming
Peer-to-peer onion router

by

Risto J.H. Tanaskoski

in partial fulfilment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Friday July 4th, 2014 at 11:00 AM.

Student number: 1365177
Supervisor: Dr.ir. J.A. Pouwelse
Thesis committee: Prof. dr. ir. H.J. Sips, TU Delft
Dr.ir. J.A. Pouwelse, TU Delft
Dr.ir. S.E. Verwer, TU Delft

An electronic version of this thesis is available at
http://repository.tudelft.nl/.
Preface

This system would never get to an implementation without the help of R. Plak, N. Zeilemaker, E. Bouman, E. Milon and the rest of Tribler team. Also I would like to thank dr. ir. J.A. Pouwelse for supervising my master thesis project.

Chris Tanaskoski
Delft, June 2014
No scalable privacy-enhancing technologies exists that is capable of anonymous HD video streaming. Our paper discusses the new anonymizer built into Tribler, a social content-sharing client.

With anonymous HD-video streaming as the main objective requirements as at least 10 Mbit/s throughput, user bandwidth donations and NAT-traversal are defined. Using the Tribler API and related tools as Dispersy the ProxyCommunity is designed. This community of proxies provides peer discovery, onion routing and multi-tunnel proxying.

Our system evolved through various stages. From the initial standalone routing prototype, to the first Tribler version. This was followed by profiling to achieve performance improvements. Finally the version with libtorrent and cryptography-readyness was implemented.

Our performance evaluation shows that the proxy community is able to discover others on the network effectively and built circuits with them. Over these circuits the required 10 Mbit/s throughput for HD streaming has been achieved. Preliminary real-world testing shows that the system works in the wild. However more testing needs to be done and important work on our security model remains.
## Contents

1 **Introduction** .............................................. 1

2 **Anonymity and onion routing** .......................... 3
2.1 Chaum’s mixes ........................................... 4
2.2 TOR - The Onion Router ................................ 4
2.3 Tarzan .................................................... 5
2.4 I2P ......................................................... 5
2.5 Gnunet ..................................................... 6
2.6 Attacks on anonymity .................................. 7

3 **Problem Description** ................................. 9
3.1 High-definition video streaming ..................... 9
3.2 User bandwidth donations ............................ 9
3.3 NAT-compatibility ...................................... 10
3.4 Why not TOR? .......................................... 10

4 **Design** .................................................. 13
4.1 Requirements .......................................... 13
4.2 System context — Existing technologies ........... 15
4.3 Network protocol ....................................... 18
4.4 Functional view point ................................. 21

5 **Continuous improvement and feedback** ........... 25
5.1 Standalone routing prototype ....................... 25
5.2 Onion routing community ............................. 25
5.3 Bypassing Dispersy messaging ..................... 30
5.4 Libtorrent and crypto-readyness .................... 32

6 **Evaluation** ............................................. 35
6.1 Real-world trial ......................................... 35
6.2 Synthetic experiments ................................. 35

7 **Future work** ........................................... 41

8 **Conclusion** ........................................... 43

**Experiment scenario** ........................................ 47

**Glossary** .................................................. 49

**References** ............................................... 53
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The RAMPART-A World Wide Tap</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>The anonymity setting (image from Pfitzmann and Hansen)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Chaum’s untraceable email</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>TOR statistics</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Dispersy peer discovery step with NAT-puncturing</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>BarterCast’s limiting effect on false information</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>The anonymising system in context</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>The setting of an onion routing network, viewed from the attacker</td>
<td>18</td>
</tr>
<tr>
<td>4.5</td>
<td>Overview of message types in our network protocol</td>
<td>19</td>
</tr>
<tr>
<td>4.6</td>
<td>Sequence diagram for creating a 3 hop circuit</td>
<td>20</td>
</tr>
<tr>
<td>4.7</td>
<td>Routing table update after extending a circuit</td>
<td>20</td>
</tr>
<tr>
<td>4.8</td>
<td>Protocol stack</td>
<td>21</td>
</tr>
<tr>
<td>4.9</td>
<td>UML class diagram of the ProxyCommunity</td>
<td>22</td>
</tr>
<tr>
<td>4.10</td>
<td>UML class diagram of the SOCKS5 server</td>
<td>22</td>
</tr>
<tr>
<td>4.11</td>
<td>Data flow from SOCKS5 server to the ultimate destination and back</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Standalone prototype; parameters from standard input</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Anonymous libswift download over 3 hops, local hops and local server</td>
<td>28</td>
</tr>
<tr>
<td>5.3</td>
<td>The anonymity panel showing a bunch of 8 hop tunnels.</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Download speed with the new socket handler</td>
<td>32</td>
</tr>
<tr>
<td>5.5</td>
<td>Improved architecture of the SOCKS 5 data interface</td>
<td>32</td>
</tr>
<tr>
<td>5.6</td>
<td>Average download speed for a 200 MB download, local hops, remote seeder</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>Real-world December trial</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Discovery of peers</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Download traces for a 1-hop, 2-hop and 3-hop circuit</td>
<td>39</td>
</tr>
<tr>
<td>6.4</td>
<td>Probability density of latency versus the circuit length</td>
<td>39</td>
</tr>
<tr>
<td>7.1</td>
<td>Multi-tunnel versus Multi-path routing</td>
<td>42</td>
</tr>
<tr>
<td>1</td>
<td>CREATE message byte format</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>CREATED message byte format</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>EXTEND message byte format</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>EXTENDED message byte format</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>DATA message byte format</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>Byte formats for PING and PONG messages</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Top 10 Peak Period Applications during the first half of 2014 .......................................................... 1

5.1 CPU profiling results of the Onion routing community ........................................................................ 31

6.1 Statistics that are being collected during the real-world experiment ................................................. 36
6.2 Real-world December test, speed [MB/s] by the numbers ................................................................ 37
6.3 Latency by the numbers [ms] including the 50th and 95th percentile values .................................. 38
Since the introduction of YouTube video streaming has been eating a major share of the global network traffic. With the introduction of Netflix (and others) this trend only speeds up. At peak usage during the first half of 2014 Netflix alone gobbled 31% of all network traffic in North America [2]. Youtube, iTunes and Hulu are not far behind, see table 1.1. Cisco forecasts that 80 to 90% of all consumer internet traffic in 2018 will be caused by video [3].

Another proof of video’s domination on the internet is the popularity of sports events on the internet. The official Canadian online broadcast of a hockey game during the Sochi 2014 Winter Olympics accounted for 37 % of the network traffic in Canada. During the 2014 FIFA World Cup peaks up to 40 % are expected around the globe according to Sandvine [2].

Many of the data above contains privacy sensitive information. Edward Snowden proved to the world that there exist no such thing as privacy on the internet. Anti-terror laws leave intelligence agencies like the NSA, GHCQ, etc with the power to keep an eye on all our online activity. For example GHCQ claims to be on the right side of the law when tapping external

<table>
<thead>
<tr>
<th>Rank</th>
<th>Upstream Application</th>
<th>Upstream Share</th>
<th>Downstream Application</th>
<th>Downstream Share</th>
<th>Aggregate Application</th>
<th>Aggregate Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BitTorrent</td>
<td>24.53%</td>
<td>Netflix</td>
<td>34.21%</td>
<td>Netflix</td>
<td>31.09%</td>
</tr>
<tr>
<td>2</td>
<td>HTTP</td>
<td>14.27%</td>
<td>YouTube</td>
<td>13.19%</td>
<td>YouTube</td>
<td>12.28%</td>
</tr>
<tr>
<td>3</td>
<td>SSL</td>
<td>6.54%</td>
<td>HTTP</td>
<td>11.65%</td>
<td>HTTP</td>
<td>11.84%</td>
</tr>
<tr>
<td>4</td>
<td>Netflix</td>
<td>6.44%</td>
<td>iTunes</td>
<td>3.64%</td>
<td>BitTorrent</td>
<td>5.96%</td>
</tr>
<tr>
<td>5</td>
<td>YouTube</td>
<td>5.52%</td>
<td>SSL</td>
<td>3.42%</td>
<td>SSL</td>
<td>3.80%</td>
</tr>
<tr>
<td>6</td>
<td>Skype</td>
<td>2.23%</td>
<td>BitTorrent</td>
<td>3.40%</td>
<td>iTunes</td>
<td>3.33%</td>
</tr>
<tr>
<td>7</td>
<td>Facebook</td>
<td>2.17%</td>
<td>MPEG</td>
<td>2.85%</td>
<td>MPEG</td>
<td>2.62%</td>
</tr>
<tr>
<td>8</td>
<td>FaceTime</td>
<td>1.50%</td>
<td>Facebook</td>
<td>1.99%</td>
<td>Facebook</td>
<td>1.83%</td>
</tr>
<tr>
<td>9</td>
<td>Dropbox</td>
<td>1.20%</td>
<td>Amazon Video</td>
<td>1.90%</td>
<td>Amazon Video</td>
<td>1.82%</td>
</tr>
<tr>
<td>10</td>
<td>iTunes</td>
<td>1.15%</td>
<td>Hulu</td>
<td>1.74%</td>
<td>Hulu</td>
<td>1.58%</td>
</tr>
</tbody>
</table>

Table 1.1: Top 10 Peak Period Applications during the first half of 2014 — North America, fixed access (non-mobile). Table courtesy to [2]
1. Introduction

**RAMPART A**

*(TS/SI/NF) Unconventional special access program leveraging Third Party partnerships:*

- High-capacity international fiber transiting major congestion points around the world
- Foreign Partners provide access to cables and host U.S. equipment
- U.S. provides equipment for transport, processing and analysis
- No U.S. collection by Partner and No Host Country collection by U.S. – there ARE exceptions!
- Shared tasking and collection

---

(a) Special Source Operations (SSO) Overview

Presenting the RAMPART-A program.

http://www.information.dk/databloggen/501278

(b) Leaked slide from the Snowden Archives

http://hbpub.vo.llnwd.net/ol6/video/olmk/holt/greenwald/NoPlaceToHide-Documents-Compressed.pdf

Figure 1.1: The RAMPART-A World Wide Tap

Communications. These communications concern data that never has been in or leaves the British Islands at any point. Even if the originator and the destination are in the UK.

Thus a user located in the British Islands posting a message on Facebook will communicate with a Facebook web server. If the Facebook data centre is outside the British Islands then the message will be “external communication”.

(Charles Farr, Director of the Office for Security and Counter-Terrorism ¹)

Months after the initial Snowden-leaks, new information still surprises us regularly. The latest news is the reach of NSA’s RAMPART-A operation, see fig. 1.1. Where at least 30 countries collaborate to tap our private communications on the world wide web.

There are several ways to take control of our own private communications on the net. For low-traffic applications, like browsing web pages anonymously, there are solutions like Tor and I2P. TOR, the market leader in anonymisation, says "Tor isn’t designed for high volume streams" [4]. So for the lion’s share of the Internet’s traffic there exists no scalable privacy-enhancing technology. As all privacy-enhancing technologies are quickly filled and overwhelmed with video streaming and file sharing traffic. Our work aims to be the first anonymiser explicitly designed for high volume streams, used for example to support HD video streaming.

Anonymity and onion routing

In this chapter the concept of anonymity is discussed. Next Chaum’s mixes, an anonymising solution with an age of 30 years, will be explained. His work is at the basis of many modern networks. The core of this chapter will focus on the onion routing systems TOR - The Onion Router, Tarzan and I2P. An anonymous network not based on onion routing, Gnunet will be discussed as well. The chapter ends with some well-documented attacks on anonymous systems.

To avoid making up new terms ourselves we will use the terminology as described by Pfitzmann and Hansen [1]. Their document has been well maintained through the years and we believe using their terminology will make this thesis easier to understand. We will briefly summarise their efforts, starting from the setting, to a definition of anonymity and its specification using the concept of unlinkability.

The setting is simple, see fig. 2.1. There is a set of senders who send messages to the recipients over the communication network. The senders and recipients are subjects of the system. The setting is defined from the the view of an attacker, someone who wishes to compromise the anonymity of certain items of interest. Such items can be messages or subjects. Using the sets of subjects Pfitzmann and Hansen define anonymity as follows.

Anonymity of a subject means that the subject is not identifiable within a set of subjects, the anonymity set.

In short this means that an attacker cannot distinguish the real sender or the recipient, for example, from a group of relays. An important concept is unlinkability. Two items of interest are unlinkable if the attacker cannot decide whether they are related or not. Anonymity can be specified using unlinkability, for example sender anonymity is achieved when the sender cannot be linked to the messages it sends. Similar to sender anonymity there is recipient anonymity, and a (weaker) combination of both is relationship anonymity. For more details we refer you to the paper of Pfitzmann and Hansen [1].
2. Anonymity and onion routing

Chaum was the first to introduce an untraceable anonymous mailing system. The system was based on a Chaumian mix, a relay which hides the sender of the message to the recipient. A sequence of mixes offers even better anonymity. Due to the public-key cryptography scheme the identity of the recipient is only known by the last Chaumian mix and the identity of the sender only known by the first. This is property is achieved by adding a layer of encryption for each relay used.

First the sender encrypts the message $M$ with recipients $A$'s public key $K_a$. Next the sender determines the sequence of mixes to use to relay its message to the recipient. In the reversed order of this sequence it encrypts $K_a(M)$ iteratively with the public key of each mix. When relaying the packet each mix $i$ can only peel off one layer of encryption using its private key $K_i^{-1}$. It then forwards the packet to the next mixer. At the end of the tunnel $A$ receives $K(A)$, which only $A$ can decrypt with its private key $K_a^{-1}$. A short schematic showing which data is sent over the line can be seen in fig. 2.2.

At each mix messages are queued and reordered before forwarding them to prevent timing attacks. Other approaches to “flushing” can be used [5] as well. However queuing of messages means that Chaum mixes are not appropriate for low latency and high bandwidth applications.

2.2. TOR - The Onion Router

In 2002 the second generation Onion Router was presented [6]. With consistently more than 1 million users since 2012 (see fig. 3.1a), TOR is the most popular anonymous network. The name onion routing comes from the fact that several layers of encryption are used, together these layers form the onion. Onion routing can be seen as a special low-latency implemen-
2.3. Tarzan

Tarzan is a peer-to-peer anonymous network developed as a master thesis project at MIT in 2001. The peer discovery is similar to Name-Dropper, a gossiping algorithm [7]. Using gossiping Tarzan finds the (IP-address, port, hash(pubkey)) of all peers on the Tarzan network. When all peers know all other peers they can validate whether a peer does not inject fake entries in the distributed database, protecting against the sybil attack [8].

Circuits are set up by selecting validated peers from their local database and iteratively grow them relay per relay. Selection is not completely random. Assumed is that malicious peers often are grouped in the same IP-prefix the selection requires relays from different prefixes. Within these domains of addresses peers are selected at random [8].

To protect the network against traffic analysis attacks Tarzan imposes additional restrictions on the relay selection. Each peer maintains cover traffic with a fixed set of other peers in the network. Such a link between peers they call a mimic. The first hop is therefore a mimic of the originator and the second must be a mimic of the first and so on. The specifications of the cover traffic guarantee at least $1/3$ of the bandwidth is available to the real data that is forwarded.

An advantage over TOR is that Tarzan is a drop-in-replacement for IP and can therefore tunnel both UDP/IP and TCP/IP over its network. A clear disadvantage compared to TOR is that there has not been any development since 2003 and therefore it is not being used in practice.

2.4. I2P

The Invisible Internet Project (I2P) is a fully distributed and self-organising system, like Tarzan. It can be seen as IP protocol with enhanced security and offers both datagram using their SSU
Anonymity and onion routing protocol\(^2\) and stream characteristics using their NTCP protocol\(^3\).

Similarly to Tarzan a circuit is created using hops from different domains. The domains are given by the /16 subnet of the IPv4 address. As an improvement to Tor and Tarzan, I2P uses garlic routing instead of onion encryption routing. There are circuits and hops as in TOR and Tarzan. However an onion message is never transmitted alone. They are grouped in a garlic clove with another onion and random padding before they are forwarded to the next hop [9].

I2P is designed for hidden services and claims their design is much faster than TOR. Built-in to every I2P client there is a web server. Using this web server the user can create its own website on the I2P darknet. Contrary to Tor it does not define a protocol for accessing data on external networks, however there are proxies on the I2P-network which can tunnel data from and to the internet.

2.5. Gnunet

Gnunet is a framework for reliable anonymous distributed file-sharing\[^{10,11}\]. This is a more specific case of anonymous networking. Therefore additional problems as file lookup and distribution have to be solved.

In Gnunet files are distributed over the network. Big files files are split into fixed 1 KB sized chunks called DBlocks. Groups of 50 hashes of these DBlocks form a single IBlock. On top there is a root IBlock, containing the hashes of its child IBlocks. File information, the description and the hash of this root IBlock is stored in an RBlock.

These blocks are spread around the network to increase availability and to provide sender anonymity. Each block can be fetched using indirection, where an intermediary fetches the block for you. This means that the sender and recipient can always claim they only indirected the message. To avoid that these intermediaries know what they host the IBlocks are encrypted with the hash of the block itself. As the intermediaries don't know the original hashes they cannot decrypt the blocks.

A block B in Gnunet is stored as \(E_{H(B)}(B)\) under \(H(H(B))\), where \(E_k\) is the encryption function using key \(k\) and \(H(x)\) is the hashing function. To find these encrypted blocks and the encryption keys Gnunet offers a keyword-based lookup algorithm. Keywords are stored in the RBlock of a file as the hash of the IBlock \(H(B)\) encrypted with hash of the keyword, \(E_{H(K)}(R)\). For every keyword an encrypted copy of the RBlock is stored as \(E_{H(K)}(R)\) under \(H(H(K))\). To find files matching a keyword \(K\) the searches hashes it twice to \(H(H(K))\) and decrypts incoming results with \(H(K)\). Intermediary nodes perform the search using the double hashed value but do not know which keywords are used, nor the content of the file.

For a broad overview of anonymising networks we refer to numerous surveys on the subject \[^{12,13}\]
2.6. Attacks on anonymity

This section will focus on some well-known attacks on anonymity. First the sybil attack and the eclipse attack will be discussed, followed by a group of traffic analysis attacks.

Before we go in detail on some attacks it helps to be able to categorise the abilities of the attacker. We will assume the taxonomy: Internal-External, Passive-Active, Static-Adaptive [5]. This taxonomy is only one of many. Another example is the taxonomy defined by Syverson et al. [14]. There attackers are characterised as observers, disrupter, hostile user and / or compromised routers.

2.6.1. Sybil and eclipse attacks

When a single faulty entity can present multiple identities we speak of a sybil attack⁴ [15]. This attack can be performed by an internal, active, adaptive adversary. A trivial Sybil attack could be that the first hop in a circuit impersonates the next hops by misleading the originator by creating the identities of these hops locally.

As there is no central authority only cooperative resource demanding challenges can provide resilience to the Sybil attack, under assumptions that are unjustifiable and unpractical [15]. However there are several minimal defenses against the attack. Like introducing recurring costs and fees, auditing and reputation systems [16].

An eclipse attack is an attack where colluding nodes isolate a particular entity such that it can only communicate with malicious nodes, eclipsing it from the rest from the network [17]. A successful eclipse attack breaks all anonymity concepts as it has complete control of the traffic flow from the originator to the exit node and external service. A successful Sybil attack can be used as a starting point for an Eclipse attack, however a Sybil attack is not required. Therefore measures against Sybil attacks offer only limited protection.

In a lot of distributed systems a peer A helps a peer B to discover different peers in the network by sharing the other peers it knows, its neighbour set, with peer B. Malicious peers could share only colluding neighbours and therefore start an eclipse attack. Anonymous auditing as described by Singh et al. limit the number of malicious entries in a neighbour set to the upper bound of $f / (1 - f)$, where $f$ is the fraction of malicious nodes. Other systems like Tarzan and I2P try to limit collusion by applying constraints to its routing systems, like that hops in a circuit should come from different IP ranges [8, 9]

2.6.2. Traffic analysis attacks

Low-latency networks are known to be vulnerable to traffic analysis attacks as messages can be correlated by their send-receive times. In this subsection three classes of attacks will be discussed. The simple but computationally unattractive brute-force attack, the timing attacks and finally the predecessor attack.

Brute-force attack In a brute force attack an external, passive adversary observes all traffic in the network [5]. By following every path a message could have taken. For example starting at the external service the adversary could find the exit node. Monitoring the exit

---

node will show connections to other hops. It will continue in a Breadth-First-Search till a certain level \(d\), the expected length of a circuit. If the adversary is lucky this procedure yields only a few possible paths. It can then focus its efforts narrowing down the number of paths by other, for example by comparing their latencies to the true path taken[18].

**Timing attacks** In a timing attack the send and receive times of messages are used to deanonymise items of interest.

An example of a timing attack, which can be performed by an external passive adversary, is the attack where colluding malicious nodes try to determine whether they are the first and last hop on the same circuit. If this is the case both sender and receiver anonymity are compromised.

The attackers problem is that they do not know whether they are on the same circuit. However they can accurately estimate whether this is the case[19].

The malicious node on path \(I\) will discretise time in non-overlapping and adjacent windows of fixed duration \(W\) and record the number of packets received on its path in each window \(k\) in \(X^I_k\). The colluder on path \(J\) will do the same and record these statistics in \(X^J_k\). By cross correlating \(X^I_k\) and \(X^J_k\) the colluding nodes can guess whether \(I = J\), that they are on the same circuit, with little error margin. In the same paper a defense measure is introduced: cover traffic with defensive dropping [19].

**Predecessor attack** The predecessor attack is an attack performed by a passive external attacker which can estimate who the initiator of an anonymous connection is [20]. The attack depends on the following assumptions.

1. Initiator makes repeated connections to same responder
2. Packets contain information which uniquely identify the current connection
3. Hops in a circuit are selected at random

Each time an anonymous connection breaks a reset occurs where the initiator sets up a new connection toward the same responder. The time between resets is a **round**. Each time an adversaries in the hops forwards a packet to the responder it logs the predecessor of this packet. After \(T\) rounds all participants of the network are logged with equal probabilities due to the third assumption. Except for the initiator, the initiator has a much higher probability of being logged. This difference in probability grows as \(T \rightarrow \infty\).

In onion routing only the last node has uniquely identifying information concerning the current connection, the IP-address of the destination. However only a single extra colluding node is required as first hop to successfully launch a predecessor attack on these networks. The timing attack as described before will help the adversaries to collude, such that the first hop can log the initiator using the uniquely identifying packet information gained by the last hop. In the case of onion routing the number of rounds required for a successful attack is \(O((\frac{c}{n})^2)\), where \(n\) are the number of participants in the whole network and \(c\) are the number of attackers.
As described in chapter 2 there are several systems already deployed on the internet. Before we can evaluate their usefulness it is important to determine the problem we need to solve.

Provide anonymous high-definition video streaming

We are explicitly targeting Tribler, a social based peer-to-peer BitTorrent client [21]. This puts some additional requirements on the implementation. These requirements will be discussed in more detail in 4.1 concerning the systems architecture. In this chapter the focus will not be on implementation details but on the goals the system needs to achieve.

3.1. High-definition video streaming

The goal is to provide an anonymous service that provides enough resources to stream an HD video from the internet. There are countless of different specifications and codecs concerning video streaming. Research shows that Netflix, a popular video streaming solution, uses about 3 Mbit for HD videos [22]. They offer a higher bitrate HD stream (dubbed SuperHD) requiring around 5 Mbit and recommend a line speed of 25 Mbit for 4K streams [1].

In contrary to Netflix our system is a P2P system. Therefore some additional margin is welcome to help buffering the stream; to smooth out the irregularities natural to downloading from a P2P swarm. We hope to come to a solution that achieves a throughput of at least 10 Mbits.

3.2. User bandwidth donations

Tribler has no central infrastructure which could provide anonymous streaming to its current user base. For the system to scale with the number of users the users must bring resources (bandwidth) to the system. Ideally all users bring at least as much bandwidth to the system as they consume. The only way to achieve these goals is to use a peer-to-peer architecture.

3.3. **NAT-compatibility**

Research has shown that most people are behind a NAT. Of all peers using Tribler nearly 90% are behind a NAT. This severely limits the connectability of these computers in a peer-to-peer network. A solution, NAT-traversal, is possible in the case that the peer is behind a P2P-friendly NAT. Related research concludes that about 60% of these peers are behind P2P-friendly NAT [23, 24]. The goal is that users behind NATs can contribute and consume resources from the system like every other user in the network.

3.4. **Why not TOR?**

In chapter 2 four systems were discussed. Gnunet immediately is eliminated as an option as it cannot anonymise BitTorrent-traffic. Unfortunately Tarzan is eliminated as well. The latest source is from January 2003. This means that the project has been deserted a long time ago. Also its cover traffic puts a heavy burden on the networks capacity, at worst case $\frac{2}{3}$ of all resources are lost by design.

The next candidate, I2P, does not support connections outside the network natively. This means that we cannot natively connect to peers in ordinary BitTorrent swarms. Also they use their own Internet-Protocol alternative, including their own flow and congestion control algorithms. These might actually interfere with the protocols we are using.

Therefore only TOR seems promising at first sight. However there are some limitations that we expect to run in to when we would use it as anonymizer in Tribler. These limitations concern the scalability of TOR, protocol-limitations and the lack of NAT-traversing algorithms.

**Scalability**  Although TOR is a distributed system it is not completely decentralised. There is a central registry containing Entry node and Exit node. For TOR to be a scalable system the number of entry and exit nodes must scale with the number of users. This scaling process lags behind the number of users, this is the reason why at unexpected peaks in users (fig. 3.1b) the performance of the network diminishes (fig. 3.1d).

**Protocol limitations**  Due to the chained TCP streams from hop to hop in a circuit there is a substantial performance loss. Data remains idly in buffers caused by TCP’s congestion control. The fact that multiple circuit streams are multiplexed over a single TCP connection causes cross-circuit interference, thus additional performance problems [26].

Since TCP is the underlying transport protocol applications expecting UDP transport characteristics cannot be tunneled over TOR. For instance Libswift and BitTorrent perform application level congestion and flow control which will interfere when TCP congestion control is being used as well.

**Lack of NAT-compatibility**  The TOR-protocol assumes that all hops of a circuit are fully connectable. This means that it is hard to make TOR a scalable P2P system as circuits cannot be set up for the majority of peers without forcing them to correctly set up their routers. An UDP-protocol could offer a system that does support clients behind a NAT or firewall.

We conclude this chapter with the fact that we cannot use existing systems to solve our problem. The next chapter, will describe our design which aims to solve the problems mentioned.
3.4. Why not TOR?

(a) TOR user numbers over the years [25]

(b) Close up; number of users spike

(c) TOR performance over the years

(d) TOR performance drop due to spike in users

Figure 3.1: TOR statistics
We modified the TOR-protocol in order to support anonymous HD video streaming. In this chapter the requirements from the problem description are formalised in more detail. With the requirements in mind a new design will be developed. Next we will focus on the network protocol, before giving more details about the rest of its system architecture.

4.1. Requirements

Our TOR-like protocol is designed to scale, support NAT-traversal and allows users to donate their bandwidth to the network. These and other requirements will be divided into several groups. First we consider the functional requirements, followed by safety requirements.

4.1.1. Functional requirements

In this subsection the behavioural requirements of our to-be-built network are discussed. The requirements will be numbered, such they can be referred to in the rest of this thesis.

Integration with Tribler The goal is to provide anonymity to users of the Tribler download client. Therefore the solution should easily integrate within Tribler. Several points are crucial to yield an easy to use and well integrated anonymous network layer.

R1 Downloading over anonymous networks should be as transparent as possible. This means that it must fit in the normal scenario of downloading a file from Tribler.

R2 Broad support for download engines. Tribler currently uses libtorrent and libswift as download engines. The anonymous layer should support both and should be tolerant to other download engines as well. It is important that these engines work without any modification as it would break compatibility with other clients and put an additional burden on the development team.

R3 The layer should provide an abstract interface. Tribler should not depend on implementation details of the network. The bare necessities to start an anonymous download should do.
Protocol requirements  As libswift and BitTorrent’s μTP both use the UDP protocol, therefore they expect UDP characteristics as datagram based messaging and they implement congestion and flow control themselves.

  R4  UDP-based protocol
  R5  No congestion and flow control, nor any retransmission as this is implemented by the download engines

Scalable system  A scalable system is important as there are no resources to provide everyone with high speed proxies. Also a scalable system is self providing and handles peak loads as the example in fig. 3.1b better.

  R6  Each participant should contribute to the capacity of the network. See section 3.2.
  R7  The system must be able to find other participants in the network without the use of a central authority or a centrally hosted directory service.
  R8  There must be several tunnels to minimise the influence of slow peers in the network

Performance  Performance is an issue while creating anonymous networks. There is a trade-off in anonymity and performance. For example: Chaum mixes (section 2.1) offer resilience to timing attacks but increases latency to an unacceptable amount. To set some hard requirements we specify constraints on the bandwidth and latency of the network.

  R9  Bandwidth of at least 10 Mbit/s to provide the video streaming services in Tribler with enough data. Motivation for the number 10 Mbit/s can be found in section 3.1.
  R10  Latency should be restricted as application protocols used in BitTorrent and Swift use latency as an indicator of line quality in congestion and flow control. Ideally it should not grow more than linearly with the length of the tunnel in the number of hops.

Cryptography-ready  It should be possible to integrate several encryption models and adjust the parameters of the system without editing the core. As this falls outside the scope of this master thesis project a modular approach is crucial to aid work done by Plak.

  R11  Encryption ready. Even though the product is without any cryptography, there should be primitives and hooks available for a cryptography module in the future.

NAT compatibility  As mentioned in section 3.3 there are too many computers behind a NAT or firewall to ignore. The design should be able to utilise peers behind a P2P-friendly NAT.

  R12  Clients behind a P2P-friendly NAT should be able to download and donate through the anonymising network
4.1.2. Privacy-preserving requirements

Safety is a broad concept, for this project we define safety as the concept of anonymity. It is safe to use the system if you remain anonymous. We would like to keep the same anonymous properties like TOR.

R13  Preserve TOR quality when possible

To contain the complexity of the design the scope needs to be restricted. The priority of the solution is to provide anonymous high-definition video streaming to Tribler users.

A certain level of trust is needed to avoid sybil attacks. Trust is an open problem in peer to peer networks and distributed systems in general. A lot of research has been done focusing on this particular subject. However there is no consensus for a single solution [16]. In the future we will use BarterCast [28], a reputation system. The BarterCast identities will be used as identities on the anonymous network. As these are not cheap to create it will help against attacks using many malicious identities, like the Sybil and Eclipse attack. During the next section we will summarise the characteristics of Bartercast.

Cryptography is at the core of anonymous networks. As part of another project cryptography models will be developed for our anonymous network [27]. Therefore the scope of this design is limited to providing right primitives such that he can do his work.

Tarzan’s cover traffic model is elegant yet effective, however it will not be implemented at this stage. However there will be room in the protocol to do this at a later stage. More on cover traffic can be found at chapter 7. To leave room for cover traffic, messages are not to be padded in the initial implementation.

4.2. System context — Existing technologies

Tribler has seen continuous development for several years and offers several libraries and API’s which we can consume. It makes sense to identify the tools which are available to us for ‘free’.

Tribler Core

As can be seen in fig. 4.3 the end user only operates on the GUI component of Tribler. The GUI operates on the Tribler Core, the package containing the high level API of Tribler. When a user enqueues a download the download engines libtorrent and libswift are eventually responsible for the download process. The anonymising layer needs to interface some how with these engines.

Dispersy

Another important part of Tribler are its communities. These are maintained by Dispersy, a project developed at the same Parallel Distributed Systems (PDS) department at the TU Delft. Dispersy offers decentralised data dissemination in challenged networks. These are networks suffering from intermittent connectivity, delays and absence of end-to-end paths [29].

To bootstrap peer discovery Dispersy contacts one or more trackers to find the initial peers. The discovery continues in steps. In fig. 4.1 we see a single discovery step in Dispersy. Both pairs (A, B) and (B, C) know each other and can send messages to each other. Instance
Figure 4.1: Dispersy peer discovery step with NAT-puncturing

\(A\) receives \(C\)’s address from \(B\) as respond to \(A\)’s introduction request. However both \(A\) and \(C\) are behind a NAT and cannot communicate with each other yet. To puncture this NAT instance \(B\) sends a puncture request to \(C\). Upon receiving the request \(C\) sends a packet to \(A\), puncturing his own NAT. At this stage \(C\) can receive packets from \(A\) and when \(A\) sends a packet to \(C\) it will puncture its own NAT as well. Now all three instances can communicate with each other.

It makes sense that our anonymous network uses Dispersy for peer discovery (requirement \(R7\)) and for messaging between peers. As it relieves us from responsibilities like NAT-traversal and bootstrapping. Not only it eases the burden on our design, it also avoids duplicate code.

**BarterCast**

High speed proxied downloads can only be achieved if a significant amount of users is willing to donate these resources. As incentive mechanism the reputation system BarterCast has been developed at the PDS at the Technical University of Delft.

BarterCast is an epidemic protocol in which peers broadcast real-time upload and download statistics to others they know. Peers compute each others reputation based on their local view of direct and indirect traffic information.

Direct traffic information concerns traffic statistics of its own peer \(A\) with other peers \(B\), this information is known to be correct. This direct traffic is shared with others. For these others this becomes indirect traffic information. Indirect traffic information contains all the statistics from a known other \(C\) with a third party \(B\).

Using this accumulated data a directed graph can be made containing upload and download statistics, both direct and indirect. The reputation of a peer is then derived from the maxflow of the graph, where the capacity of a directed edge is equal to \(\text{uploaded} - \text{downloaded}\). This limits the effect of false information as the maxflow is bound by the edge with the smallest flow. In figure fig. 4.2 can be seen how the maxflow algorithm reduces the effect of false information.

We can see a BarterCast identity as a proof of work. If we use these identities in the future the sybil-attack would be much more expensive for the adversary as it would need to actually upload a lot before a newly generated identity can be used. See the chapter 7 for more information.
4.2. System context — Existing technologies

(a) B lies about direct traffic  
(b) B lies about indirect traffic  
(c) B and D collude and lie

Figure 4.2: BarterCast’s limiting effect on false information from node B [30]

Figure 4.3: The anonymising system in context
4.3. Network protocol

This section elaborates on the network protocol used for the anonymous network layer in Tribler. Since TOR’s design is widely reviewed, and is actually used by many on the internet, it will be starting point for our network protocol. However requirement R12 implies that UDP should be used as transport protocol, as it allows NAT-traversal.

Next to computing entities in the anonymous network we will discuss entities outside the network. We will call entities outside the network an external service, see fig. 4.4. The peer in the network initiating contact to such an external server we will call an originator.

4.3.1. Message definitions

The protocol stack used by our network is based on UDP datagrams to conform with set requirements. A datagram contains a single message. The protocol is similar to TOR’s.

Each message is prefixed with a 4 byte circuit number. This number is chosen by the initiator of a circuit and together with the initiator’s IP-address it is an unique identifier for the circuit. The 5th byte denotes the message type. There are seven distinct message types as listed in fig. 4.5. The rest of the byte format is message-type depended. The complete message formats can be found in the appendix on Message byte format.

4.3.2. Conversations

The messages are used in several distinct type of conversations between peers in the system. From creating a system at the originator, to creating a circuit for someone else, to relaying data forward and back.

Creating a circuit

The creation process is initiated by sending a CREATE message (fig. 1). There are two distinct cases when a circuit is made. In the first case the originator initiates the creation of the circuit. In the second case the current end of the circuit creates a circuit by request of the originator, extending the original circuit. It is important that the recipient of the CREATE cannot distinguish between these two cases, otherwise the identity of the originator can be trivially determined. Both cases occur in the sequence diagram in fig. 4.6.
CREATE Each time a node wants to create a new circuit it sends the CREATE message to the first hop.

CREATED When a node confirms a CREATE request it sends back a CREATED message.

EXTEND When a node wants to extend its circuit it sends an EXTEND message along the circuit.

EXTENDED If the circuit has been extended the EXTENDED message propagates back to the origin.

DATA Transmitting data between nodes is done using the DATA packet. If a packet cannot be routed it will be sent over a direct line, the sending node acts as EXIT node.

PING Request a PONG for keep-alive purposes, also used for circuit breakdown detection.

PONG Reply to incoming PING messages

---

When the originator wants to create a new circuit it sends a CREATE message to the first hop, containing the circuit id the originator assigned to this circuit. When the first hop accepts the CREATE request it sends a CREATED message back as confirmation.

**Extending a circuit**

When the circuit between the originator and the first hop has been established, the circuit is ready to be extended. The originator has no idea whether the first hop has a NAT, nor to which proxies the first hop can connect to. To solve this problem a hop returns a list of extend candidates¹ together with the CREATED message.

The originator can select a proxy from this list by selecting a public key. This key is then given to the first hop in the EXTEND message which creates the circuit between itself and the new hop. **We acknowledge this procedure is a weak point in the current design as it is vulnerable to both the sybil and eclipse attack.** Later this public key should be replaced by the BarterCast identity of the extend candidate. Such that the originator can decide whether it is a reputable candidate or not.

Whenever the ProxyCommunity extends a circuit for someone else it creates a new entry in the routing table, stored in relay_from_to, see fig. 4.7. From now on upcoming packets will be forwarded to the new next hop.

The extension procedure is done iteratively, until the circuit reaches its desired length and is marked ACTIVE.

**Data transport across a circuit**

Active circuit can be used to transport data over and back. The originator wraps the payload in a DATA message and forwards it to the first hop. The first uses the routing table to determine

---

¹This candidate list also helps us implement the mimic cover traffic method used by Tarzan at a later stage
that it needs to forward it. When the message reaches the end of the circuit at hop \( n \) the packet is stripped to the raw payload and sent to the ultimate destination, the responder.

When the responder sends a message back to hop \( n \), the hop wraps it in a DATA message and sends it back along the circuit. This wrapping and unwrapping leads to a protocol stack as can be seen in fig. 4.8.

**Keeping the circuits alive and broken circuit detection**

As UDP is a connectionless protocol we have to implement some mechanism to check whether our circuits are still online. We have implemented a simple ping-pong scheme. The initiator of a circuit sends these at regular intervals when a circuit is idle. If there is no reply on the pings it will break the circuit. In that case intermediary relays will notice that no traffic is flowing through them and remove the entries in their own routing table as well.
4.4. Functional view point

In this section the functional and behavioral properties of the design will be discussed. A major component of the design is the `ProxyCommunity`, the community of proxies. Next modularity will be discussed. However a major part of this section is about the data interface of our system, the link between the `download engines` and our new system.

### 4.4.1. ProxyCommunity

From the system context it was decided that dispersy will be used for peer discovery and messaging. To this end a new `community` has been defined: the `ProxyCommunity`. With responsibilities as peer discovery and messaging it takes its place in the core in our system. Also it is the class responsible for building circuits and relaying packets for circuits of others. It offers primitives to send and receive data over the circuit that it creates. To send data it offers the `tunnel_data_to_end()` and `tunnel_data_to_origin` methods.

**Peer discovery and circuit building** At regular intervals the `ProxyCommunity` asks Dispersy for known members in the community. It will send CREATE messages to these candidates to create multiple tunnels. With multiple tunnels we comply to requirement R8.

**Modularity** The ProxyCommunity uses the observable design-pattern to increase modularity. Whenever data is received on the circuits the `ProxyCommunity` will notify its `TunnelObservers`. This event will be used by the data interface to return these incoming packets to the initiator.

Other events concern circuit / relay breaking, sending / receiving data as exit node and incoming `STATS` messages. These `STATS` messages will be discussed in chapter 6. Currently there are several observers. The `DefaultExitStrategy` uses the events to perform EXIT node duties, the `StatsCollector` generates statistics based on the events and the `SOCKS5` data interface uses a `TunnelObserver` as well.
**4.4.2. Data Interface**

Requirement **R2** dictates that a standardised protocol should be used by our system to interface with the download engines. The SOCKS5 protocol is a widely used, standardised, simple, yet sufficient protocol. It has been described in RFC 1928 [31]. The client-side of the protocol is already implemented in the **libtorrent** download engine. That leaves us with development of a client in the **libswift** engine and a server implementation which forms the link between the **ProxyCommunity** and the engines. We would like to note that the choice for SOCKS5 has additional value for libswift as it will suddenly support thousands of proxies available online.

As can be seen the **SOCKSServer** is a **TunnelObserver**. It listens to the event of incoming data and acts upon the breakage of a circuit.

**Circuit selection** After the **ProxyCommunity** has created (several) circuits the SOCKS5 server comes online and accepts incoming connections. Upon an UDP-relay request it has the opportunity to decide which circuit to use. There is one single restriction in circuit selection: If
4.4. Functional view point

Figure 4.11: Data flow from SOCKS5 server to the ultimate destination and back

**a circuit has been selected for the destination address before, it must use the same circuit again.** This constraint comes from the fact that we cannot change our endpoint address as viewed by the external service. Currently our implementation uses a round-robin selection strategy, such that all circuits are utilised. In this case we are not bottlenecked by a slow peer in a single circuit.

**Download recovery upon circuit failure**  If a circuit fails the bittorrent or libswift peer connections running over this circuit are broken. We cannot change to another circuit as it would change our endpoint address as seen from the external service. Most, if not all, external services will drop packets from unknown addresses. To solve this problem we force a new handshake by re-adding the affected peers to the download. The next time a packet is received for these peers they will be mapped to a different, working, circuit.

An overview of the data flow through the three networks, the local SOCKS5 interface, the P2P anontunnel protocol and the plain UDP protocol at the exit nodes can be seen in fig. 4.11.
Continuous improvement and feedback

By using four continuous improvement cycles we have implemented and deployed anonymous HD video streaming. This methodology is used to learn by doing and limit the amount of work at each step. Such that results are achieved quickly and it gives testable intermediary versions.

The goal of the first iteration was to simply forward data over a number of hops. In the next iterations features like peer discovery, external data interfaces and Tribler integration. This chapter will end with the last implementation, the version available at the time of writing.

5.1. Standalone routing prototype

The standalone prototype has been developed as a standalone console application written in Python. Its purpose is to get TOR-like routing working over UDP sockets. In this version routing can be altered and extended at runtime.

A simplified version of the protocol described in section 4.3 was used. Instead of the byte format as published in the appendix, the messages were serialised using pickle. Pickle is part of the Python libraries and serialises Python object hierarchies into a bytestream and back. This decision was purely based on time constraints. Pickling and unpickling only takes a few lines of code, where manual serialisation takes significant extra efforts.

Peer discovery was not part of this first prototype. Also there was no data interface yet with Tribler. This means that the routing and message data had to be passed along in a different way. The first version takes both from the standard input stream as illustrated in fig. 5.1

5.2. Onion routing community

The second improvement cycle focused on getting base functionality as peer discovery, autonomous peer selection and circuit building working. This section will discuss how Dispersy
helps us with peer discovery and messaging. As dispersy uses its own messaging protocol
the anonymisation protocol has been adapted to use dispersy messages. This will be the
second topic of this section. Next the developed data interface, which will link the down-
load engines to our proxy, will be presented. Finally this section will conclude with the first
measurement of performance.

5.2.1. Peer discovery
The systems architecture as described in section 4.2 uses Dispersy peer discovery. Dispersy
maintains a list of communities. Each community is identified by a public key. Users can
join these communities and discover other members by a walk. First core bootstrap servers
will guide the member to some other members. And from these members the user can walk
further to discover any other. Dispersy makes sure that NAT-boxes and firewalls are traversed
such that a walk may return these members behind a NAT. To discover other proxies on the
internet a new community has been created. In this iteration a start was made with this
ProxyCommunity.

Dispersy has several modes of communication, from point-to-point to full synchroni-
sation of messages amongst the community. As for every dispersy community the Proxy-
Community’s messages must be defined. For each message type several settings must be
configured regarding message signing and destination policies. As the anonymous network
will provide onion routing signing messages is not required. Therefore the NoAuthentication
policy is set. Since proxies send messages directly from hop-to-hop, instead of syncing each
message to everyone, the CandidateDestination policy is set. Finally all participants are al-
lowed to send these messages, denoted by the PublicResolution policy.

5.2.2. Autonomous peer selection and circuit building
In contrary to the standalone prototype in section 5.1 this version selects peers autonomously
and build circuits with them. Currently a circuit will be created for each peer discovered as
long as we have insufficient circuits. The length of the created circuit is determined by the
value the used Circuit Length Strategy. This value, the desired number of hops, is stored in a
new instance of the Circuit class.

Next a CREATE message is sent to the peer. Upon completion of the circuit between the
originator and the first hop a new peer will be selected if the circuit has not reached it desired
length yet. The set Extend Strategy decides which peer to use for the next hop. An EXTEND
message is sent to the tunnel and the first hop will extend the circuit as discussed previously.
in section 4.3.2.

5.2.3. Download interface

As the architectural description stated a SOCKS5 interface is used to redirect outgoing packets of libswift (and in the future libtorrent) over the created circuits. The interface consists of two parts a SOCKS5 client and a server. First the server will be discussed in more detail before this section concludes with the new libswift SOCKS5 client.

**Server** The SOCKS5 server implements a subset of the RFC 1928 protocol [31]. Its main limitation is that it only proxies UDP packets. It listens to a TCP port for incoming SOCKS5 connections and accepts these only if the ProxyCommunity has ready circuits. Upon the clients request the server sets up an UDP relay port. Every packet received on this port is mapped to a circuit by its destination. By default destinations are mapped round-robin to the available circuits, however there are several different Circuit Selection Strategies.

To provide a return link for the data the server listens to the ProxyCommunity for packets coming back on the circuits used by the server. These packets are returned back over the UDP relay to the download engine.

The server also listens to broken circuit events. When a circuit, for any reason, goes offline the server checks whether the circuit was allocated to this session. If this is the case the peers will be re-added to the download and assigned to another circuit.

**Libswift SOCKS5 client** LibSwift had no proxying interface yet. To support the SOCKS5 protocol a separate C++ class Socks5Connection has been created. Each channel, a connection between two swift peers, share the same instance of this class. The SOCKS5 connection is created if the command line arguments contain the proxy argument along with the IP-address, port tuple as parameters. Immediately upon launch libswift sends the opening handshake and requests an UDP relay. While there is no UDP relay channels are not allowed to sent any messages.

After the UDP relay has been established packets sent by every channel (in Channel::Send) are redirected through the proxy instead. A SOCKS5 header will be prepended to the data-gram. Upon incoming packets in Channel::Receive the SOCKS5 header will be parsed and the originators address will be set as the datagrams source before being passed back to LibSwifts original code.

When the server disconnects the SOCKS5 connection libswift will reset the state associated to all channels. This important since when a new circuit will have another exit address and therefore the swift protocol will need to redo the 3-way handshake to make sure the IP address is not spoofed.

The SOCKS5 code uses only existing dependencies, namely libevent, to perform its task. Therefore no further system requirements were introduced. The component has been tested on both Windows 7/8 and Ubuntu 12.04/13.04/13.10.

5.2.4. Tribler integration

The Anonymity panel has been made, it shows all current circuits created by the proxy. Each circuit is also represented in a graph where each relay of the circuit is a node in the graph.
Continuous improvement and feedback

Figure 5.2: Anonymous libswift download over 3 hops, local hops and local server.

For more information a log has been added as well, showing all important events related to circuit building. For an example see fig. 5.3.

Also a new download state has been introduced, the Waiting for tunnel state is shown when libswift is waiting for a SOCKS5 connection to the proxy. Therefore informing the issue that it is not quite ready yet to start downloading. It is also used to decide when a download actually starts in the experiments.

5.2.5. First measurement data
The first measurements show that the important components of the system work. However the performance is abysmal. The test’s set up is a local Swift seeder, 3 local hops and a Tribler instance. A screenshot of the result can be seen in fig. 5.2. Clearly the achieved performance, peaks up to 6 Mbit, is not the result we are looking for. A quick look at the CPU usage confirms our initial suspicious, the performance is clearly CPU bound.
Figure 5.3: The anonymity panel showing a bunch of 8 hop tunnels.
5.3. Bypassing Dispersy messaging

We created a performance breakdown using the Yappi¹ profiler to investigate the poorly performing proxy. After profiling the the application in CPU-time and wall-time no clear bottleneck was found, as can be seen in table 5.1.

If we group functions by category it an be seen that a lot of time is consumed handling incoming and outgoing Dispersy messages. As Dispersy supports many different messaging models there is quite some overhead parsing them. If a gain was to be made all these stages should be bypassed.

5.3.1. Options of improvements

It is clear that we needed to shorten the path between receiving data on the socket and acting on its content. There are multiple options available to achieve this goal. Three where considered: creating a new socket loop dedicated to the anonymous tunnel, use the experimental Tribler’s Single-Socket or modify the current socket handler to divert AnonymousTunnel packets away from Dispersy.

A new dedicated socket loop Contrary to the ad-hoc event loop of Dispersy, a standalone event loop for our system could be used. For example a loop based on Twisted, which has proven to be a versatile high-performing event loop. The disadvantage of this approach is that yet another port needs to be used. Which means we need to do our own NAT-puncturing.

Tribler’s Single-Socket Another solution to the problem is Tribler’s Single-socket. To reduce the number of ports used by Tribler a project has been started to multiplex all data over one libswift socket. Swift runs an efficient libevent event loop. However the Single-Socket is not fully functional yet. Therefore it cannot be deployed at the moment of this decision.

Modifying the current socket handler The least intrusive option of the three is to adjust the current socket handler. The idea is to prefix our messages, as the single-socket approach would. When incoming packets have this prefix they will bypass Dispersy and be delivered in its raw format to the ProxyCommunity. The advantage is that a switch to the Single-Socket is easy when it becomes available. Also there wont be additional NAT-traversal problems.

5.3.2. Changes to architecture

We have opted to implement the last option, modification of the current socket handler. Inevitably the message serialization protocol was altered, all messages are crafted and parsed in our own code.

An additional advantage of this new approach is that packages do not have to be parsed completely. From the first few bytes it can be determined whether a packet is meant for us or needs to be relayed. In terms of relaying this should yield a considerable performance improvement.

Together these improvements helped us achieve a huge performance improvement. While downloading a 1 GB file the speed climbed up to a steady 5 MB/s, as can be seen in fig. 5.4.

¹ For more information about yappi see https://bitbucket.org/sumerc/yappi
### Table 5.1: CPU-time profile of a test download over the anonymizing network layer

<table>
<thead>
<tr>
<th>Times called</th>
<th>Total time [s]</th>
<th>Method</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>26701</td>
<td>6.352</td>
<td>tokenize_loop</td>
<td>Profiler</td>
</tr>
<tr>
<td>2696104</td>
<td>4.090</td>
<td>BlockFinder.tokenizeater</td>
<td>Profiler</td>
</tr>
<tr>
<td>112803</td>
<td>3.957</td>
<td>Implementation.<strong>init</strong></td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>79479</td>
<td>2.635</td>
<td>RawserverEndpoint.send</td>
<td>Socket I/O</td>
</tr>
<tr>
<td>26702</td>
<td>2.086</td>
<td>Dispersy.on_message_batch</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>79469</td>
<td>1.876</td>
<td>Message.impl</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>66465</td>
<td>1.865</td>
<td>SocketHandler.handle_events</td>
<td>Socket I/O</td>
</tr>
<tr>
<td>79479</td>
<td>1.710</td>
<td>RawserverEndpoint._process_sendqueue</td>
<td>Socket I/O</td>
</tr>
<tr>
<td>304973</td>
<td>1.695</td>
<td>_RLock.acquire</td>
<td>Proxy</td>
</tr>
<tr>
<td>78761</td>
<td>1.641</td>
<td>DispersyTunnelProxy.send_data</td>
<td>Proxy</td>
</tr>
<tr>
<td>620870</td>
<td>1.640</td>
<td>Logger.isEnabledFor</td>
<td>Proxy</td>
</tr>
<tr>
<td>79469</td>
<td>1.629</td>
<td>ProxyConversion.encode_message</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>112792</td>
<td>1.593</td>
<td>Implementation.<strong>init</strong></td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>676226</td>
<td>1.544</td>
<td>Implementation.<strong>init</strong></td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>304973</td>
<td>1.464</td>
<td>_RLock.release</td>
<td>Proxy</td>
</tr>
<tr>
<td>33345</td>
<td>1.462</td>
<td>DefaultConversion._decode_message</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>79259</td>
<td>1.348</td>
<td>ProxyCommunity.send</td>
<td>Proxy</td>
</tr>
<tr>
<td>26617</td>
<td>1.337</td>
<td>Dispersy.on_incoming_packets</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>54733</td>
<td>1.179</td>
<td>getsourcefile</td>
<td>Proxy</td>
</tr>
<tr>
<td>26686</td>
<td>1.173</td>
<td>Callback.register</td>
<td></td>
</tr>
<tr>
<td>99214</td>
<td>1.152</td>
<td>ProxyCommunity.fire</td>
<td>Proxy</td>
</tr>
<tr>
<td>620642</td>
<td>1.070</td>
<td>Logger.getEffectiveLevel</td>
<td></td>
</tr>
<tr>
<td>78761</td>
<td>1.065</td>
<td>_encode_data</td>
<td>Proxy</td>
</tr>
<tr>
<td>82764</td>
<td>0.934</td>
<td>getfile</td>
<td></td>
</tr>
<tr>
<td>48521</td>
<td>0.883</td>
<td>UdpRelayTunnelHandler.data_came_in</td>
<td>Proxy</td>
</tr>
<tr>
<td>26697</td>
<td>0.795</td>
<td>Dispersy.store_update_forward</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>323754</td>
<td>0.789</td>
<td>Logger.debug</td>
<td></td>
</tr>
<tr>
<td>79468</td>
<td>0.739</td>
<td>ProxyConversion._encode_direct_distribution</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>112803</td>
<td>0.730</td>
<td>Implementation.<strong>init</strong></td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>113346</td>
<td>0.722</td>
<td>is_address</td>
<td></td>
</tr>
<tr>
<td>28031</td>
<td>0.718</td>
<td>findsource</td>
<td></td>
</tr>
<tr>
<td>32767</td>
<td>0.703</td>
<td>DispersyTunnelProxy.on_data</td>
<td>Proxy</td>
</tr>
<tr>
<td>396602</td>
<td>0.692</td>
<td>&lt;genexpr&gt;</td>
<td></td>
</tr>
<tr>
<td>78761</td>
<td>0.669</td>
<td>decode_udp_packet</td>
<td>Proxy</td>
</tr>
<tr>
<td>79358</td>
<td>0.660</td>
<td>DispersyTunnelProxy.active_circuits</td>
<td>Proxy</td>
</tr>
<tr>
<td>133518</td>
<td>0.659</td>
<td><strong>instancecheck</strong></td>
<td></td>
</tr>
<tr>
<td>12719</td>
<td>0.637</td>
<td>_Condition.wait</td>
<td></td>
</tr>
<tr>
<td>32755</td>
<td>0.632</td>
<td>Socks5Server.on_tunnel_data</td>
<td>Proxy</td>
</tr>
<tr>
<td>644831</td>
<td>0.626</td>
<td>_RLock._note</td>
<td></td>
</tr>
<tr>
<td>60047</td>
<td>0.600</td>
<td>Dispersy._convert_batch_into_messages</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>79469</td>
<td>0.593</td>
<td>ProxyCommunity.get_conversion_for_message</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>112803</td>
<td>0.590</td>
<td>Implementation.<strong>init</strong></td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>112803</td>
<td>0.545</td>
<td>Implementation.setup</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>59962</td>
<td>0.517</td>
<td>Dispersy._convert_packets_into_batch</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>26713</td>
<td>0.517</td>
<td>Dispersy.on_batch_cache</td>
<td>Dispersy messaging</td>
</tr>
<tr>
<td>267015</td>
<td>0.515</td>
<td>lower</td>
<td></td>
</tr>
<tr>
<td>190453</td>
<td>0.484</td>
<td>Logger.info</td>
<td></td>
</tr>
<tr>
<td>237809</td>
<td>0.483</td>
<td>_RLock.<strong>exit</strong></td>
<td></td>
</tr>
<tr>
<td>132946</td>
<td>0.481</td>
<td>RelativeTime.get_time</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: CPU-time profile of a test download over the anonymizing network layer results show that a lot of time is spent in dispersy messaging functionality.
Figure 5.4: Download speed with the new socket handler, improving the performance by a factor in the order of 10

This surpasses requirement R9 and leaves headroom for any performance drop due to cryptography at a later stage.

5.4. Libtorrent and crypto-readyness

Up till now our focus was on libswift. However BitTorrent is the more popular network, so it is actually libtorrent that is the most used engine in Tribler. Therefore this final cycle brings libtorrent-integration. Also we have implemented support for cryptography modules and several performance improvements.

Multiple download engine support So far the SOCKS5 server implementation only supports only a single client. This means only a single download engine can be connected at the time.

To support multiple clients several changes have to be made. The key idea behind these changes is to achieve complete isolation between libtorrent and libswift, such they never receive each others data. To this end each SOCKS5 Session maintains its own pool of circuits. This guarantees that incoming data on a circuit is destined for a single SOCKS5 client. At server level keeps a pool of circuits to be used for future sessions. These pools are filled by the ProxyCommunity After the pool is filled to a certain threshold the SOCKS5-server comes online and starts accepting requests. The changes to the data interface can be seen in fig. 5.5.

Whenever a session receives a request to setup an UDP relay a subset of the reserved circuits is allocated to the session. If the pool runs out of free circuits these UDP-associate requests fail until the ProxyCommunity fills the empty spots in the server’s pool.

With these changes libtorrent support is easily configured by setting the correct SOCKS5 proxy address. To avoid leaking data at the application layer, libtorrent is instructed to use anonymous mode. Anonymous mode enforces that all data, to peers and trackers, is sent over the SOCKS5 proxy and disables DHT and the PeerID is stripped of the client’s fingerprint.²

²http://libtorrent.org/manual.html#session-settings
5.4. Libtorrent and crypto-readyness

Related work on the system as performed by R.S. Plak[27] focuses on the cryptography model our system. To aid his implementation several changes have been made such that the crypto module can be plugged in without too much changes to the core. To prove the methodology works a quick crypto test was done, see fig. 5.6b. With these adjustments our system complies with requirement R11.

- **Method `send_message`** which applies a list of decorators to the input. Cryptography module can read and alter the content of messages here before they are serialised and send to the other participant.
- **Method `send_packet`** which applies a list of decorators to the serialised message data. The complete payload of the message can be encrypted by one of the decorators.
- **Method `on_packet`** which calls a decorator on the incoming serialised data. A crypto module could use this hook to decrypt incoming data.

Performance improvements By optimising message (de-)serialisation and other miscellaneous refactoring operations and optimisations a performance gain has been achieved. The testing machine reaches an average speed of 7 MB/s as can be seen in fig. 5.6a.

PEX support - Poor man’s anonymous seeding An adjustment concerning exit nodes has been made. Now the exit node uses a single port per circuit. Which means that each packet can be mapped deterministically to a single originator. Packets of unknown origin will be forwarded just the same as packets from external services the exit node knows about. This means that we use a user-level full cone NAT strategy at the EXIT node, although it effect can be limited by the NAT in front of the exit node.

A real world example of packets from unknown hosts is Peer-EXchange protocol (PeX), where peers share other seeders and leechers they know with other peers. Thereby peers in the swarm discover others in a distributed environment.

In our case the exit node’s address will be shared with others. Others will contact the exit node and this node will relay it back towards the originator. With this small change it becomes possible to anonymously seed what you have downloaded.
6.1. Real-world trial

As the anonymising layer is developed for Tribler, a production-level download client, real-world data is available to us. To gain some knowledge about its performance a simple experiment has been developed: a test download using the anonymizing proxy. The test was conducted by using libswift as download engine.

During the download some statistics are being recorded. Upon completion (or when the user shuts down Tribler) the statistics will be openly shared with all other members in the community. This full sync of the distributed database is a standard feature of Dispersy. As can be seen in table 6.1 no privacy sensitive data is being shared.

The first public trial was performed in December 2013. A portable Microsoft Windows version of Tribler was published on the forum, together with the source code for Linux users. At start-up of Tribler an anonymous download will be scheduled. The circuit length was determined at random between 0 and 4. Where 0 would be effectively equal to not using a proxy at all. In total 83 trials were done, by an unknown number of users.

As can be seen in fig. 6.1a and fig. 6.1b the performance was in generally poor. However we have no data of the connections and relays used, therefore the only conclusion we can draw from this quick test is that the downloads actually worked. However after this initial test many performance improvements have been implemented. The synthetic experiments will give us a better idea of the systems performance.

6.2. Synthetic experiments

For the synthetic experiments Gumby, an experiment runner framework for Dispersy and Tribler, has been used. Gumby allows us to script a scenario and run it on a distributed cluster and gather and aggregate measurements from all instances running on each node. The cluster used for these experiments is The Distributed ASCI Supercomputer 4 (DAS-4), specifically the location at the Vrije Universiteit of Amsterdam. This cluster has 74 nodes, each node is configured with two quad core CPUs at 2.4 GHz and 24 GB RAM. The nodes are connected
<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid</td>
<td>GUID</td>
<td>identifier representing the test</td>
</tr>
<tr>
<td>encryption</td>
<td>bool</td>
<td>whether onion encryption has been enabled</td>
</tr>
<tr>
<td>download size</td>
<td>long</td>
<td>bytes downloaded</td>
</tr>
<tr>
<td>download time</td>
<td>long</td>
<td>seconds it took to download these bytes</td>
</tr>
<tr>
<td>bytes enter</td>
<td>long</td>
<td>bytes received as EXIT node</td>
</tr>
<tr>
<td>bytes exit</td>
<td>long</td>
<td>bytes sent as EXIT node</td>
</tr>
<tr>
<td>bytes return</td>
<td>long</td>
<td>bytes returned to SOCKS5 client</td>
</tr>
<tr>
<td>broken circuits</td>
<td>int</td>
<td>the number of broken circuits during the test</td>
</tr>
<tr>
<td>circuits</td>
<td>CircuitStats[]</td>
<td>circuit statistics</td>
</tr>
<tr>
<td>relays</td>
<td>RelayStats[]</td>
<td>relay statistics</td>
</tr>
</tbody>
</table>

**CircuitStats**

<table>
<thead>
<tr>
<th>hops</th>
<th>int</th>
<th>length in hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes up</td>
<td>long</td>
<td>bytes uploaded</td>
</tr>
<tr>
<td>bytes down</td>
<td>long</td>
<td>bytes downloaded</td>
</tr>
<tr>
<td>time</td>
<td>time</td>
<td>circuit download time in seconds</td>
</tr>
</tbody>
</table>

**RelayStats**

<table>
<thead>
<tr>
<th>bytes</th>
<th>long</th>
<th>bytes relayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>time</td>
<td>relay time in seconds</td>
</tr>
</tbody>
</table>

Table 6.1: Statistics that are being collected during the real-world experiment

Using gigabit ethernet and InfiniBand interfaces at QDR speeds.¹

We will start with measurements of four download traces, a trace of a zero, one, two and three hop download. The traces will give us the bandwidth achieved, the load on the processor and how the number of hops affects these values. Next we will show some averaged data on the bandwidth achieved and the latencies caused by our network.

All experiments in this sector are conducted without the cryptography module developed by Plak, we direct you to his thesis for crypto-enabled experiments.

During the experiment we simulate 1000 users of our network. We do this by creating 50 instances on 20 DAS-4 nodes. First we start a 0-hop download, which we can use as a baseline. The 200 MB download test file used is hosted as a BitTorrent seed on the same cluster.

Two minutes later, when peer discovery and circuit building has finished, the latency test is initiated. A minute later we let 3 random instances sequentially download the test file, each with a different circuit length. After 10 minutes we wrap up the experiment by stopping all instances. With help of some custom R-scripts Gumby will generate the output of the experiments. The Gumby scenario used can be found in appendix Experiment scenario.

6.2. Synthetic experiments

<table>
<thead>
<tr>
<th>hops</th>
<th>tests</th>
<th>$\mu$ [MB/s]</th>
<th>$q_5$ [MB/s]</th>
<th>$q_{50}$ [MB/s]</th>
<th>$q_{95}$ [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27</td>
<td>1.39</td>
<td>0.20</td>
<td>0.81</td>
<td>5.21</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0.64</td>
<td>0.06</td>
<td>0.75</td>
<td>1.35</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>0.59</td>
<td>0.11</td>
<td>0.32</td>
<td>2.70</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>0.20</td>
<td>0.01</td>
<td>0.07</td>
<td>0.57</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.20</td>
<td>0.07</td>
<td>0.23</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 6.2: Real-world December test, speed [MB/s] by the numbers

(a) Average speeds versus circuit length, including median and 25th/75th percentile

(b) Speeds distribution versus circuit length. Vertical lines are mean values.

Figure 6.1: Real-world December trial

The results of these experiments can also be found online\(^5\).

**Peer discovery** Dispersy provides us with peer discovery. To test whether we other proxies could be found we have set up a community of 1000 proxies. Each of them join the community and discover peers and get discovered by other peers.

As can be seen in fig. 6.2a the number of peers discovered grows steadily during the experiment. At the end each peer knows about 20 other peers in the network, according to fig. 6.2b. These figures prove that we comply with requirement \textbf{R7}. As all discovered peers can be selected for data relaying we also comply with requirement \textbf{R6}

**Bandwidth and CPU load** In this experiment we have created oversupply, there are only four instances that download and there are 1000 of relays to chose from. The goal of this experiment is to see how the implementation performs when the network is not the bottleneck.

As can be seen in fig. 6.3a the implementation yields about 5 MB/s for an one-hop circuit. This is about half of the 0-hop configuration, where the initiator’s \textit{ProxyCommunity} acts as its own EXIT node. For two and three-hop circuits the performance is only slightly affected. We attribute this to the fast network used in the DAS-4 cluster.

\(^5\)\url{http://jenkins.tribler.org/job/Experiment_AnonTunnel_gumby/224/label=das4_vu/}
From the CPU load, more specifically the time spent in user land code, it can be seen that the bandwidth is limited by computational power. This is no surprise as this was also the case in the first version as described in section 5.2.

During the one-hop download two nodes consume the most resource. The initiator is under the highest load, as it has the most work to do. The other peak is the first-hop, which also acts as exit-node. The two-hop and three-hop download are similar. They respectively show 3 and 4 active instances on the cluster.

The difference in performance between the different hop count should become more apparent when onion cryptography is enabled. In that case the amount of work at the initiator is \( n \) times as big as the other hops, where \( n \) is the number of hops. Preliminary tests revealed in chapter 5 showed we still comply with requirement R9. In any case future improvements of this performance are welcome.

**Latency** The latency has been measured as the round-trip time of a DATA message that is sent to the end of a circuit, and then echoed back to the originator. For each circuit length (1, 2 or 3 hops) around 33000 packets where sent. The results can be seen in table 6.3. As you can see the average latency scales less then linear from 1 to 3 hops. This complies with requirement R10. To get an idea of the variation the probability density plot is provided in fig. 6.4. As can be seen the variance grows with hop length. Also the median latency is under the average latency, which is to be expected.

<table>
<thead>
<tr>
<th>hops</th>
<th>( \mu ) [ms]</th>
<th>( q_{50} ) [ms]</th>
<th>( q_{95} ) [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.22</td>
<td>0.80</td>
<td>1.77</td>
</tr>
<tr>
<td>2</td>
<td>2.22</td>
<td>1.30</td>
<td>5.22</td>
</tr>
<tr>
<td>3</td>
<td>3.08</td>
<td>1.78</td>
<td>9.57</td>
</tr>
</tbody>
</table>

Table 6.3: Latency by the numbers [ms] including the 50th and 95th percentile values
6.2. Synthetic experiments

(a) Bandwidth in bytes per second

(b) CPU load (user time) during the downloads. Each line represent a single instance, there are 1000 instances in total

Figure 6.3: Download traces for a 1-hop, 2-hop and 3-hop circuit

Figure 6.4: Latency probability density versus the circuit length, including the mean values as vertical lines.
Future work

At the moment of writing we acknowledge the following weaknesses in our work.

**We need a reputation system**  Our circuit building algorithm is vulnerable. Since there is no trust or reputation system at this stage the sybil attack is easily executed. The originator can only select from the list of hop candidates received from the last node. The last node could simply make up these candidates and gain control of all following hops. As the requirements stated BarterCast [28] should provide us with a distributed reputation system that checks the identity of the hop candidate and its contributions to the system. Reputation systems (like BarterCast) can be effective against Sybil Attacks [16]. Additionally it should be used as an incentive system to promote relaying for others.

**We need end-to-end cryptography**  Having exit nodes leaves us with another vulnerability. A malicious exit node can provoke a response from the originator by manipulating the packets it returns. It can drop packets, modify or returning specially crafted BitTorrent or libswift packets to the originator. The attacker can use this information to correlate a sender to a receiver [14].

A specific example of such an attack is the PeX attack. External peers or exit nodes can perform a Send n' Seek [5] attack by PeX’ing a colluding peer with the originator. The current implementation will probably connect to this peer over a different circuit, creating ‘multiple ways to Rome’ pointing towards the originator. This is particularly useful to a predecessor attacker (section 2.6.2).

Disabling PeX is the easiest way to prevent this specific attack. Another measure is that swarm peers found using PeX should be contacted over the same circuit as they were found. However this is difficult as it requires application level knowledge at the anonymous transport layer.

Even with the improvements above exit nodes will be the weak link of our solution. Therefore we should aim for end-to-end encryption with anonymous seeding. In this scenario all communications remain inside the anonymous network. Preliminary work on anonymous seeding has been started and will be implemented during the next year.
7. Future work

**Figure 7.1:** Multi-tunnel versus Multi-path routing from originator A to external services B using 1-hop circuits.

(a) Single tunnel  
(b) Multi-tunnel routing

(c) Multi-path routing  
(d) Combined multi-tunnel and multi-path routing

**We need cover traffic** As there is no cover traffic it is easier to perform Traffic analysis attacks. Tarzan solves by routing traffic over *mimics*, circuits initially created to route cover traffic. In our design the exact same approach can be taken. The extend candidate list (section 4.3) can be used by the hops to limit the candidates to the mimics they have up and running. However as cover traffic lowers performance with up to **67%** [8] it has not been implemented so far.

**Multi-path routing** Multi-tunnel is supported but it is not possible to have multiple paths between the anonymous leecher and a peer in the BitTorrent swarm, for the difference between multi-tunnel and multi-path see fig. 7.1c. If we don’t want to modify libtorrent to support multiple download sessions from the same IP we have to simulate it at a higher level. A straightforward approach is to implement another network address translation step at the originator.
Conclusion

The focus of this thesis work is to *Provide anonymous high-definition video streaming to users of Tribler.* As basis we have used the widely used and reviewed TOR-protocol. However some adjustments have been made to workaround some *limitations* in TORs implementation.

The design relies heavily on Dispersy as peer discovery system and on UDP as transport protocol. With the new design it took 4 iterations before all functionality was implemented. The last offers libtorrent support and a cryptography stack as implemented by *Plak*. During the evaluation we have seen that peers in our network are able to discover others and create connections, *circuits*, between them. Our performance evaluation has shown that we achieved the 10 Mbit/s goal. And that the number of hops does have a slight effect on the achieved throughput. Since these tests were conducted without cryptography, and using homogeneously performing hops, we expect different results in practice. Latency tests show that the number of hops has more or less a linear effect on the round-trip time.

Like we have saw during the implementation: our system is currently not IO-bound but CPU-bound. This has to be improved in the next phase of this design, together with an improved security model to resolve issues as end-to-end cryptography and using a reputation system to defend ourselves against the sybil and eclipse attacks.
### Message byte format

<table>
<thead>
<tr>
<th>Circuit ID</th>
<th>Host length</th>
<th>Host port</th>
<th>Origin length</th>
<th>Origin port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dest. hostname ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Origin hostname ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 5: DATA message byte format

<table>
<thead>
<tr>
<th>Circuit ID</th>
<th>6</th>
<th>Circuit ID</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) PING message byte format</td>
<td>(b) PING message byte format</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 6: Byte formats for PING and PONG messages
Experiment scenario

@0:0 start_session

@0:5 peertype 0_hop {1}
@0:5 peertype 1_hop {2}
@0:5 peertype 2_hop {3}
@0:5 peertype 3_hop {4}
@0:5 peertype others {5-1000}

@0:6 init_experiment 0 {1}
@0:6 init_experiment 1 {2}
@0:6 init_experiment 2 {3}
@0:6 init_experiment 3 {4}
@0:6 init_experiment 1 {5-335}
@0:6 init_experiment 2 {336-668}
@0:6 init_experiment 3 {669-1000}

@0:50 online
@0:60 build_circuits 3

@0:65 annotate start-0-hop
@0:65 start_download {1}

@0:120 annotate start-experiment
@0:120 do_latency_test 100 {2-1000}

@0:140 start_download {2}
@0:200 start_download {3}
@0:260 start_download {4}

@0:320 stop
adversary  See attacker. 7

attacker  A set of entities working against the anonymity provided by an anonymous network. 3, 7, 49

BarterCast  Protocol to exchange the altruism levels of peers. Altruism is defined as the amount of uploading versus downloading of a person. The goal is to create incentives for cooperation. 15

BitTorrent  A set of peer to peer filesharing protocols. Originally TCP streams were supported, later uTP introduced an UDP based protocol. See http://www.bittorrent.org/beps/bep_0000.html for a list of protocols. 9, 10, 32, 41, 49, 50

Chaumian mix  A relay that queues up a batch of messages before they are forwarded in a mixed order. This makes it harder for a global eavesdropper to determine who is talking with whom. 4

circuit  A sequence of hops, or relays, where data is forwarded over and back. 5

circuit id  Unique identifier of a circuit between initiator A and acceptor B. The identifier is a tuple in the form of \( \{IP_A, n_A\} \). Where \( n_A \) is a number selected by A specifically for this circuit. 19

DAS-4  The Distributed ASCI Supercomputer 4. 35–37

DHT  Distributed Hash Table. Decentralised alternative to BitTorrent trackers. 32

Dispersy  Distributed data dissemination framework. It is used in our system for peer discovery. For more information on Dispersy see [29]. 15, 21, 25, 49

Dispersy community  a community of Dispersy members with set message protocol, authentication, permission schemes and data synchronisation policies. 15, 21, 26

download engine  Components in Tribler which are responsible for actually downloading file from peer-to-peer networks like BitTorrent and libswift. 13, 21

eclipse attack  An attack where malicious node isolate a particular entity such that it can only communicate with malicious nodes. 7

entry node  The first hop in a circuit. It talks with the originator but has no way of knowing this. Neither does it know the ultimate destination of the messages it forwards. 10
exit node The final hop in a circuit that interfaces between the anonymous network and the open internet. It can read, and can be held responsible for, the data send by the originator and returned by the service. 5, 7, 10, 33, 41

Gumby Experiment runner framework for Dispersy and Tribler. See https://github.com/Tribler/gumby. 35, 36

I2P The Invisible Internet Project. 5

libswift UDP-based P2P file-sharing protocol developed at the PDS department at the TU Delft.. ix, 10, 13, 15, 22, 28, 30, 32, 35, 41, 49

libtorrent A library implementing peer-to-peer protocols defined in the BitTorrent standards. 13, 15, 22, 32, 42

NAT-box Network Address Translation box, often part of a consumer router. Used to share a single external address with the whole LAN. 50

NAT-traversal Traversing NAT-boxes such that peers which normally are not open for connections can accept incoming connections. Commonly a third peer helps to setup a connection in peer to peer systems. For more information see [29]. 13, 16

onion encryption Layered encryption scheme that limits the known information to the source and destination hop of the message. The payload is not readable by intermediary hop, but only by the destined hop. 5, 6

P2P-friendly NAT A NAT that can be traversed by the means of a third peer. For instance if A and B are connected and A has a P2P-friendly NAT, then B can help C traverse A's NAT box. 10, 14

PDS Parallel Distributed Systems. 15, 16, 50

peer discovery The act of discovering other peers in the same network. Most implementations rely on sharing discovered peers with others. 16

Peer-EXchange protocol A distributed peer discovery protocol extension to BitTorrent. Peers share the leechers and seeders they know with others. Thereby the swarm reliance of a central tracker is limited. 41

PeerID BitTorrent peer identifier of 20 bytes. Contains a random identifier, usually some bytes are reserved to identify the application and its version. 32

PeX Peer-EXchange protocol. 33

ProxyCommunity The dispersy community containing that proxies join to discover other proxies on the internet. 21, 22, 26, 32
**public-key cryptography**  Instead of a single key as in *symmetric cryptography* encryption uses a pair of keys: A public and a private key. Data encrypted with one key can only be decrypted with the other. The public key is assumed to be known by everyone, while the private key should never be shared. 4

**sybil attack**  An attack where a single entity presents multiple identities. Therefore it will look like multiple entities to the outside world. An example of such an attack is where the first hop of a circuit forges the identities of the following hops. Which compromises all anonymity. 5, 7, 15, 41

**symmetric cryptography**  A single key is used for both encryption and decryption. An example of a symmetric encryption algorithm is AES. 51
References


