NTNU – Trondheim
Norwegian University of
Science and Technology

Performance of hidden services in Tor

Bram Bezem
Petter Solberg

Master of Science in Communication Technology
Submission date: July 2013
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Performance of hidden services in Tor

Student: Bram Bezem & Petter Solberg

Problem description:

Tor is free software and an open network that helps you defend against traffic analysis. Tor makes it possible for users to hide their locations. Tor users can connect to hidden services, each without knowing the other's network identity. This functionality could allow Tor users to set up a website where people publish material without worrying about censorship.

The goal of the thesis is to learn how the hidden service protocol works, to successfully configure hidden services and to measure the performance of the in-bound/outbound traffic of the established environment. Hidden services are to be deployed on both the public Tor-network and on a self-configured private network consisting of several dedicated servers.

The students will look at the performance of different types of applications which can be run as a Tor hidden service. The performance will be measured using different parameters. For example initial access time, connection latency, bandwidth/throughput, reliability and other possible options. In addition they will look at the performance of hidden services compared with running the same application through the Tor network or other methods.

If the equipment allows, the experiment can be done in a high-bandwidth local area network environment in order to minimize the influence of other traffic and factor out network bandwidth as a limiting factor.

Responsible professor: Danilo Gligoroski, ITEM

Supervisor:

Abstract

In this thesis we will provide an overview of the Tor hidden service protocol. We will demonstrate the steps of setting up Tor hidden services and a private Tor network for testing purposes. The performance of the hidden services will be measured with regards to throughput, initial access time, connection latency, and reliability. The results show that hidden services on the public network can be used for web servers and instant messaging. On a private network, voice over IP and video streaming can be used as well. The hidden services are currently limited by the public network performance of Tor, but we obtained significantly higher performance on a private Tor network.

Sammendrag

I denne oppgaven vil vi beskrive protokollen for skjulte tjenester i Tor nettverket. Vi vil vise hvordan en kan settes opp en skjult tjeneste og sette opp et privat Tor nettverk som skal brukes til testing. Ytelsen til skjulte tjenester vil bli målt etter flere kriterier som båndbredde, tilkoblingstid, forsinkelse på eksisterende tilkoblinger og pålitelighet. Resultatene viser at skjulte tjenester kan bli brukt til webservere og chatteprogrammer på det offentlige Tor nettverket. På et privat Tor nettverk kan også IP-telefoni og videostrømming brukes. Vi vil vise at skjulte tjenesterer er begrenset av ytelsen i det offentlige Tor nettverket, men vi oppnådde betydelig høyere ytelse i et privat nettverk.

Preface

We present this thesis as the finalization of our 5-year MSc in Communication Technology with specialization in Information Security. The work has been carried out in Trondheim during the spring of 2013 at the Department of Telematics, Norwegian University of Science and Technology.

We would like to thank our supervisor Danilo Gligoroski for his guidance and assistance. Thanks to Pål Sturla Sæther for providing us with the necessary network and server equipment. We thank Roger Dingledine, Nick Mathewson and Paul Syverson for designing the Tor anonymity network, and all the volunteers for running Tor routers.

Regards,
Bram Bezem & Petter Solberg

Contents

List of Figures **ix**

List of Tables **xi**

List of Acronyms **xiii**

1 Introduction **1**
1.1 Motivation and goals . . . 1
1.2 Structure . . . 2
1.3 Methodology . . . 2
1.4 Tools . . . 2

2 Theory **3**
2.1 Tor . . . 3
2.1.1 Technical details . . . 4
2.1.2 Hidden services . . . 5
2.1.3 Hidden services protocol . . . 6
2.1.4 Client authorization . . . 11
2.1.5 Stealth authorization . . . 12

3 Setup **13**
3.1 Hardware . . . 13
3.1.1 Servers . . . 13
3.1.2 Network . . . 14
3.2 Software . . . 15
3.2.1 Operating systems . . . 15
3.2.2 Tor . . . 15
3.2.3 Deployment . . . 16
3.2.4 Git . . . 16
3.2.5 Administration . . . 16
3.2.6 Munin . . . 17
3.2.7 Jmeter . . . 17
3.2.8 Apache . . . 18

3.2.9 Logging scripts . . . 18
3.2.10 Log analysis . . . 19
3.2.11 RTTpinger . . . 19
3.2.12 Private Tor network . . . 20
3.3 Differences between private and public Tor network . . . 22
3.3.1 Tor configuration . . . 22
3.3.2 Network composition . . . 23
3.3.3 Usage differences . . . 23

4 Measurements **25**
4.1 Network performance . . . 25
4.2 Test length . . . 28
4.3 Hidden service throughput test . . . 31
4.4 Guard test . . . 34
4.5 Tor bandwidth test . . . 37
4.6 Access time test . . . 39
4.7 Connection latency test . . . 42
4.8 Reliability . . . 47

5 Discussion **49**
5.1 Increasing the number of guard nodes . . . 49
5.2 Application viability . . . 50
5.2.1 Web-server . . . 50
5.2.2 Instant messaging . . . 51
5.2.3 Voice over IP . . . 51
5.2.4 Video streaming . . . 51

6 Conclusion **53**

7 Future work **55**

References **57**

Appendices

A Results **59**
A.1 Tor bandwidth in the public network . . . 59
A.2 Test length in the private network . . . 59

List of Figures

2.1 Tor circuit creation 4
2.2 Hidden services protocol outline 7
2.3 Determination of responsible directories 9

3.1 Network topology 15

4.1 Hidden service throughput 29
4.2 Bandwidth variation over time 30
4.3 Measured throughput for changing number of clients 32
4.4 Public network throughput 33
4.5 Throughput with different guard configurations 35
4.6 Throughput of public network without guard nodes 36
4.7 Tor performance without hidden servicesin private network 38
4.8 Private network individual client delay variation 44
4.9 Private network delay variation for changing load 45
4.10 Public network delay variation 46

A.1 Public Tor throughput without hidden service 59
A.2 Private hidden service throughput over time 60
A.3 A graph of eth0 throughput on the hidden service node 61

List of Tables

3.1 *HP 3031h* specifications 13
3.2 *HP 304Ah* specifications 14

4.1 Public throughput using Tor without hidden services 38
4.2 Access time for the private network 41
4.3 Access time for the public network 41
4.4 Averange time of connection steps to hidden service in public network . 42
4.5 Reliability results 48

5.1 Probability of choosing at least one malicious guard node 50

List of Acronyms

AES	Advanced Encryption Standard
AES-NI	AES New Instructions
CET	Central European Time
CK	Client Key
C&C	Command and control
CPU	Central processing unit
CTR	Counter mode of operation
DC	Descriptor cookie
DH	Diffie-Hellman
DHT	Distributed Hash Table
DIR	Directory
DNS	Domain Name System
ESK	Encrypted session key
Gb	Gigabit
GbE	Gigabit Ethernet
GRO	Generic Receive Offload
GSO	Generic Segmentation Offload
HP	Hewlett-Packard
HTTP	HyperText Transfer Protocol
IP	Internet Protocol

IP Introduction Point

ITEM Institute of Telematics

ITU International Telecommunication Union

IV Initialization Vector

Mb Megabit

MB Megabyte

MSc Master of Science

MSS Maximum segment size

MTU Maximum transmission unit

NTNU Norwegian University of Science and Technology

OR Onion Router

RMI Remote method invocation

RP Rendesvouz Point

RTT Round-trip delay time

SK Session key

SHA1 Secure Hash Algorithm 1

SOCKS Socket Secure

SSH Secure Shell

TCP Transmission Control Protocol

TLS Transport Layer Security

TSO TCP segmentation offload

URL Uniform Resource Locator

UTF Unicode transformation format

VoIP Voice over Internet Protocol

VPN Virtual private network

Chapter 1
Introduction

1.1 Motivation and goals

Anonymity on the internet is an important concept regularly debated in modern society. Free speech, free press and freedom from oppression are a few human rights that can be strengthened by anonymity. Yet, governments conduct large surveillance operations on their own citizens[1], they might even disconnect entire countries from the internet in order to silence political opponents.[2] Anonymity can allow political opponents to express their views without fear of oppression; it can keep the identity of whistle-blowers safe when they make the public aware of surveillance, so that a government can be held accountable.

Several tools exist to provide such levels of anonymity. Examples include i2p[3], Tor[4], and countless VPN-based solutions. They all have one thing in common; they mask the origin of traffic by having servers pass on traffic on behalf of others.

For this thesis we wish to learn how the Tor hidden service protocol works, for example how an anonymous connection can be set up between a hidden server and an anonymous client. We want to know how this anonymity for both parties is provided and how anyone can participate as either a client or server. Additionally we will set up our own hidden services on both the public Tor network, and configure a complete private Tor network. We will then measure the performance of hidden services. Most of the performance analyses of Tor have been performed on the current public network, while we get a chance to see how Tor can perform with when nodes are connected through high-bandwidth links like Gigabit Ethernet. With steadily increasing network speeds, we can get a glimpse of how Tor will perform in the future,

[1] http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html

[2] http://www.bbc.co.uk/news/world-middle-east-22446041

[3] http://www.i2p2.de/

[4] https://www.torproject.org/

when residential connections might be significantly faster than today, thus allowing anyone to operate a high-capacity relay.

1.2 Structure

Following this introduction, we will provide the theoretical background necessary for this thesis by introducing the reader to the principles behind Tor and the hidden service protocol. We will then present a detailed overview of the hardware and software used for this thesis together with an overview of how the resources were set up and configured. Afterwards, we present the various test methods, the results from those tests, and discuss their significance. Then, we will formulate a conclusion, answering the questions from our project description. Finally, we will present any open challenges which others might choose to undertake to further explore Tor and its hidden services. In addition to this report, we have created a digital attachment containing the results together with self-made scripts and software.

1.3 Methodology

We measure the performance of hidden services using different criteria. The criteria include bandwidth of a hidden service connection, how long setting up a connection takes, what the delay over an active connection is, and how reliable a connection is. Based on the resulting performance characteristics, we will attempt to look at which services are possible as a hidden service. The tests will be performed on both the public network and the private network.

1.4 Tools

We will use a variety of tools to reach our goals. Some are hardware resources and consist of computers and switches to setup our private Tor network, and which will be used as hidden servers in the public network. On the software side we will use the Ubuntu linux distribution together with the Tor anonymity network software, Apache HTTP server, JMeter testing software, Git source code management tool, and various other applications.

Chapter 2
Theory

2.1 Tor

Tor, formerly an acronym for The Onion Router, is a network of virtual tunnels used to hide the origin and destination of TCP-traffic on the internet. The goal is to provide privacy by allowing users to hide their identity so that services like websites do not know who is using them. This can be utilized by regular persons to avoid tracking of their web-surfing habits, or by whistle-blowers who want to report illegal behavior. Tor allows users to avoid censorship by hiding the final destination and the type of traffic, thus potentially allowing access to blocked websites and circumventing other access policies.

Tor provides this anonymity through the principle of setting up encrypted paths called circuits through several nodes in a network of many. Tor acts as a SOCKS proxy which is a open RFC standard [LGL+96] and Tor can thus be used by any application supporting SOCKS. Data is transmitted in fixed size cells of 512 bytes and wrapped in several layers of encryption, like the layers of an onion. [DMS04] This is where the term The Onion Router originated. The network consists of many nodes, onion routers that pass on traffic to other nodes in the Tor network and sometimes to the outside. Nodes providing the latter are called exit nodes, since they allow traffic to exit the Tor network.

In figure 2.1 we show an omniscient view of Alice's connections. When Alice wishes to connect to Bob, she sets up an encrypted connection to a node (1), and uses that connection to connect to another node (2). Since the packets are seen as coming from the first node, the second node does not know the IP-address of the user. The connection to the second node is then used to establish a connection to an exit node (3), which then is oblivious with regard to both the user and the first node used. The user can then use the circuit it has established to connect to websites, here represented by Bob, or any other service supporting TCP. If an attacker controls at least two nodes in the network, there is a chance they are the first and the last

hop chosen of a circuit. Given enough circuits, this probability approaches 1. Using information about data volume and timing, they could find the identity of the user. Attacks using this technique have been published. [OS06] Therefore Tor was modified so the first hop is chosen from a set of guard nodes, these are selected to be the same for lengths of time. By using the same few guard nodes for a longer period of time this threat is mitigated. [WALS03] The default number of guard nodes is three. So when Alice decides to set up a connection to Dave, she will use the same node as before (5) as the first hop. However, she will choose new nodes at random for the next hops (6, 7) and ultimately reach Dave (8) through a different path.

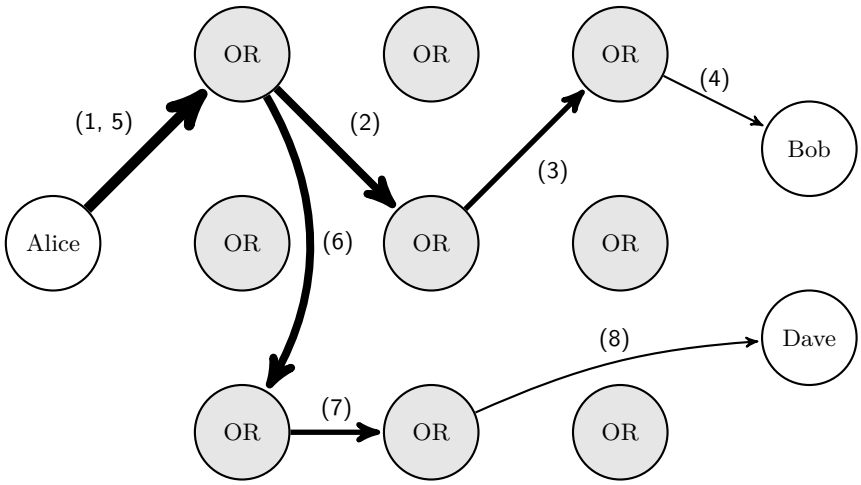

Figure 2.1: Outline of two Tor circuits set up by Alice, thicker lines indicate more layers of encryption.

2.1.1 Technical details

The explanation of Tor in the previous section did describe the outline of how Tor works. To be able to understand how hidden services works on top of Tor a better understanding of some key elements is needed. The current Tor is based on the original design paper [DMS04] but several changes have been made since the paper's release. The current Tor implementation is available on the official Tor homepage torproject.org [1].

Each router in the Tor network has a unique identifying RSA key. They keep their private key secret and publish their public keys to a few *authority* servers. The authorities control which routers are a part of the network by voting among each other in which routers are available and running. The result of the voting is

[1] https://gitweb.torproject.org/torspec.git/tree

called the *consensus*. It is a list of the valid routers and relevant information about them including, measured bandwidth, public key, IP-address and Tor version. Each router has attached *flags* which identifies the router's roles in the network. The consensus is signed by the authorities private keys and is later distributed to routers having the *Directory* flag. These directories deliver the consensus to the routers and to the connecting clients. The directories cannot alter the consensus because it is signed by the authorities. When clients want to use the Tor network, they use the information in the consensus to decide which routers to connect to. The algorithm to calculate which router to choose is complicated and depends on the routers flag and their bandwidth. In short: Greater bandwidth equals greater probability to be chosen. Clients and router creates *circuits* to routers by selecting a *path* through two middle routers. The connection between each successive router is secured using the well-known TLS (Transport Layer Security) protocol. In addition the client creates multiple layers of encryption along the path towards the destination router. Where each router along the way peels off a layer of encryption until finally the destination can read the plain text. This makes it impossible for the destination to be aware of who initiated the connection and impossible for the middle nodes and an eavesdropper to decrypt the contents. Inside a circuit there can be multiple *streams* which are an analogue to a TCP (Transmission Control Protocol) stream.

2.1.2 Hidden services

In addition to providing an anonymous way to access the regular internet services, Tor also contains methods to provide hidden services. Here not only the clients are anonymous but the server as well. Following is a simple description of the idea for the hidden services, based on the original idea and publication. The Tor specification has since been updated, but this is considered V0. A hidden service announces its public key along with several Introduction Points to the directory servers in the Tor network. Any client that knows the name, which is a 16 character sequence derived from the public key, can request the information about the service. The client will establish a Tor circuit to a randomly picked Tor relay which will act as a Rendezvous Point and hand over a single-use secret. Once the Rendezvous Point is ready, the client asks an Introduction Point to pass on an introduce message to the hidden service. The introduce message is encrypted with hidden services public key and contains the secret and information identifying the Rendezvous Point. It is decrypted by the hidden service, which then attempts to establish a Tor circuit to the Rendezvous Point. The secret allows the Rendezvous Point to verify the two parties which wish to communicate, and it will inform the client when a complete connection is established. Finally, there is now an anonymous connection between the client and the hidden service through the Rendezvous Point.

2.1.3 Hidden services protocol

The Tor hidden service uses the *Tor Rendezvous Specification* [2] to operate. There have been two revisions of the original specification and the current version is the third (V2) version. In this section we explain the current version using an example where Alice, as a client, wants to connect to a hidden service operated by Bob. Some unused and unessential fields are omitted from the description. For more details, see the specification. Figure 2.2 shows the outline of the protocol. The notation used for this section is as follows.

K_a	is a RSA public-private key used by a
$K_{a,b}$	is a RSA public-private key used by a to b
$P(K)$	is the public part of a RSA public-private key K
$S(K)$	is the private part of a RSA public-private key K
$H(x)$	is SHA1 [rJ01] digest of x
$a\|b$	is a concatenation of a and b
$E(K,x)$	is an AES encryption of x with the symmetrical key K
$D(K,x)$	is an AES decryption of x with the symmetrical key K

Creation of hidden service identity

Bob must first create a RSA [RSA78] public-private key-pair K_{Bob} to identify his hidden service. The URL to connect to the hidden service is created by base32 encoding [Jos06] the first 80 bits from a SHA1 hash of the public key and an additional suffix *.onion* to identify the URL as an hidden service. An example address is *rnuct7uhsi4ig77i.onion*. Then Bob can publish the url to clients who want to use his hidden service.

[2] https://gitweb.torproject.org/torspec.git/blob/HEAD:/rend-spec.txt

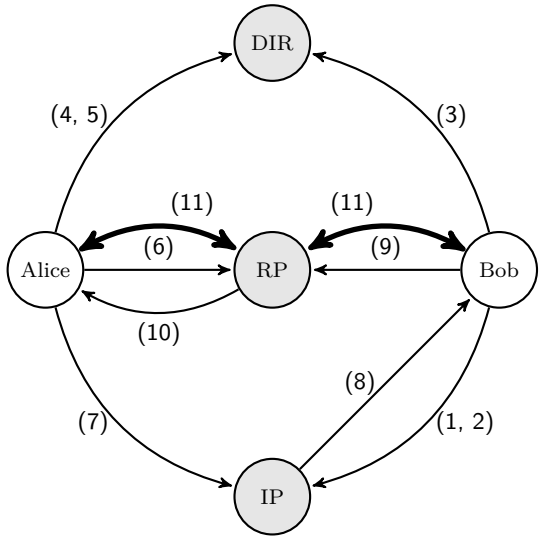

Figure 2.2: Outline of the hidden service protocol.

Publication of hidden service

1 . When Bob starts his hidden service, he creates circuits to between 1 and 10 Tor routers. These routers will be used as Introduction points (IP). Bob creates RSA key-pairs $K_{Bob,IPn}$ for each individual IP. Thereafter Bob sends a RSA signed message containing the corresponding public key $P(K_{Bob,IP})$ and session information between IP and Bob signed by $S(K_{Bob,IP})$.

2. The IPs accept the messages and send response messages back to Bob. After connecting to all IPs, Bob's client creates two *service descriptors*. A service descriptor is a collection of information about the hidden service and its introduction points which is published to the directory servers. These servers function like a DHT (Distributed Hash Table). The descriptor is a combination of many fields, which are described below. The first element *descriptor-id* is used as a key and the rest of the fields is the value.

descriptor-id = *H(permanent-id | H(time-period | descriptor-cookie | replica))*.
permanent-id is the first 80 bits of the SHA1 hash of $P(K_{Bob})$, *time-period* is a number derived from the current time and $P(K_{Bob})$ which changes every 24 hours, *descriptor-cookie* is an optional key if client authentication is added which makes the *descriptor-id* only derivable by authenticated users, and *replica* is an replication index which is used to distribute the descriptor to different directory servers.

permanent-key = Bob's public key $P(K_{Bob})$.
Is used to verify the *descriptor-id* and the *signature*.

secret-id-part = *H(time-period | descriptor-cookie | replica)*.
By using this element everyone can verify the descriptor belongs to the *descriptor-id*.

publication-time = Time this descriptor was created.
This is used by the directories to update old descriptors when newer are uploaded, and to discard out of date descriptors.

IP-list = List of IP information.
This list contains information about the IPs which are accepting incoming requests for Bob's hidden service. If client authentication is enabled the list of IPs is encrypted. Each IP part of the list includes:

introduction-point-id
The Tor router identifier for the IP.

ip-address-and-port
The ip-address and TCP port to the IP

onion-key
The public key for the IP

service-key = $K_{Bob,IP}$.
The public key for Bob's hidden service at this IP.

signature
This is a RSA signature of all the elements above except the *descriptor-id* using Bob's secret private key $S(K_{Bob})$. This element verifies that the descriptor is created by Bob.

3 . Bobs client publishes the service descriptor to the directory servers. Each hidden service has six responsible directory servers which store their descriptor. To calculate which directory servers the descriptors is to be published to. Bob filters the Tor consensus file to get a circular list of directory servers which have the *HSDIR* flag and sorts them by identity-ID. For each of the two descriptors, Bob looks up the *descriptor-id* in the circular list as shown in figure 2.3. The first three directory servers which have an identity-ID lexicographically following the *descriptor-id* is responsible for the descriptor. Bob creates a circuit to these directories and publishes the descriptor.

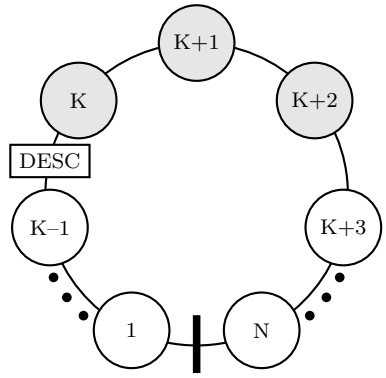

Figure 2.3: Determination of responsible directories. Circular lexicographically list of N directory list-ids. The three marked subsequent identity-IDs after the *descriptor-id* is responsible for the descriptor (DESC).

Alice connects to the hidden service

4. Alice receives the URL to Bob's hidden service outside Tor. She calculates the corresponding *descriptor-id* from the current time, url, descriptor-cookie and replica values. If authentication disabled the descriptor-cookie is omitted. Thereafter she calculates which directory servers are responsible for the hidden server the same way as Bob. Then she connects randomly to one of the responsible directory servers and asks for the descriptor by sending the *descriptor-id*.

5. The directory returns the most recently uploaded descriptor connected to the *descriptor-id* to Alice.

6. Alice creates a circuit to a randomly chosen Tor router now called a rendezvous point (RP). Then she sends a randomly generated *rendezvous-cookie* to the rendezvous point.

7. Alice creates a circuit to one of Bob's IP which she learned from the descriptor. Then she generates the start of a Diffie-Hellman key exchange [DH76] and sends a message to the IP. The message is divided into two parts. The first containing the **Service key identifier** = $H(P(K_{Bob,IP}))$ which identifies the hidden service. The second part is encrypted by the $P(K_{Bob,IP})$ and contains:

timestamp
: The current time the request is made.

RP-address-and-port
: The ip-address and TCP port to the RP.

RP-onion-key
The public key for the RP.

RP-id
The Tor router identifier for the RP

rendezvous-cookie
Identifies the circuit for the RP to Alice.

auth
Authentication information

DH-public-value $= A = g^a \ mod \ p$
Diffie-Hellman public value for Alice using her own secret a with the generator g and p from the second Oakley group of the Internet Key Exchange [HC98].

8. The IP receives the message from Alice and checks if the Service key identifier matches any known hidden services that it is an IP for. It matches Bob's key and the IP resends the second part to Bob and an acknowledge message back to Alice.

9. Bob decrypts the message from IP with the private service key $KS_{Bob,IP}$ and extract the information. Bob verifies the signature and if authentication is enabled it is also verified. If successful, Bob chooses his own Diffie-Hellman private value b and calculates his public value $B = g^b \ mod \ p$ and the shared secret $SS = A^b \ mod \ p$. From the shared secret Bob generates a handshake digest $KH = H(SS \mid [00])$, and the keys $Kf = H(SS \mid [01])$ and $Kb = H(SS \mid [02])$. Then Bob sends a message to RP containing the *rendezvous-cookie*, Bob's public value B and the handshake digest *KH*.

10. The RP receives the message and extracts the *rendezvous-cookie*. It matches the previous cookie received from Alice and the RP creates a link between the two circuits. RP forwards the rest of the message to Alice and from now on the RP forwards any messages between Bob and Alice.

11. Alice extract Bob's public value B and the handshake digest *KH'* form the message. She generates the shared secret $SS = B^a \ mod \ p$ and thereafter she verifies that Bob sent the message by checking if $KH' = H(SS \mid [00])$. If successful she generates the keys $Kf = H(SS \mid [01])$ and $Kb = H(SS \mid [02])$. Alice and Bob have now two shared secret keys *Kb* and *Kf*. *Kf* is used to encrypt/decrypt messages from Alice to Bob and *Kb* is used in the other direction. Alice and Bob have now created a circuit between each other through the RP. To access the hidden service, Alice creates a stream to bob and sends the Tor message *RELAY_COMMAND_BEGIN* with the special address "" and the hidden service port. The rest of the transport protocol is equal to normal Tor relay behavior.

2.1.4 Client authorization

In the example above Alice never authorizes herself to Bob. The hidden service protocol supports the ability for the hidden service operators to enable client authorization. The authorization can happen in two steps of the protocol: 1. When Alice downloads and decrypts parts of the hidden service descriptor. 2. At Bob's client before he connects to the RP. There are two implemented and specified authorization protocols:

Basic authorization

The basic authorization protocol is a protocol to authorise many clients and consumes few additional resources compared to no authentication. A hidden service operator can allow access for many clients while deny access for all others. The authorization is done by symmetrically encrypting the list of IPs in the hidden service descriptor with 128 bits AES in CTR mode. If the client manages to decrypt the *IP-list* then they are authenticated. This works because all clients need to know which IPs and corresponding service keys $P(K_{Bob,IP})$ to be able to connect to the hidden service. However, an unauthorised client or a formerly authorized client can monitor if the service is still available by only knowing the hidden service URL.

To accomplish the basic authentication, Bob generates a individual 128 bits *descriptor-cookie (DC)* for each individual client or group of clients. Then he transfers the cookies to his clients outside of Tor in addition to the hidden service URL. When Bob generates his hidden service descriptor, he generates a random session key SK and initialization vector IV which he uses to encrypt the *IP-list*. The IV is added to the descriptor. To make only those clients which got a valid DC to be able to derive the SK. Bob creates a authentication list containing a *client-ID* $= H(DC|IV)$ and encrypted session key $ESK = E(DC, SK)$ for each *descriptor-cookie*.

The *Tor Rendezvous Specification* does not specify if the *descriptor-cookie* for basic authentication is different from the *descriptor-cookie* used when generating the *descriptor-ID* in the descriptor. If they are the same, the hidden service needs to generate two descriptors for each client and therefore increase the directory resources significantly when there are many clients. We checked the Tor source code and it omits the *descriptor-cookie* field when generating *descriptor-ID* if basic authentication is enabled. Therefore only two descriptors are created when using basic authentication.

For a client (Alice) to authenticate themselves, she begins by fetching the descriptor from the directory server. Thereafter she generates her *client-ID* by using her DC and the IV from the descriptor. Alice looks for a match in the authentication list and retrieves the encrypted session key $SK = D(DC, ESK)$. With the SK Alice can decrypt the *IP-list* and is therefore able to connect to the hidden service.

2.1.5 Stealth authorization

The stealth authorization is more sophisticated compared to the basic authentication. In addition to authenticate the clients it also hides the hidden service for unauthorised clients or former authorised clients. it is computationally more expensive and is only used for limited number of users. The stealth authentication works by creating a descriptor designated for each client. Bob creates a RSA client key *CK* and *descriptor-cookie* for each client and transfers them to the clients outside Tor. When creating hidden service descriptor for each of the clients. Bob replaces the *permanent-key* with the *CK*. In addition he uses the clients *DC* when generating the *descriptor-id* and encrypting the *IP-list*. This makes it only possible for the client with the designated *DC* and *CK* to be able to calculate the changing *descriptor-id* and to be able to decrypt the *IP-list*. This makes the hidden service "stealth" because clients does not know about each other and cannot link other descriptors to the same hidden service. In addition to the changes to the descriptor, the clients include their *descriptor-cookie* in the encrypted *auth* field in the message to the IP and later relayed by the IP to Bob. Bob then looks if the *descriptor-cookie* is allowed to connect. If yes Bob continues with connecting to the RP.

Chapter 3
Setup

3.1 Hardware

3.1.1 Servers

For the thesis we acquired 19 computers of the type *Hewlett-Packard (HP) 3031h*. The computers are old workstations but have reasonably good specifications. The processor is from 2008 [1] which makes it over four years old. The computers are under our control and we are the only users. These computers are later used as Tor nodes in the private network. The relevant specifications for *HP 3031h* are listed in table 3.1.

Processor	Intel Core 2 Quad Q9400 2.66 GHz
RAM	4GB 400Mhz PC2-6400 CAS 6
Chipset	Intel Q45
Network adapter	Intel 82567LM-3

Table 3.1: *HP 3031h* specifications

We also acquired three computers of type *HP 304Ah* and were given access to 30 additional computers of the same type. The additional computers are shared with other students in a computer lab named *Sahara* which is administered by the Institute of Telematics (ITEM). We did not have physical access to verify hardware in these, but the system administrator confirmed they are the same model as the three we acquired. These computers are newer compared to the *HP 3031h* and have a CPU which was released in 2009 [2]. The first of the three computers is going to be

[1] http://ark.intel.com/products/35365/Intel-Core2-Quad-Processor-Q9400-6M-Cache-2_66-GHz-1333-MHz-FSB

[2] http://ark.intel.com/products/41316?wapkw=i7+860

used as hidden service provider for the private Tor network (HS Priv), the second as a hidden service provider for the public Tor network (HS Pub) and the third as a command and control (C&C) server which is used for remote controlling the other computers. The 30 computers that constitute the *Sahara* lab (Sah03-Sah32) are used as Tor clients in the public network. The relevant specifications for *HP 304Ah* is listed in table 3.1.

Processor	Intel Core i7 860 2.80 GHz
RAM	4GB 666Mhz PC3-10700 CAS 9
Chipset	Intel Q57
Network adapter	Intel 82578V

Table 3.2: *HP 304Ah* specifications

3.1.2 Network

To be able to have a large as possible Tor private network where all nodes have Gigabit connection we needed a 24 ports Gigabit Ethernet (GbE) switch. The institute did not have any we could borrow. Therefore we acquired a brand new switch of type *HP 1410-24G* [3]. Some switches are not capable to fully utilize all ports at the same time, but according to the specifications, the *HP 1410-24G* is capable of a throughput of up to 35.7 million pps (packets per second) of packet size 64 bytes. This equals a total throughput of more than 18 Gigabit per second.

The topology of the network is shown in figure 3.1. We have connected all the computers which are part of the private Tor network to the gigabit switch. The switch is also connected to another switch administered by the ITEM institute. This second switch is connected to all the computers in the Sahara lab and to the *C&C* computer and to the computer running the public hidden service (HS Pub). We decided on this topology because we wanted to divide the public and the private networks while still have internet connection to all computers. A factor was also that the system administrator was not able to give us a dedicated gigabit internet connection to the Internet. Therefore all computers share a single gigabit uplink to the Internet.

[3] http://h10010.www1.hp.com/wwpc/no/no/sm/WF06b/12883-12883-4172267-4172280-4172280-4220258-4220265.html?dnr=1

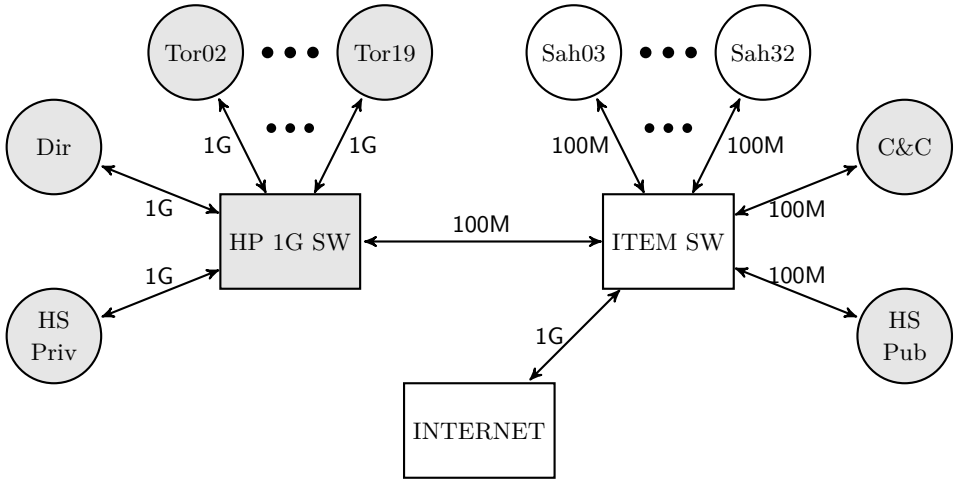

Figure 3.1: Overview of the network topology. Circles indicate computers, rectangles indicates switches. The numbers on the arrows indicate link speed. Grey color indicates the equipment is under our total control while the white color indicates the equipment is under the administration of the ITEM institute.

3.2 Software

In this thesis we used a lot of different software. In this chapter we list the most crucial software which we used and made during setup and testing.

3.2.1 Operating systems

We used the Ubuntu Linux distribution as operating system for the computers which we controlled. The specific version is *Ubuntu 12.10 x86_64* server edition. We chose Ubuntu over other operating systems supported by Tor because we are both familiar with Ubuntu, it is open-source, and it is free. Ubuntu has a lot of applications available and has great support for administering the servers remotely via Secure Shell (SSH). The computers outside our control are also running Ubuntu. The specific version is *Ubuntu 10.04.4 LTS i686* desktop. This is an older version of Ubuntu and contains a graphical user interface.

3.2.2 Tor

There is several versions of Tor running in the public network. The most used is 0.2.3 followed by 0.2.2 and 0.2.4.[4] The 0.2.3 is the current stable version and 0.2.4 is an unstable alpha version under development while 0.2.2 is an older version. We decided

[4]https://metrics.torproject.org/versions.png

to use the most recent stable Tor version (0.2.3.25 -2012-11-19) in our tests. This is mainly because it is the version which is the most used and the unstable version has no major differences related to hidden services. We also decided to build Tor from source-code and not rely on already compiled versions from Ubuntu's packages. This gives us the opportunity to investigate the source-code which we are actually using and change it if needed.

3.2.3 Deployment

To be able to administer all servers in the private Tor network we wanted a practical and reliable way to reconfigure and run programs on all servers. To manually install all servers could take a long time and servers could be configured differently. Since all the machines is equal we decided to install Ubuntu and other administrating applications on a single machine and later clone the disc and replicate the disk to all servers using the open source cloning utility clonezilla [5]. Since all the servers cannot be 100% equal we wrote a script which set the host-name of the computer and we made DNS addresses for each servers. This made it easy to differentiate between the servers.

3.2.4 Git

To ease the administration of the servers and have the ability to roll back configurations we used Git [6]. Git is an open source distributed version control system originally intended for source code management. We used Git as a way to distribute configurations, scripts, results and binaries between all servers. Git gives us the ability to see file history and who changed the file and why. This made it easy to update scripts and revert changes. Git also stores files and history for the files on all servers as long they are connected to the same repository. Therefore we connected other computers to the same repository which were in other locations. This makes a good way to protect from data-loss in case something goes wrong with all the servers e.g. fire damage or theft.

3.2.5 Administration

To administer the servers we configured a server to be a command and control server. This server where mainly used to monitor the other servers and to create configuration files and to execute commands to the other servers. As a example the procedure for updating Tor configurations for the private network explained in the following steps:

[5] http://clonezilla.org/
[6] http://git-scm.com/

1. (C&C) Change the script which generates the Tor configuration files.
2. (C&C) Run the script which generates the Tor configurations.
3. (C&C) Run git commit and git push to make the changes available to the other servers.
4. (C&C) Run a script which commands the private Tor servers pull the updated files in git. Thereafter make them run a Tor installation script. This is done by connecting to the servers via SSH with secure authentication with an RSA key pair.
5. (Tor servers) Pull the git repository and executes the tor-install script.
6. (Tor servers) The tor-install script stops Tor and deletes all Tor folders. Then it creates the directories again and copies the configuration files for the specific server identified by their hostname.
7. (Tor servers) Start a script to compile Tor from source code.
8. (Tor servers) Start Tor with new configuration.

This is only one example of the commands the command & control where commanding. The scripts used are available in digital attachment.

3.2.6 Munin

Munin [7] is an open source program which monitors the resource usage of one or more machines. It can create daily, weekly, monthly, yearly and custom time graphs of many different resources. We used munin version 2.0.2 to log and graph different resources on the servers including network throughput, cpu usage, memory usage and load average. This was accomplished by installing munin-node on all servers under our control and munin on the C&C server. We also created configuration files for aggregating the servers together to make additional graphs for easier analysis. The monitoring works by the munin process on the C&C probes all the munin-node instances on each server every five minutes and saves the results to disk. Later the statistics can be viewed. In our case the statistics were available through a website configured on the C&C server.

3.2.7 Jmeter

Jmeter [8] is an open source Java application designed to measure performance of web applications and other services. It supports running on multiple machines

[7] http://munin-monitoring.org/
[8] http://jmeter.apache.org/

simultaneously and is controlled by a single machine over RMI (Java remote method invocation). We used Jmeter to generate traffic by repeatedly sending HTTP requests to the server we are testing. The requests have configurable delay and Jmeter allowed us to view the statistics from the requests. Jmeter supports making request through a proxy server, but we found Jmeter could not connect to hidden services because of Jmeter only supported HTTP proxies. Therefore we modified the Jmeter source code and compiled our own Jmeter which made it support SOCKS proxies. The modified source code is available as part of the digital attachment.

3.2.8 Apache

To serve a HTTP service as hidden service we used the open source Apache HTTP server version 2.2.22 [9]. Apache is one of the most used web server applications used on the Internet and supports a variety of configuration options. With Apache we created a HTTP service containing multiple files. We generated a small file containing only the server's hostname and larger files of different sizes. With files of different sizes we could measure access time by downloading a small file and throughput by downloading a larger file. To generate the files we read data in blocks of 1 MB from `/dev/urandom` to provide us with pseudo-random data. The advantage of pseudo-random data over other data is that is much less compressible due to the lack of patterns. Should there be any compression mechanism that could influence bandwidth measurement, its impact is now minimized.

3.2.9 Logging scripts

We created several logging scripts which we used on all private network tests. They log information like CPU utilization of Tor, and general network usage on both the eth0 and lo interfaces. The latter is the loopback interface, which is used for traffic between processes on the same computer. Examples include traffic between Tor and Apache when using hidden services. We chose to use these scripts as an addition to Munin, they have a higher sample rate of 1 per second, plus these scripts would only generate logs during tests. We discovered Munin also downsamples the results as they get older so that it only stores data for 15 minute or even 1 hour intervals for data older than 2 days or 1 week respectively. Finally, Munin is dependent on a remote C&C node, which is less reliable when network links are fully utilized.

We created three different bash scripts that each logged CPU, eth0 bytes transferred or lo bytes transferred. Each runs a continuous while loop as long as a specific file exits, this allows us to stop the logging gracefully. Every second they log the time in nanoseconds by using the `date +%s%N` command, followed by the data to be logged. When logging CPU, we used output from the command `top`. We attempted to use the

[9] http://httpd.apache.org/

`/proc/[pid]/net/netstat`, where `[pid]` is the process-id of the Tor process. This pseudo-file holds a lot of different information about number of packets transmitted and received, and two variables; InOctets and OutOctets. As their names suggest this should be equal to the number of bytes received and transmitter respectively. However, after several test, we discovered that this pseudo-file does not as one would believe. `/proc/[pid]/` normally holds information only for the process with that process id, but we discovered that the particular netstat information was not unique to this process. It appears to be the total for all process and both interfaces eth0 and lo. When we cannot find the information about a particular process, then we would rather separate between eth0 and lo. Thus we switched to using the output from `ifconfig eth0` and `ifconfig lo`. Additionally we created scripts to automated starting and stopping these scripts on many nodes. Stopping includes retrieving all log files generated, compressing them and backing them up. For the complete logging scripts, see the digital attachment.

3.2.10 Log analysis

Our logging scripts described above generate quite a lot of data especially for the longer tests. We therefore discovered a need for something to parse and visualize the data. We created a Java application which does exactly that. It parses the different types of log files, using the JFreeChart[10] library, it can generate images with graphs of the data. It can also calculate different statistics including the minimum-, average-, maximum- values and the cumulative distribution function. The program iterates over a list of samples, keeps track of the number of samples, the minimum, the maximum, a sum of all values and a sum of all values squared. Then when it has analyzed all it can divide the sums by the number of samples to find the expected value (average) and the expected value for the square of the value. The variance equal to the sum of all values squared minus a sum of all values. A copy of the source code is in the digital attachment.

3.2.11 RTTpinger

To be able to measure round-trip time (RTT) we could not find an application which supported proxies and produced detailed results. Therefore we made our own Java application called RTTpinger. RTTpinger works by having multiple clients connecting to a single server through SOCKS proxies and records the RTT for each message sent. The client is configurable to how many messages and how much bandwidth the client sends to the server each second. The main concept for RTTpinger client is as follows.

– The client connects to the server and established a TCP socket.

[10] http://www.jfree.org/jfreechart/index.html

- The client calculates how large each message to the server by dividing the configured throughput with the configured number of packets per second.
- The client creates a separate message sending thread which schedules sending messages to the server at the configured interval.
- The message from the client to the server consists of "Ping!" followed by the current client system time in milliseconds, and finally random data to fill the message to the desired size.
- For each message response from the server the client records the current system time in milliseconds and extracts the time when the message was sent from the message. Then calculates the RTT by subtracting the sending time with time when the message was received. Then write the statistics to a log file containing the RTT and the time the message was sent.

The main concept for RTTpinger server is as follows:

- The server starts up by listening on the configured TCP port.
- For each incoming connection request, the server creates a new dedicated worker thread to handle the specific connection.
- For each incoming message to the server, the worker thread reads the whole message and replaces "PING!" with "PONG!" and sends the message back.

The complete source-code for RTTpinger is available in the digital attachment.

3.2.12 Private Tor network

To run a private Tor network some extra configuration is needed compared to running on the public Tor network. This section describes how to setup a private Tor network with authority servers that controls the network and how to convert Tor-routers and Tor-clients to use the newly created private Tor network.

The Tor network needs at least one authority server. These servers vote and obtains a consensus of which tor-routers is valid and holding the directory of hidden services among other tasks. The following steps explain how to create a single or multiple authority servers.

First step is to create a file directory for the authority server. The directory is written as the variable *$Dirpath*. Then generate a minimum Tor-configuration to make Tor be able to create a server identity key. Execute the commands below to accomplish this.

```
mkdir -p $Dirpath
echo "ORPort 9001" > $Dirpath/torrc
tor --list-fingerprint --DataDirectory $Dirpath/ -f $Dirpath/torrc
```

Second step is to fill the Tor configuration file with instructions to run Tor as a authority server. Change the nickname and contact info of your preference. The command below will make a Tor configuration which makes Tor run as an authority server.

```
echo "Nickname DirectoryServerN
 ContactInfo <Your email>
 DirPort 9030
 AuthoritativeDirectory 1
 V3AuthoritativeDirectory 1
 V2AuthoritativeDirectory 1
 DNSListenAddress 127.0.0.1
 DNSPort auto
 DataDirectory $Dirpath
 " >> $Dirpath/torrc
```

Third step is to generate keys and certificate for the directory. There are two types of keys. An *identity key* is a long-lived private key which is the authority server root private key. This key is used to sign short-lived signing keys and to create certificate for the signing key. The *signing key* is used to sign the authority's vote in the consensus. To generate these keys run the following commands. You will be prompted to enter a password to encrypt the identity key. Remember to choose a decent password.

```
cd $Dirpath/keys
tor-gencert --create-identity-key
```

Fourth step is to extract information from the directory. All other directories, Tor routers, and clients need this information to be able to find and trust the authority. Both the identity key fingerprint and the server fingerprint are needed. The first command below extracts the identity key fingerprint from the authority certificate. The second command gets the server fingerprint.

```
//Indentity fingerprint
```

```
cat $Dirpath/keys/authority_certificate | grep -i fingerprint | \
cut -d " " -f 2
// Server fingerprint
cat $Dirpath/fingerprint | cut -d " " -f 2
```

To setup more than one authority server, repeat all the previous steps in a different directory or on another server.

The fifth step is to create a sub-configuration which is to be added to all configurations on all Tor-directories, -routers and -clients to be able to use the private Tor network. The line *TestingTorNetwork 1* needs to be set to inform Tor to not use and trust the public authority servers, followed by a line specifying the trusted authority servers.

```
TestingTorNetwork 1
DirServer $Name1 v3ident=$IdKeyFingPrint1 $SrvIPAndPrt1 $SrvFingPrnt1
DirServer $Name2 v3ident=$IdKeyFingPrint2 $SrvIPAndPrt2 $SrvFingPrnt2
```

Where *$Name* is the servername, *$IdKeyFingPrint* is the Identity key fingerprint, *$SrvIPAndPrt* is the ip-address and port for the server e.g. 10.0.0.1:9030 and *$SrvFingPrnt* is the server fingerprint. This configuration must then be appended to all Tor configurations which are going to use the private network. Then start all Tor instances. The authorities should be in a consensus within 10 minutes which is the default vote period and then clients can get a valid list of Tor routers to establish connections.

3.3 Differences between private and public Tor network

3.3.1 Tor configuration

The Tor configuration for the private network is different from the public network. The standard settings for the public network are tuned for a big scale network with delay and possibly unsynchronized clocks. By setting the flag *TestingTorNetwork 1* as explained in section 3.2.12, Tor tunes the settings for a smaller network by accelerating the propagation of directory information. When the authorities are booted for the first time, they start voting at once compared to the public network where they wait 30 minutes. When the authorities first starts voting they vote every 5 minutes compared to the normal 60 minutes. The authorities also set the flag *Running* to new relays at once they know about them. Normal behaviour is to wait 30 minutes before setting the flag. The relays and clients also remove the waiting-time before they download the consensus from the directory servers. Normal behaviour is

to wait 10 minutes for the consensus to propagate from the authority servers to the directory servers. This reduces the time to start a new network and add new relays significantly. But there is additional cost of bandwidth and processing resources due to reduced voting interval and hence more fetch operations from the directory.

3.3.2 Network composition

In the private network we have a nearly homogeneous network compared to a heterogeneous public network. In our private network all servers run the same version of Tor and all relays have the same configuration, equal hardware, equal operating system and equal network resources. In contrast the public network is run by volunteers around the world. This means the Tor network consist of many different operation systems, Tor versions, Tor configurations and network configurations [11]. The size of the networks is also different. Our private network consists of 1 hidden service provider, 18 relay servers and 1 authority and directory server. The latter two are *HP 3031h* workstations while the hidden service provider is an *HP 304Ah* workstation. This network is much smaller than the current public network which has 3708 relay-, 10 authority- and 2249 directory-servers.[12]. Our servers have also much more bandwidth available compared to the public servers per server. The public network has currently about 4000 MB/s advertised bandwidth. Advertised bandwidth is the sum of the available incoming and outgoing traffic on all Tor Servers. This gives an average bandwidth of about 1 MB/s per server. This is in great contrast to our 250 MB/s per server.

3.3.3 Usage differences

Our private Tor network is under our control. We are the only who have control of the servers and we are the only users. There is also nobody else connected to the network switch and therefore we are the only one utilizing the network. This gives us total control of the server environment. This makes it possible to create and monitor system load and network throughput consistently without many unknown factors. The public network is quite different. It is not under our control and we have no way of controlling how much the network is used. With the current estimate of 500 000 active clients [13] makes it hard to predict load and throughput for the network. Since the public network is large. The impact of our testing is not dominating compared to the overall traffic. Current relay bandwidth for the public network is about 2500 MB/s. If we fully utilize a hidden service with a single 100Mb/s connection, the total traffic we create in the Tor network will be about 150MB/s. If we count inbound and

[11] https://metrics.torproject.org/network.html accessed 2013.06.11
[12] http://torstatus.blutmagie.de/ accessed 2013.06.11-22:13
[13] https://metrics.torproject.org/users.html accessed 2013.06.11

outbound traffic for the 6 nodes between the clients and the hidden service. This equals about 6% of the total traffic.

Chapter 4
Measurements

In this chapter we present the methods and result of all the tests we have executed. We also discuss the results of each test individually with regards to difference between public and private network together with our interpretation of their implications. Due to thesis length considerations only the relevant results are presented, the complete results are part of the digital attachment.

4.1 Network performance

Before we start to test the private Tor network we need to quantify what our gigabit network is capable to deliver. Without a baseline it is not possible to determine the loss in performance Tor creates. The switch and the servers could both be bottlenecks in various situations. This test will find how much throughput our equipment can handle. The hardware in use is the servers used in the Private Tor network hence the 19 *HP 3031h* computers and the *HP 1410-24G* gigabit switch.

Preparation

We researched several ways to generate traffic and tested different tools. And we found two good tools we finally found reliable and well performing for our purposes. We used *Iperf* for the generation of traffic, and *Munin* for monitoring the network throughput of many different servers simultaneously.

Using these tools we did a test-run to verify that the tools performed as foreseen and to learn how to collect the data. We set up a single iperf-server and an iperf-client on different servers. Using munin we observed a speed of above 900Mb/s in each direction. We discovered some irregularities after studying the number of packets. There were fewer packets than expected if the packets were of the size 1500 bytes per packet. This indicated that "jumbo frames" might be enabled. Jumbo frames are any frames larger than 1500 bytes, typically 9000 bytes as per Alteon Networks suggestion. [All]

Checking the MTU (Maximum Transmission Unit) on both computers stated an MTU of 1500 bytes and verified that jumbo frames were not enabled. To investigate further, we ran an packet capture at the server and the client using the tool *tshark* [1] during one of the tests. From the captured packets we observed a maximum packet size of 65226 bytes. After much research we found three settings on the network cards that could be causing this. These settings were Generic Segmentation Offload (GSO), TCP segmentation offload (TSO) and Generic Receive Offload (GRO).

GSO and TSO are two similar techniques to reduce CPU load for outbound network usage by making the applications believe the MSS is larger than it actually is. TSO is support for the network card to split segments on its own into the actual MSS. GSO is a technique to emulate TSO by doing the splitting in the lowest level in the Linux network stack. The result is less traversing of the network stack and therefore reducing CPU load [Xu]. GRO is a technique which is the reverse of GSO and TSO. It can assemble incoming packets together an therefore reduce the number of packets. This is done in the network card driver, or on the network card itself and reduces the CPU load.

We disabled GSO, TSO and GRO in many combinations at the server and client to verify the actual packets transferred had an MTU of 1500 regardless if any or all of GSO, TSO and GRO where enabled. We observed a lower throughput of 842Mb/s and significantly higher CPU usage when the three settings where disabled compared to leave them on.

Methods

To be able to compare and quantify the results of the performance tests. The data should be measured and collected as equally as possible. We chose a test length of 20 minutes for each test and each test is to be repeated three times. The mean speed of the traffic in Mb/s (megabits per second) is measured in the actual transmitted traffic. This means we are not measuring the payload traffic but also throughput from headers, retransmissions and other overhead sources. The traffic will be measured in both directions it the last 15 minutes of each test repetition. This is to reduce any transient effect in the start-up phase and to minimize measuring variance. To compare the results we will compare the bidirectional throughput per network link. To calculate the bidirectional throughput per port equation 4.1 is used. It is used to give numbers comparable numbers to the average Tor relay throughput. Because a Tor relay throughput is limited by the lowest of the outbound and inbound speeds.

$$T = \frac{\sum_{r=1}^{R} \sum_{l=1}^{L} min\left(I_{r,l}\,,\, O_{r,l}\right)}{R * L} \tag{4.1}$$

[1] Terminal version of Wireshark http://www.wireshark.org/

Where T = throughput per port, l = link number, r = repetition number, R = number of repetitions, L = number of links under test, I = inbound traffic, O = outbound traffic. This equation finds the average of the average minimum throughput per link for all the repetitions.

We have made three tests scenarios to test different aspects of the network performance.

Single server, single client This test looks at what a single server with only one connection in each direction is capable of. There is one server and one client. The client connects to the server and starts bidirectional traffic between them. The link between the server and the switch is to be measured. The settings for the server and clients are stated below.

```
SERVER:    iperf -s -p 33302
CLIENT:    iperf -c SERVERIP -d -t 1200 -i 30 -P 1 -p 33302
```

Single server, multiple clients This test is to mimic a single Tor relay. A server receives traffic from client1 - client9 and sends traffic to client10 - client 18. This should test if a single server can handle heavy utilized connections to multiple clients at the same time. Since the clients have more bandwidth than the server combined. The link connected to the server should be congested. This could make the switch drop packets and it therefore has an impact on the throughput. We will also see whether the switch can transfer packets from multiple sources into a single link and still maintain high throughput. The link between the server and the switch is to be measured and the configuration scripts for this test are part of the digital attachment.

Multiple servers, multiple clients This test is to mimic Tor relays on all used ports. All the physical servers run iperf-servers and iperf-clients that connect to all other servers. This gives a total of 18*19 = 342 active connections in total. This test should find the total throughput the switch can handle. All links is to be measured. See to the digital attachment for the complete scripts used to run this test.

Results and discussion

The throughput per port acquired by the tests is listed as follows:

- Single server, single client

  977 Mb/s

- Single server, multiple clients

  976 Mb/s

– Multiple servers, multiple clients

  427 Mb/s

From the results one can see that the throughput between a single client and multiple clients connected to a single server is almost identical. The throughput is close to the theoretical maximum of 1000Mb/s. This indicates the the server and the switch handles multiple sources and multiple destinations equal to a single source and destination with gigabit speeds and is capable of fully utilize the available bandwidth. On the other hand when running multiple servers and multiple clients the throughput is reduced to less than half to single server speeds. This indicates the switch is the bottle neck because we know from the previous tests the servers is capable of delivering higher throughput. The switch specifications indicate a higher throughput capabilities, but they don't specify how the traffic is divided between the ports and how they measured it. But with this test we confirmed that the switch is at least capable of an average of 427Mb/s throughput for all the Tor relays at the same time.

4.2 Test length

Method

The length of the tests is an important consideration, in order to look at the influence of running time on the bandwidth we decided to run several tests over a longer period of time. We use the HTTP server from Apache as hidden service serving a 100 MB large file containing pseudo-random data. The clients would repeatedly request the file during the test. Once a request completed it would continue without any delay.

For the private Tor network we chose to run three tests for as long as 12 hours to see how the bandwidth through the hidden service varies initially during the test and over longer periods. We used 18 clients (Tor02-Tor19) to connect to the hidden service during these tests and used our logging scripts to record the throughput.

On the public Tor network we do not have control over the load and usage of the network nodes during the tests. And since the traffic from our clients to our hidden service statistically will be spread all over the public Tor relays, our result will be highly dependent of usage and other factors beyond our control. We used 18 clients on the sahara computer lab to connect to the hidden service. We started four repetitions lasting several hours to see if the bandwidth changed significantly over time. Munin is used to record the throughput.

Result on the private Tor network

A small graph showing the variation in throughput during a test is shown in figure 4.1. For the full version please see figures A.2 and A.3 in the appendix, these show the throughput on the eth0 network device on the hidden service node, tor20, during the first and second iteration respectively. We can see that the bandwidth can vary between 14 and 33 MB/s over the course of a single test, which is quite significant. However for the 50 minutes of the test it is more stable, all three iterations show the same behavior of dropping off after this initial stable period. There is no significant ramp-up period since the speed increases rapidly once the test is started. For each of the three iterations we observed stable speeds for the first 50 minutes of 30, 28 and 36 MB/s respectively.

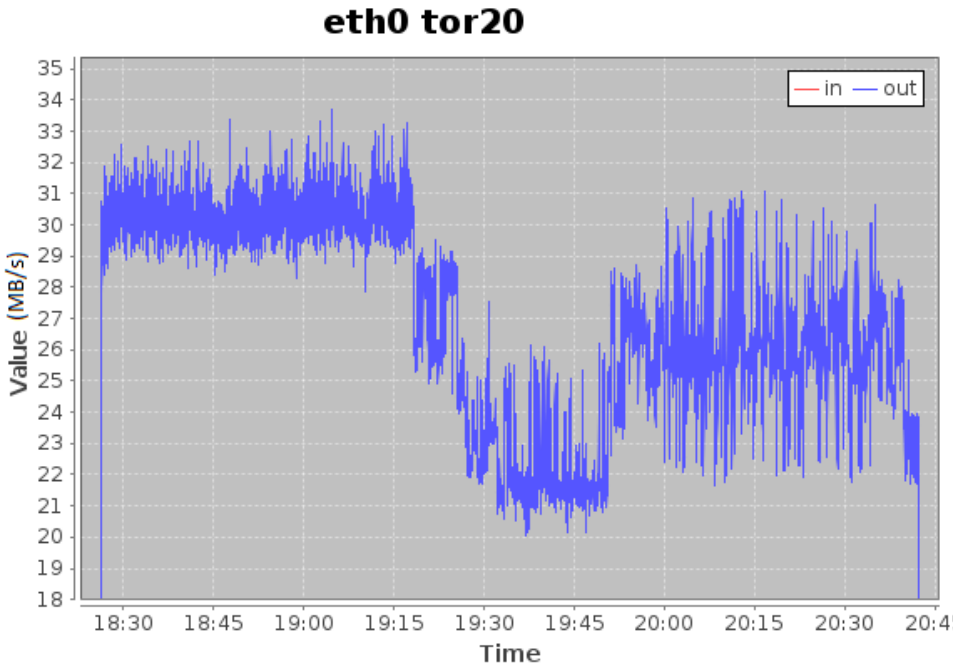

Figure 4.1: A graph of eth0 throughput on the hidden service node in the private Tor network

Result on the public Tor network

Figure 4.2 presents one sample of the test. All of the samples had different but similar results as figure 4.2.

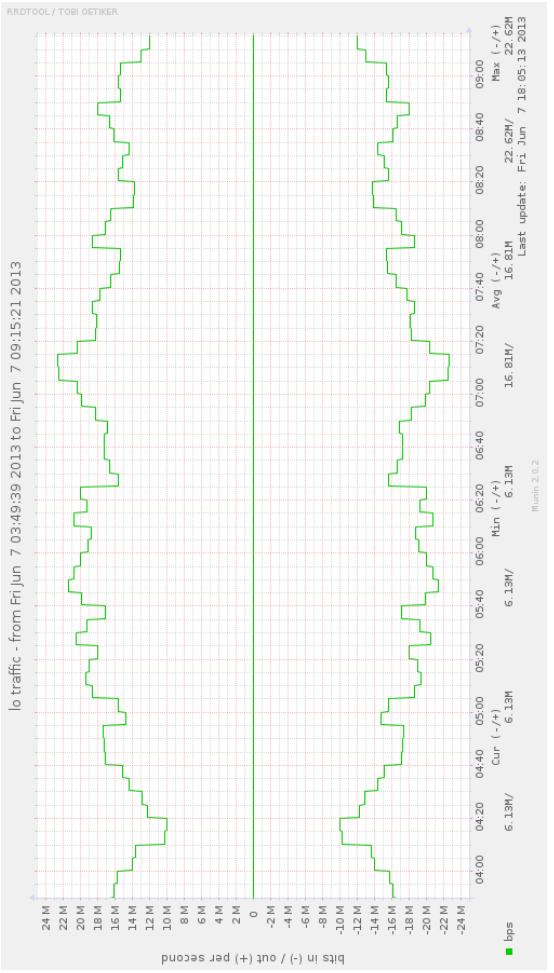

Figure 4.2: An example of bandwidth variation over time on the public network

Discussion

From the results from the private network we registered stable throughput in the beginning. If we set a test-length of 30-45 minutes the measurements in this period is reasonably stable. Typically variations within the period are ±5% from the average in these results. The initial average values vary between 28 and 36 MB/s, which is a 22% difference. This is a significant difference to the variations inside the proposed test-length. After the proposed test-length the throughputs begin to drop and vary a lot. By looking at the load of the relay servers we observed a change in the distribution of the load. For some servers the load increased, while other servers it declined. This could be an indication that some relays where chosen more often

when the clients decide on their path. Looking at the consensus we confirmed our belief. Some relays had order of magnitude greater bandwidth numbers compared to others, thus explaining the drop in speed. Therefore we conclude the throughput tests should be run in a time span of 30-45 minutes.

The throughput for the public network does not change significantly during short periods but with gradual changes for each step. Figure 4.2 presents a variation between 10 and 22 Mb/s and in the start of the test there is no significant difference from the average and no clear trends. Therefore the tests do not need to be limited to a specific time. On the other hand since Munin only samples every 5 minutes we want some samples to remove some of the variance. Therefore a test-length of minimum 30-45 minutes is sufficient.

4.3 Hidden service throughput test

Method

This test is to measure the hidden service throughput. We use the resulting test-lengths from the previous test. We used HTTP server from Apache as hidden service serving a 100 MB large file containing pseudo-random data. The clients would repeatedly request the file during the test. Once a request is completed it would continue without any delay.

For the private network we changed the number if clients to see if the number of clients have an impact of the throughput. Each of the relays ran ten Tor client instances in addition to the relay instance, but not all were utilized in every test. We changed the number of clients by making Jmeter execute a set number of simultaneous threads on a subset of the relay nodes. The threads would all continuously and repeatedly download the test file through a designated Tor client running on the same relay and the entire test would be run for at least 30 minutes.

The following client configurations were each tested three times:

- 1 client running 1 thread each
- 2 clients running 1 thread each
- 4 clients running 1 thread each
- 8 clients running 1 thread each
- 18 clients running 1 thread each
- 18 clients running 2 threads each
- 18 clients running 5 threads each
- 18 clients running 10 threads each

For the public test, we set up 18 sahara workstations and, just as in the private

counterpart, each runs 10 Tor clients to connect to the hidden service. We repeated the test 7 times, to attempt to mitigate other factors.

Results

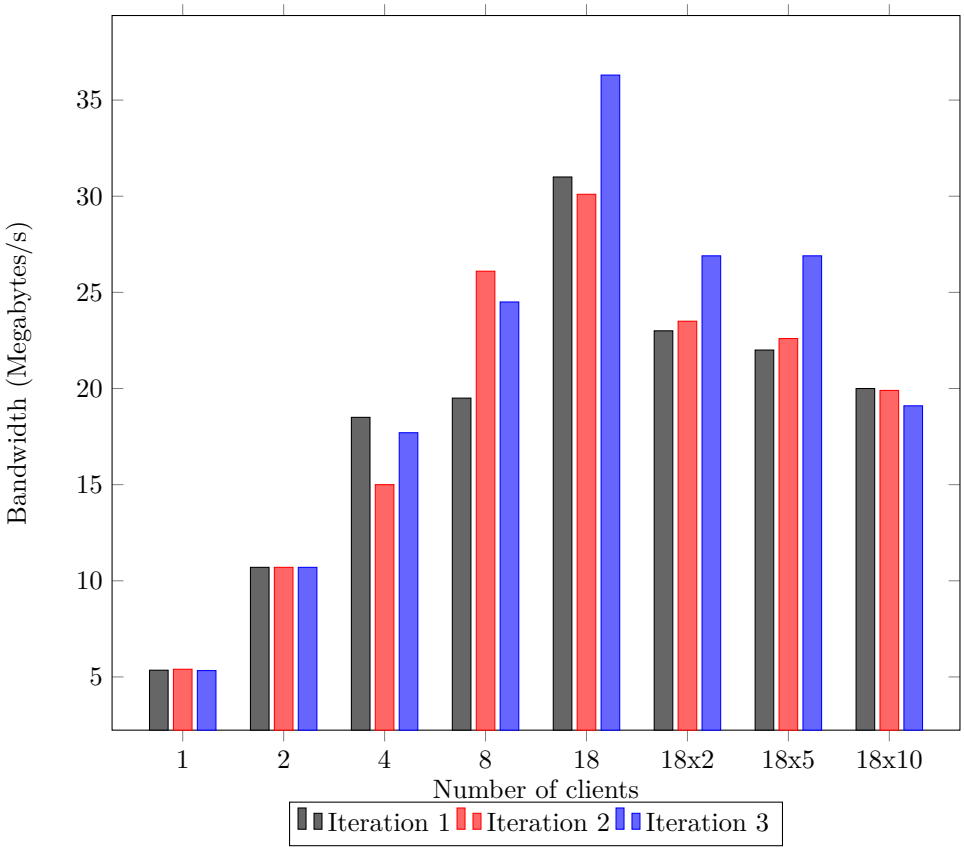

Figure 4.3: A bar plot of the measured throughput in the private network with changing number of clients

Figure 4.3 presents the throughput results for the various client configurations for the private hidden-service throughput test. Each coloured line represents a different iteration of the same test. It is important to note that any number of clients greater than 18 have been tested using multiple clients running on the same physical machine. The initial doubling from one to two clients lead to a doubling in throughput, subsequent doubling of the number of clients however, do not have the same result.

The highest throughput is achieved with 18 clients, and when increasing the number of clients on each machine a small performance decrease is observed.

Result on the public Tor network

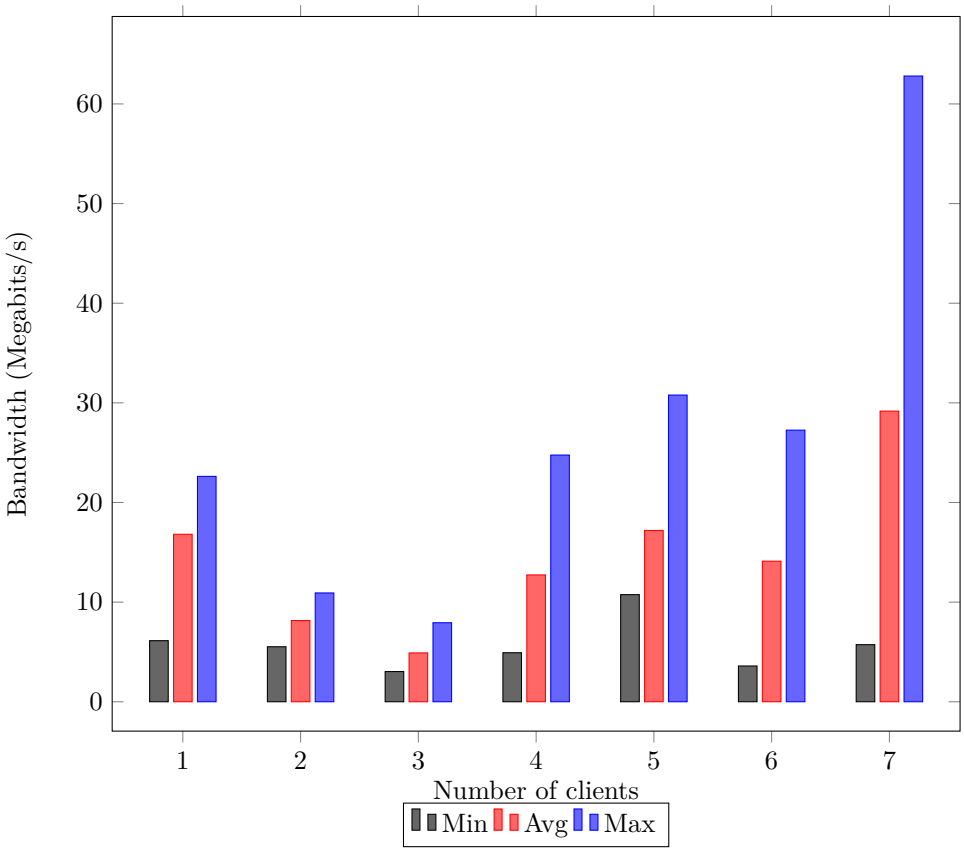

Figure 4.4: A bar plot of the measured throughput for the hidden service on the public network with 10 clients running on each of 18 sahara computers

Figure 4.4 shows the minimum, average and maximum bandwidth values for each of the different test on the public network. The average varies significantly between each test and the maximum is even more extreme. The largest maximum is almost eight times greater then the lowest maximum.

Discussion

As expected, the results for the public network are very varying. The tests where not run in parallel which makes the time of day and hence network load varying. The hidden service also changes guard nodes between each test. Since all the connections for the hidden service goes through these guard nodes the load on these will have great impact creating the varying results. This is why we chose not to test different configurations on the public network. It would be complicated to factor in such parameters and the number would therefore not be comparable.

The minimum bandwidth observed in the public Tor network is low, with a minimum over all results of 3.03 Mb/s. This is within one order of magnitude of typical residential connections, but this is shared among 30 clients. The median over all tests of the average bandwidth is 14.11 Mb/s which shared over 30 clients does not deliver very much capacity to any of those, unless the sharing is very disproportional. Even in that case the reasonable service of some clients is partially negated by the consequentially bad service for others. The private network shows that Tor itself is capable of a higher throughput, a single client manages 5.5 MB/s which equals over 40 Mb/s, but the public network is limited by other means. Based on this, one to three clients might use all available bandwidth from a single hidden service in the public network.

The tests on the private network give interesting information, we see a linear growth when increasing from one to two clients, but when further increasing to four, the increase is just 70% instead of the expected 100% increase (doubling). This gave us reason to believe there might be a limiting factor related to the number of clients, and from section 2.1 describing Tor we know there are three guard nodes. Thus we thought testing the number of guard nodes could provide us with relevant insight. Up to and including the test with 18 clients, the total throughput is increasing, but further increases in clients lead to less and less throughput with every increase. We do not believe that this will necessarily be true for a network with a much larger number of onion routers, like the current public Tor network.

4.4 Guard test

Method

Similar to the hidden service bandwidth test described in sections 4.3, this test will measure performance by looking at the bandwidth of the traffic leaving the server when it is accessed as a hidden service. This test was done with 18 clients each running a single thread, this is kept constant for this test. We look at how different settings concerning the guard nodes of the hidden server influence the measured bandwidth. This is done by setting the `UseEntryGuards` and `NumEntryGuards` in

the `torrc` configuration file for the hidden server. The following configurations were used:

- `UseEntryGuards` 1 and `NumEntryGuards` 1
- `UseEntryGuards` 1 and `NumEntryGuards` 3 (the default)
- `UseEntryGuards` 0

The default configuration was not repeated, but results from the hidden service bandwidth test will be used. The procedure for this test is the same as for the hidden service bandwidth test with 18 clients, and therefore a valid result.

Since the throughput on the public network varied significantly, we believe it would be difficult to identify whether a low result was caused by changing guard node settings, or because of other factors in the network. Therefore we only chose to disable the guard nodes, and to run the test for longer than a day. This would allow us to better analyse results with regards to change compared to the results with default guard node settings from the test in section 4.3.

Result on the private Tor network

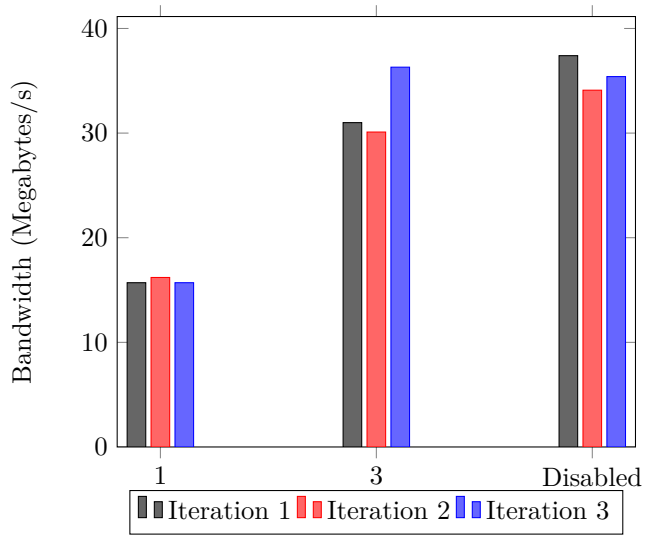

Figure 4.5: A bar plot of the measured throughput for different guard configurations

Throughputs of approximately 16, 32 and 36 MB/s for 1, 3, and without guard nodes respectively. The results with 3 guard nodes are taken from the hidden service bandwidth on the private network, which was executed in the same way as this.

Result on the public Tor network

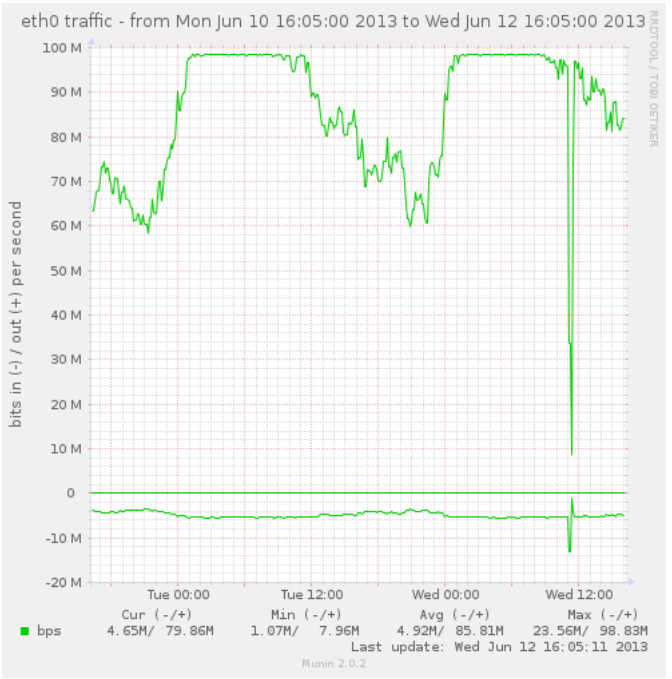

Figure 4.6: A graph of eth0 throughput on the hidden service node without guards

We can see that the test on the public Tor network varies with time, between the hours of 02:00 and 08:00 local time it manages to saturate the 100 Mb/s link from the server. The sudden drop near the end of the test period is due to a misconfiguration by the internet provider of the university which lead to duplication of packets, and a subsequent collapse of switching equipment.

Discussion

As figure 4.5 shows, the number of guard nodes influence the throughput. This is as expected, since all traffic must pass through one of the guard nodes. As long as a single guard node does not have the capacity for all the traffic, having more allow several routers to share the load. A significant difference is seen between 1 and 3 guard nodes, but not as much between 3 and with guard nodes disabled. However, other factors might be the cause here. For example we also observe an average single core CPU utilization of 98%, 93% and 95% on the hidden server for the three iterations without guard nodes. With 3 guard nodes these numbers are 89%, 86% and 96%, and with 1 guard nodes 65%, 67% and 65%. A high throughput

is correlated with a high CPU usage and the CPU in question appears to be an additional limiting factor.

For the public test, we see that at night between 02:00 and 08:00 Central European Time (CET), the bandwidth reaches its peak very close to the maximum theoretical bandwidth on the link between the server and the switch. While during the rest of the day, throughput can drop as low as 55 Mb/s. Our assumption would be that this is due to other users using the network. The important issue to note though, is that the throughput during the mentioned nightly hours is significantly (at least 50%) higher compared to accessing the same data with hidden service limited to three guard nodes.

4.5 Tor bandwidth test

Method

This test is designed to observe the bandwidth of a HTTP server accessed through Tor, but not as a hidden service. The server would therefore not be anonymous, but the total number of hops between client and server would be reduced. Otherwise the test was identical to the hidden service bandwidth test in the private network described in section 2.1.2 with various numbers of clients and similar to the guard test of section 4.5.

On the public network, we the procedure of this test is the same as the hidden service bandwidth test, so that we can compare the results.

Result on the private Tor network

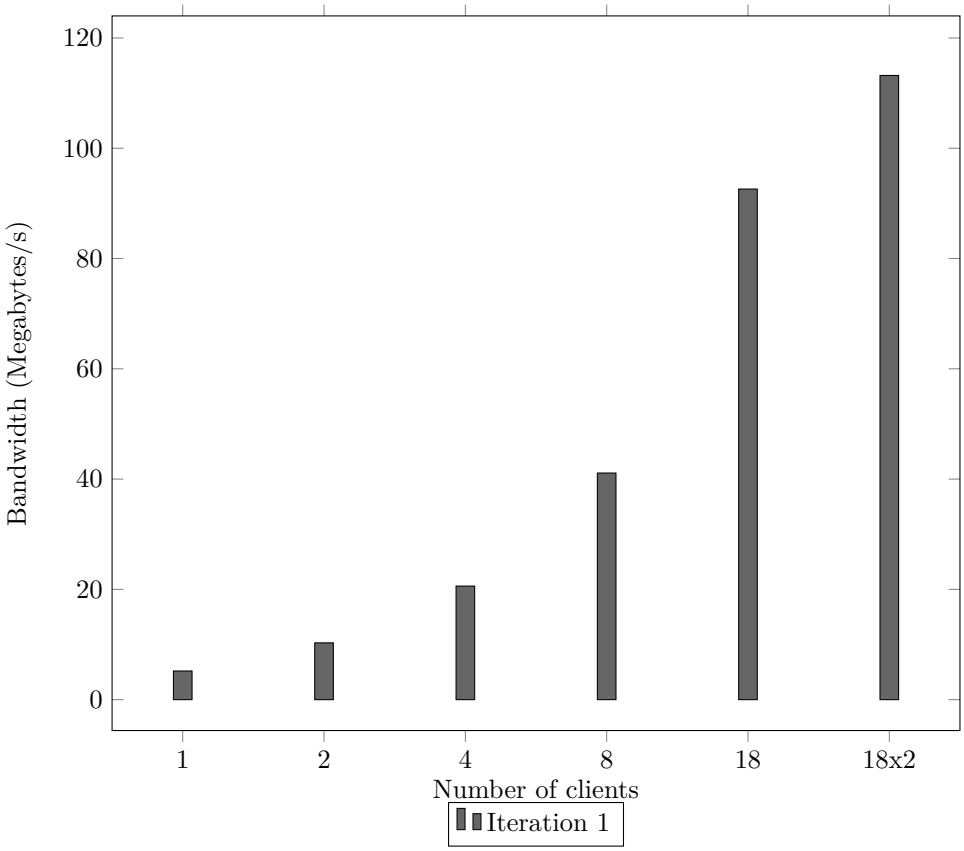

Figure 4.7: A bar plot of the measured throughput when using Tor directly

Figure 4.7 shows the bandwidth out from the server when using Tor, not as a hidden service, but only to provide anonymity to the clients. We observe a bandwidth slightly over 5 MB/s per connected client, with linear scaling until the link bandwidth is reached.

Result on the public Tor network

Min	Average	Max
97.56 Mb/s	98.44 Mb/s	99.39 Mb/s

Table 4.1: Bandwidth statistics for the Tor bandwidth test on the public network

Table 4.1 shows the statistical aggregates of the results from the bandwidth test on the public network. These numbers are taken from figure A.1 in appendix A, which in turn shows the graph of the throughput for the duration of the test.

Discussion

We see that the throughput scales linearly in the private network. The server is not limited by any guard nodes, since it is oblivious to Tor and believes it is just communicating with the exit nodes of all circuits. Therefore these results could be compared to the hidden service when it is not using guard nodes, but also to the default hidden service as the guard nodes are part of the price of anonymity.

On the public network, the results show that the server manages to saturate its link to the switch, which becomes a bottleneck. When comparing to the results of the hidden service bandwidth we see that none of those came close to a similar throughput. The highest average is approximately 30 Mb/s compared to the 98 Mb/s of this test. This is as expected, since when accessing through hidden service, the same 3 guard nodes are used by the hidden server. This means that the maximum throughput is the sum of each of the guard nodes' throughput. Compared to hidden services we observe that the direct use of Tor does not suffer from the same drops during peak hours. The most probable cause is the fact that more nodes are involved, 6 instead of 3. Therefore there is a bigger chance of a less capable node being part of the path. In addition the hidden service node will still have to do encryption and decryption operations. When using Tor directly, the large number of nodes in the public Tor network means that very few nodes are shared. Each client can have a completely different path, thereby no bottlenecks are present, except the link from hidden server to the internet, which they still have to share.

4.6 Access time test

Method

The following test is used to view initial access time of HTTP requests and its relation to the load on the hidden server. 15 clients would 10 threads each which independently retrieve a small text file containing the hostname of the server. Since the amount of transferred data is small, we can assume that the time for the request to complete is overwhelmingly due to the connection setup. Since circuits are reused if they have been used within the last 10 minutes, at least 10 minutes should be between requests on the same client if we want to include circuit setup time.

On the private network we wanted at least 600 samples for each load, thus the test needs to run for more than 40 minutes, so that each of the 150 clients (10 clients on 15 hosts) manages 4 samples each. We also need to discard the first 150 of

results as they include the act of retrieving the hidden service descriptor from the directory. We were unable include this step in all requests and if included they would skew the results. 3 clients would be reserved for putting a bandwidth load on the hidden server. They would repeatedly download the 100 MB large pseudo-random file through HTTP. The load on the server would be increased on a regular basis with the following steps:

- 1 client running 1 thread each
- 2 clients running 1 thread each
- 3 clients running 1 thread each
- 3 clients running 4 threads each

On the public server, rather than attempting to put a load on the server, we wished to explore the different steps that make up the hidden service protocol. We designed three use cases which is matches Tor's reuse of circuits and descriptors. A user is accessing the hidden service and has:

- Never visited the service before or have not accessed the service during the last 24 hours.
- Visited the site for less than an hour ago but more than 10 minutes.
- A user has visited the site for less than 10 minutes ago.

These use cases is made to fit how Tor with hidden service is caching the descriptors and reusing old circuits. Since the descriptor *descriptor-id* is changing every 24 hours, clients need to get the new descriptor from the directory when they are accessing the hidden service. Therefore use case 1 must ask the directory for the descriptor while case 2 and 3 does not. Tor also reuses old circuits for hidden service connections if they were used in the last 10 minutes. Therefore clients in use case 2 must start in step 6 of the hidden service protocol while clients in use case 3 can just send a new request through the already established circuit.

The test set-up is as following: We set up a HTTP hidden service running on *torpub* and used 30 workstations on the sahara computer lab which run 10 instances of Jmeter and Tor clients each. For use case 1 we did 6 repetitions of accessing the hidden service for each client where we removed and recompiled Tor for each repetition. This is to ensure no information is stored in the Tor client and to create a new Tor identity for the clients. For use case 2 and 3 we did 7 repetitions accessing the hidden service and discarding the first repetition because it would include the access to the directory. The delay between the repetitions is for use case 2 11 minutes and for use case 3 8 minutes. This makes a total of 1800 data points per use case.

Result on the private Tor network

Table 4.2 shows the results of the access time test. We can see that an increase in the load put on the server and the network leads to slower access times. No errors occurred during any of the tests. Only small increases are observed when the load is incremented initially, but the influence grows with an increasing load.

Load	Samples	Average	Median	90% Line	Min	Max	Error
No load	600	96	48	74	21	2877	0,00%
1 client	1520	86	63	116	20	3542	0,00%
3 clients	696	109	84	166	22	2087	0,00%
3 clients 4 threads each	623	322	243	660	35	2423	0,00%

Table 4.2: Access time for the private network. Numbers are in milliseconds (ms).

Result on the public Tor network

The results we got from the access time test is in table 4.3.

Load	Samples	Average	Median	90% Line	Min	Max	Error
Use case 1	1800	17266	13388	31480	2737	166820	0,06%
Use case 2	1800	9504	7632	15268	1663	109195	0,00%
Use case 3	1800	2815	1788	4870	337	119346	0,06%

Table 4.3: Access time for the public network. Numbers are in milliseconds (ms).

Discussion

The results here indicate that access time can increase significantly with a heavy load, for the 3 cases with less than three clients generating a load on the network, the initial access time is very usable for uses like web browsing where many connections are made to different servers. With averages around 100 ms the access delay will be minor compared to transfer times. When the network is more actively used though, initial access delay quickly becomes more significant. Still, 90% will be below 660 ms, while noticeable and a detriment to usability it may be a worthwhile trade for the anonymity it provides for both sides.

As the results in table 4.3 shows, the median, 90% line and minimum are largest for case 1 and decreasing for case 2 and 3. This is as expected since in use case 1

there are more steps in the hidden service protocol to acquire access compared to the other cases.

Under the assumption that the steps from use case 3 uses the same average time when they are performed in case 2. It is possible to calculate the average time it takes to do the extra steps in case 2 by subtracting the average time in case 3. The same method can also be utilized on case 1 by subtracting the average time from case 2. This gives the following times for the tasks:

Task	Average time
Acquire hidden service descriptor	7762
Establish circuit to hidden service	6689

Table 4.4: Average times for a client to do different tasks when connecting to hidden services. Times is in milliseconds (ms) and is derived from access times in table 4.3.

When comparing the case 3 average access time to both tasks. It shows that the hidden service protocol is time consuming compared access time when the circuit is already established. For the use case 3 with an average access time of 2815 ms is higher than the median of 1788 ms. These number show that a most results are relatively low with a few very high results which pull up the average.

4.7 Connection latency test

Method

This test is looks at the round-trip time (RTT), also called connection latency, of an active connection. RTT is a measure of the network delay of packets, and is typically measured by sending small message and time when the response arrives. A high RTT can be acceptable for many applications, particular non-interactive applications like data transfers where the time taken is more dependent on the time to transmit all packets (bandwidth) than the time for a single packet to arrive. For interactive application like Voice over IP (VoIP) however, the International Telecommunication Union finds that users start to get dissatisfied when the mouth-to-ear delay exceeds 280-300 milliseconds. [ITU03] When there is a lot of variation in the delay, a typical approach is to buffer any incoming data. However, buffering increases the general delay.

In order to simulate VoIP traffic we created client and server application that transmit and receive messages on a regular basis. The client times how long it takes for the server to respond, and it can be configured at different bandwidth targets

and packet rates. We determined that 24 kilobit per second is a typical data-rate for VoIP codecs like Skype and Speex. [KC11] The former is also capable of adjusting the packet rate up to a maximum of 100 ms between packets. [Ptá12] Due to the fixed cell size in Tor, we believe a high number of packets will lead to inefficient resource usage due to padding. Thus the slower packet rates are a better choice if VoIP software is to be used with Tor. In the end, we chose to measure when sending 10 packets per second, which means a 100 ms inter-packet delay. With a rate like that we quickly get a large number of samples, so that tests do not need to be longer than one hour.

Since the network is expected to have other users simultaneously, 3 clients would be putting a bandwidth load on the hidden server. Like in the initial access time test in section 4.6, they would repeatedly download the 100 MB large pseudo-random file through HTTP. The load on the server would be increased on a regular basis with the following steps:

- 1 client running 1 thread each
- 3 clients running 1 thread each
- 3 clients running 4 threads each

For the public test we do not repeat tests with varying load, but run a single test over a long period of time.

Result on the private Tor network

Figure 4.8 shows the cumulative distribution plots of RTT of the various clients used during the first delay variation test without any load. We observe that they are roughly similar for all nodes, but the median varies between 20 and 60 ms. Another observation is that several clients exhibit a large number of samples around the 100 ms mark, shown by a near-vertical line in the plot.

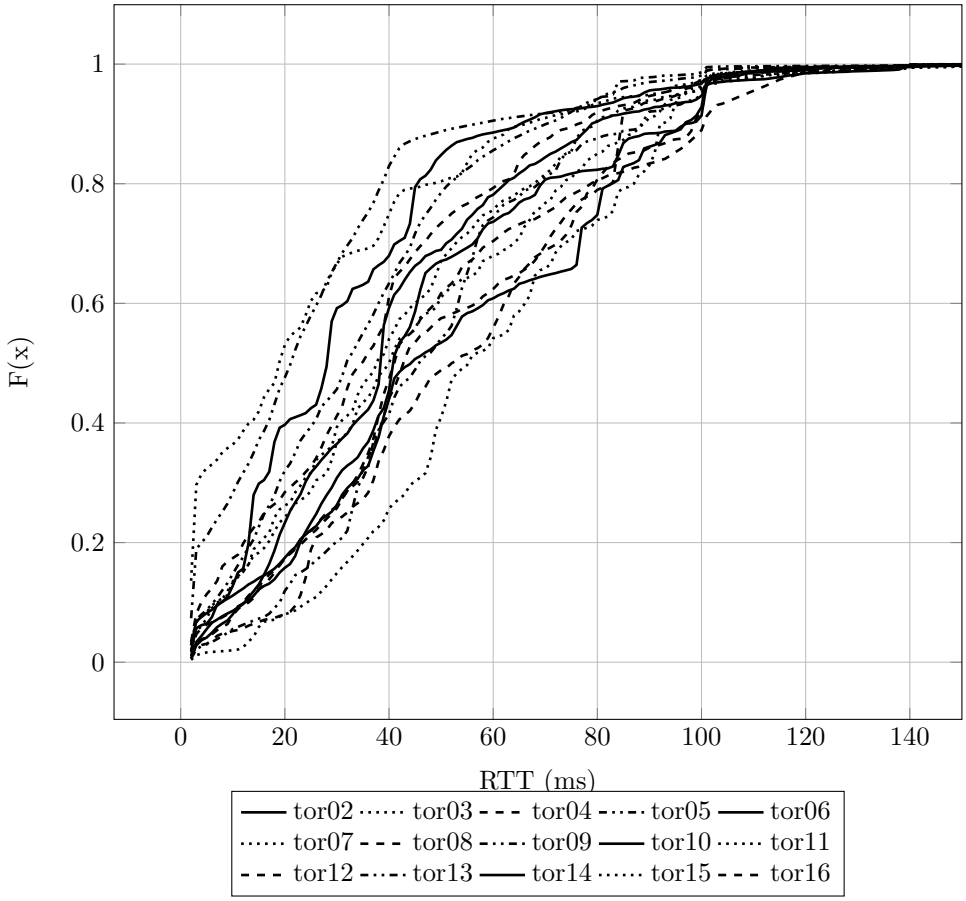

Figure 4.8: The cumulative distributions for clients during a single delay variation test on the private network

Figure 4.9 shows the cumulative distribution plots of RTT during four different loads on the hidden server. We see that a heavy load increases the delay, the median is 100 compared to 40 ms for the test without any load. A load consisting of a single client improves the delay, with a median of 21 instead 40 ms. However, the minimum and maximum are still very similar.

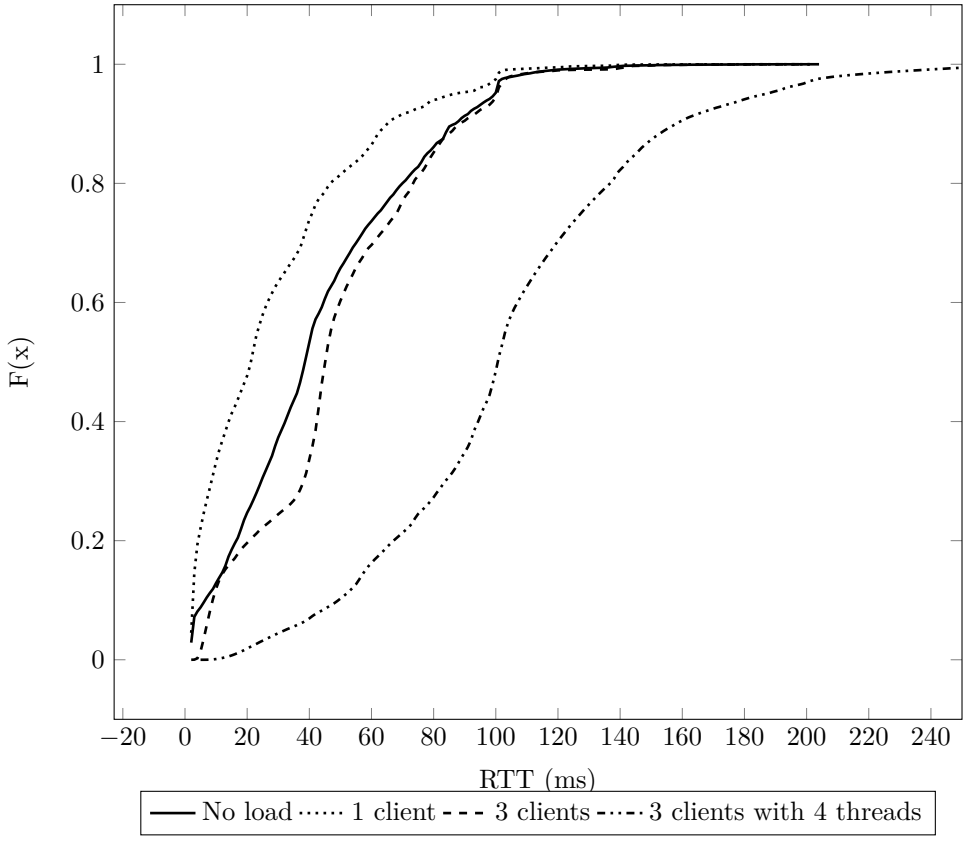

Figure 4.9: The cumulative distributions for different loads during a delay variation test

Result on the public Tor network

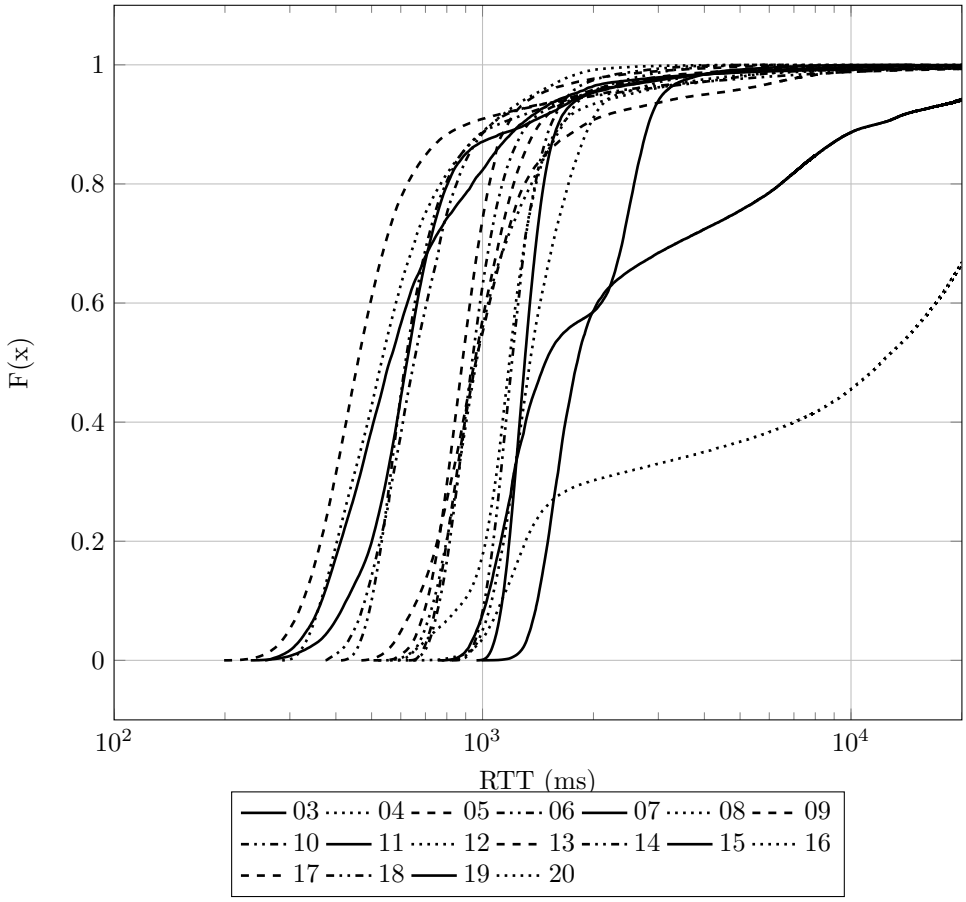

Figure 4.10: The cumulative distributions for clients during a single delay variation test

We see that many of the results are roughly similar for all nodes, but some vary quite significantly from the rest.

Discussion

Contrary to our expectation, a single client load improves the delay. For example, 58% of the samples fall within 25 ms while for the test without load it is 31%. We believe this is due to congestion control mechanisms like Nagle's algorithm used in TCP. [Nag84] Due to this algorithm the TCP stack will wait for more data if the

current amount waiting is less than the Maximum Segment Size and it is waiting for an acknowledgement for data previously transmitted. Its purpose is to avoid sending a entire packet for small amounts of data. For example, when sending 20 single bytes without Nagle's algorithm enabled, it might send 20 segments, each with a 40 byte header. This would mean 820 total transmitted bytes for 20 bytes of payload, a very low efficiency. With Nagle's algorithm enabled, it might send the first byte in a segment since there is no unacknowledged data, but wait for the remainder before sending a second segment. This equals a total of 60 bytes transmitted, a significant improvement. In the case with no load, the highest published bandwidth figure in the last consensus for the test is 191 for Relay03. For the case with 1 client load, seven different relays all have bandwidth figures over 5500, this figure equals the number relays involved in a single hidden service circuit (three for each circuit to the RP and the RP itself). Thus we believe these nodes are able to demonstrate their bandwidth and since bandwidth is a weighting factor for clients when selecting which relays to use, the seven will be used more frequently. When sharing a path, the link between two relays is less affected by Nagle's algorithm as more cells are transmitted, and there will be enough queued data for a complete TCP segment more often. The GSO and TSO network interface options which we discovered in section 4.1 are similar. They also attempt to reduce the processing time introduced for each packet by combining more packets so the processor can be used more efficiently. Like Nagle's algorithm this may introduce delays when waiting for more packets to combine into one.

Another interesting factor is the peak around 100 ms, we believe these are not coincidental, but due to the fact that we send packets every 100 ms and thus if a packet is combined with the next one a constant 100 ms delay is added. Such a combination may be due to Nagle's algorithm present as part of TCP congestion control or similar attempts to improve efficiency by increasing the payload per packet or cell.

Some nodes are clearly unlucky in the public network, these few have an overall higher connection latency. This can be caused by several factors like the initial choice of guard nodes not able to handle the bandwidth, and therefore accruing queuing delays. Other parts of the complete circuit may be slow as well, since it is in constant use it will remain the same for the duration of the test.

4.8 Reliability

Method

To measure reliability we chose to run long term tests, with 15 servers running 10 threads. Like the access time tests, each thread would request a small page, with

regular, on average, 11 minute intervals. This means that as many steps as practical are part of the process, and therefore possible failure causes.

On the public network, we increased the number of servers to 30 for increased number of samples.

Results

Network	Samples	Error
Private	11569	0,00%
Public	25009	0,10%

Table 4.5: Reliability results

Table 4.5 shows the number of samples and the percentage of failed requests for both the public and the private tests.

Discussion

As we can see from the results, both the public and the private networks are highly reliable. This shows hidden services cope with congestion and packet loss which could happen to the public Tor routers. Since TCP is used as a transport layer protocol for Tor, this is as expected. TCP provides reliable transport, but does not make guarantees about timing. The source for the errors could most likely be congestion of a Tor router and thus creating a time-out.

Chapter 5
Discussion

5.1 Increasing the number of guard nodes

As we demonstrated in section 4.4, the number of guard nodes is a factor in maximizing throughput from the hidden server. Disabling the guard nodes in the torrc config-file allows for more data to be transferred. However, guard nodes were introduced for a reason; they limit the exposure to possible malicious routers in the Tor network. [WALS03] The principle of guard nodes is that a Tor client attempts to use the same routers as the first hop when creating circuits. Normally it selects three routers used for this purpose. For normal, 3 hop, Tor usage, anonymity of the client can be compromised if an attacker control both the first hop and the last hop. With timing and volume analysis the attacker can then deduce which of the exit traffic originates from the client. For hidden servers, only a single guard node needs to be controlled by the attacker. [OS06] Since hidden services can be contacted, an attacker can generate traffic and if he controls the first hop of the circuit from the hidden server to the rendezvous point, he can determine the location of the hidden service by timing and volume analysis of his generated traffic and the traffic relayed to the hidden service node. It is still possible that he is another part of the circuit, but over time the IP-address of the hidden server will have a stronger presence than any others in the dataset because it is part of every circuit while the others are chosen at random from a the list of all routers. Øverlier and Syverson have shown that with this type of attacks, it takes only minutes to find the location of a hidden server. [OS06]

If we assume that there are m malicious nodes part of a Tor network of size n, and that nodes are chosen completely randomly rather than through weighted selection. Then the probability that a hidden service picks at least one malicious node as its guard node when selecting g guard nodes:

$$P(x) = 1 - \prod_{i=0}^{g-1} \left(\frac{n-m-i}{n-i} \right)$$

There are as of the writing of this report approximately 3500 routers in the Tor network.[1] If an attacker manages to maintain several onion routers, then the chance of an hidden service choosing at least one of the attackers routers as a guard is given in table 5.1. We see that even with the three guard nodes chosen by default an attacker has a chance of nearly one percent to become a guard node.

Malicious nodes	1 Guard	3 Guards	6 Guards	12 Guards
10	0.29%	0.85%	1.70%	3.38%
20	0.57%	1.70%	3.38%	6.66%
30	0.86%	2.25%	5.04%	9.83%

Table 5.1: Probability of choosing at least one malicious guard node

In reality, weighting is more complex, rather than solely dependent on number of nodes, it is weighted by bandwidth and the different possible flags. With 10 servers all capable of fully utilizing 100 Mb/s full-duplex links, a total of 2 Gb/s = 250 MB/s can be used for Tor traffic. This equals about 10% of the current Tor network as we found in 3.3.3. For example, our C&C node which has been a Tor relay for the duration of our work on this thesis, has achieved a guard probability of approximately 0.5% over this period of time.[2] Even peaks above 1% are seen, this is all achieved on a 100 Mb/s full-duplex Ethernet connection.

5.2 Application viability

5.2.1 Web-server

HTTP is a commonly used protocol for retrieving web pages. Some important requirements for the usability for HTTP web-browsing are an acceptable initial access time and enough bandwidth when loading pages. When looking at the bandwidth results discussed in section 4.3, even the lower figures can be acceptable. The total bandwidth is low when considering the sharing however the typical usage pattern of web has bandwidth demands for short periods of time when a page is loaded. In between page loads the need is negligible, and statistically users will not be synchronized, thus spreading out load. User can also adapt to slow loading speeds and start loading a page while reading another, thus pipelining requests. The latter also has benefits when access times are longer, the main performance metric is the total time for request completion which means the access time plus page loading time. This makes running a web service possible in in the public network with decent performance.

[1] https://metrics.torproject.org/network.html
[2] https://atlas.torproject.org/?#details/E2D280A2A2BAC993704BDB8878843C4B64794472

5.2.2 Instant messaging

Instant messaging has latency requirements. However, contrary to what the name suggests, it does not have to be instant. Typing a message will normally take several seconds, so in relation, the delay introduced by hidden services is not very significant. The bandwidth requirement for instant messaging is also low thus making hosting a instant messaging service on the current public Tor network possible.

5.2.3 Voice over IP

Voice over IP has high requirements for a low and predictable round trip time. The International Telecommunications Union has recommends that mouth-to-ear delay is less 150 ms. [ITU03] The ITU also finds that user get dissatisfied once delay exceeds 280 ms. From the results presented in this thesis, we can see that the current performance of hidden services do not meet either of these for the public network. It has a long way to go, but from the results of the private network we can estimate the performance for times when bandwidth is more readily available. However, the private network also does not suffer from propagation delay due to the close proximity of the onion routers. If the public Tor network manages to increase its capacity, we might be able to see a RTT within the recommendations of the ITU, even when factoring in longer delay due to geographic spreading of the nodes. With a larger number of available routers, it might be feasible to set up circuits using routers on the same continent, thus eliminating transoceanic propagation delays. Although, intentionally selecting routers to be used rather than selecting at random, might lead to higher chances of someone compromising the user's anonymity. Therefore running a hidden voice over IP service on the current public Tor network can not give satisfactory service to most users.

5.2.4 Video streaming

Video streaming has both bandwidth and access time requirements. Delay variation is less of a factor since it can be mitigated by delayed playout and buffering, this introduces addition initial delay, but in return can avoid interruptions during playback. Research by Krishnan and Sitaraman shows that users often expect videos to start within a short amount of time. [KS12] For short videos, defined as videos with a duration of less than 30 minutes, more than 40% may abandon the video if playback does not start within 10 seconds. Looking at the results in section 4.6, we see that a hidden service on the current public Tor network has a median value of a little over 13 seconds. From Krishnan's and Sitaraman's work, we see that such a delay can lead to abandonment rates of up to 50%. However, the users' expectations have a significant impact, for example they found that for mobile devices the abandonment rate is nearer to 20% while for users with high-speed internet access this figure approaches

80%. Thus if users are aware of the cost of anonymity in this access time aspect, this might not have a large impact.

Concerning bandwidth however, Netflix, a stream provider, recommends at least 5 Mb/s for a HD video stream.[3] The bandwidth figures from the bandwidth test discussed in section 4.3 indicated the total bandwidth for the public network is larger then the Netflix recommendation. But for increasing number of streams the bandwidth per stream will drop below the recommended bandwidth. Therefore hosting a hidden video streaming service over the current public network is not usable with more than a few clients. On the other hand hosting a hidden video streaming service in the private Tor network is possible. The network has a low access time and can deliver enough bandwidth for many multiple clients simultaneously. This indicates hosting a hidden video streaming service may be possible in the future if the public network is upgraded to have more bandwidth available.

[3] https://support.netflix.com/en/node/306

Chapter 6
Conclusion

In this thesis we have learned how the hidden service protocol works. We have successfully configured hidden services both in a high-bandwidth private Tor network consisting of several dedicated servers, and in the public Tor network. We conducted multiple tests to investigate how the hidden services perform under different types of applications. Tested parameters include initial access time, connection latency, bandwidth/throughput and reliability in both networks. We also measured the performance of accessing services through hidden services compared to services through normal Tor. Our measurements indicated hidden services are reliable with low error rates.

Our private Tor network delivered high throughput, low access times, and low round trip times making multiple services possible including web services, instant messaging, voice over IP and video streaming. This shows the hidden service protocol is capable of delivering a variety of different services with high performance.

For the public network the available bandwidth is limited and varying. The access time and round trip time is significantly higher compared to our private network and is varying a lot. The results concluded that web services and instant messaging is possible to host as hidden services in the public Tor network. On the other hand the high round-trip times makes voice over IP services not satisfactory for most users. The reduced bandwidth makes also video streaming not possible for more than a few users.

Chapter 7
Future work

There are still several open research topics with regard to hidden service performance we have not looked at. First, we observed that Tor is limited by CPU processing capability. Intel has added new instructions for their processor aimed at adding hardware support for AES encryption and decryption. For CTR operation, which is used in Tor, they claim performance is improved by more than an order of magnitude. [Gue10] This can lead to significantly improved performance for Tor. We have not measured it, but it is reasonable to believe that the triple encryption of cells over a circuit does make up a significant proportion of Tor's processing time.

Secondly, we observed that Tor uses a single CPU core, often with near 100% utilization. There is work on separating cryptographic operations so they can be run simultaneously on other cores, however with AES-NI crypto functions are becoming proportionally less. Since Tor relays process data from independent circuits, there are options of increasing the multi-threading capability of Tor so it utilizes more available processing resources.

Attempts can be made to improve the Tor implementation for increasing performance for hidden services. Tasks might be balancing the line between anonymity and performance with regards to guard nodes and other configuration parameters. Alternatively, optimizing the source code of the current implementation may also be an interesting challenge, and using the private Tor network setup we have demonstrated in this thesis can be used to measure the effect of any changes made.

References

[All] Ethernet Alliance. Ethernet jumbo frames. http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[Gue10] Shay Gueron. Intel advanced encryption standard (AES) instructions set. *Intel White Paper, Rev*, 3, 2010.

[HC98] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[ITU03] ITU-T Study Group 12. ITU-T Rec. G.114: One-Way Transmission Time. Technical report, International Telecommunication Union, 2003.

[Jos06] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[KC11] Kyungtae Kim and Young-June Choi. Performance comparison of various voip codecs in wireless environments. In *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '11, pages 89:1–89:10, New York, NY, USA, 2011. ACM.

[KS12] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 211–224. ACM, 2012.

[LGL+96] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard), March 1996.

[Nag84] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[OS06] Lasse Overlier and Paul Syverson. Locating hidden servers. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[Ptá12] Luboš Ptáček. Analysis and detection of skype network traffic. 2012.

[rJ01] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFCs 4634, 6234.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[WALS03] Matthew Wright, Micah Adler, Brian Neil Levine, and Clay Shields. Defending anonymous communication against passive logging attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 28–43, May 2003.

[Xu] Herbert Xu. GSO: Generic Segmentation Offload. http://lwn.net/Articles/188489/.

Appendix A
Results

A.1 Tor bandwidth in the public network

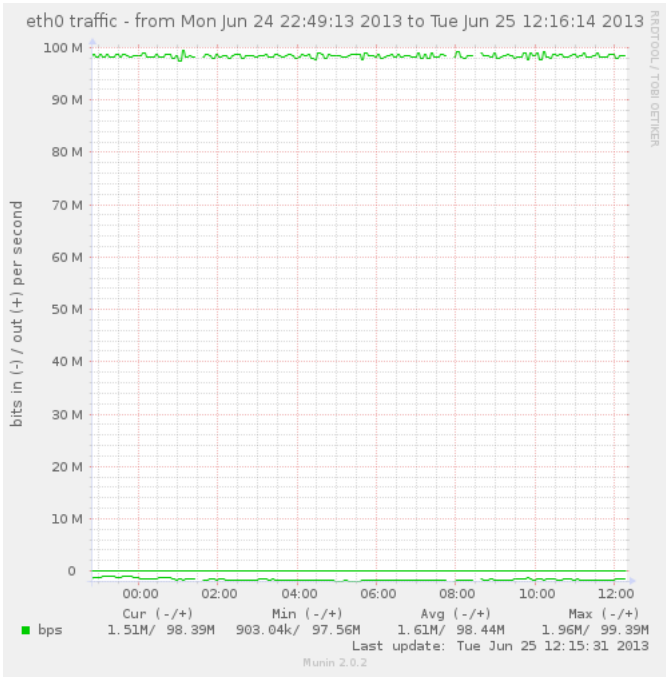

Figure A.1: A graph of eth0 throughput on the hidden service node using Tor without hidden service

A.2 Test length in the private network

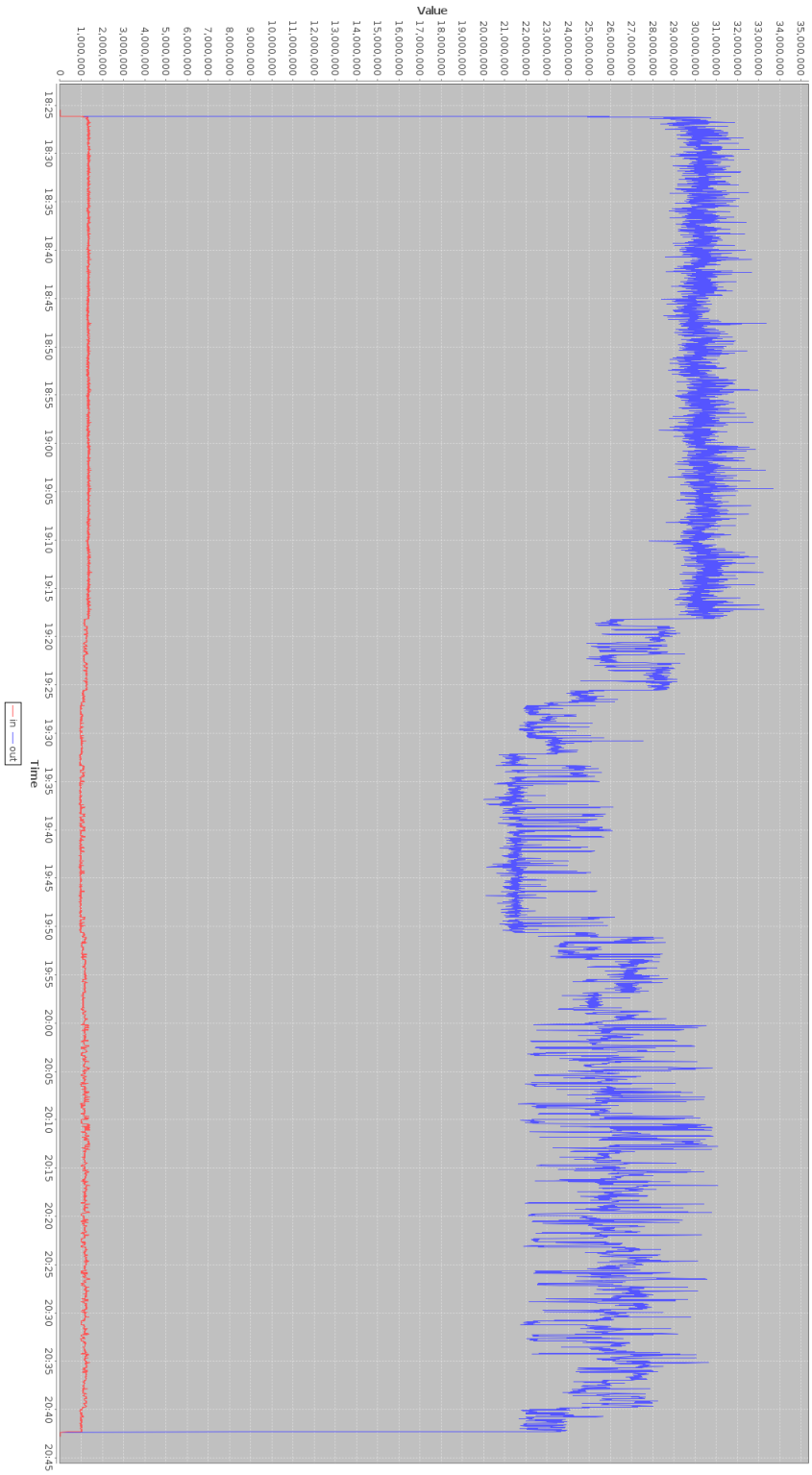

Figure A.2: A graph of eth0 throughput changing over time for the hidden service node

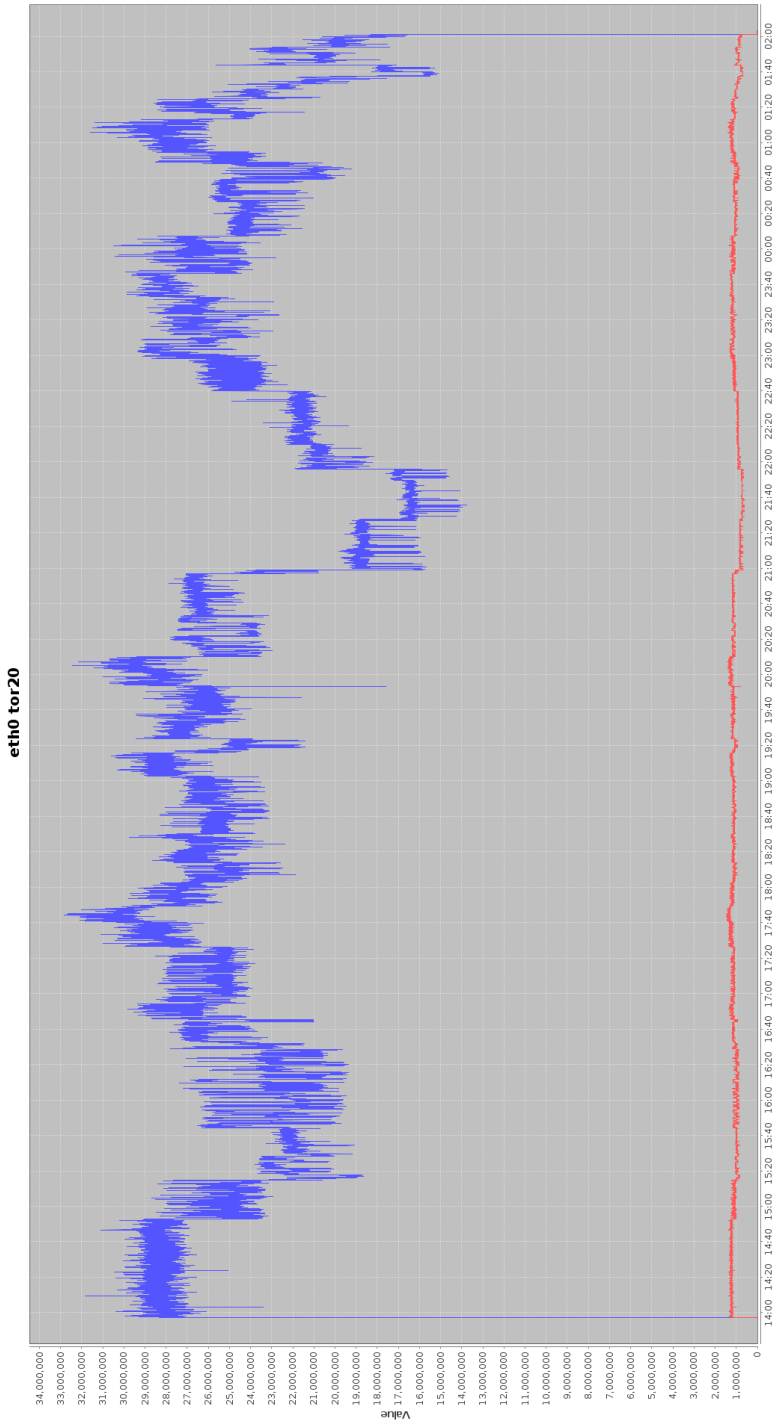

Figure A.3: A graph of eth0 throughput on the hidden service node