University College London

Thesis

Tor: Hidden Service Scaling

Author:
Ceysun Sucu

Supervisor:
Steven J Murdoch
Alec Muffett

MSc in Information security

September 2nd 2015

Abstract

Tor's hidden services in its current state does not fully utilise multi-core architecture or provide any load balancing options in regards to multi-server systems. This thesis explores possible techniques which could be used in scaling Tor's hidden services horizontally, more precisely looking at possible load balancing techniques and hidden service resiliency. The first section of the thesis will give an detailed overview of Tor and look at relevant scaling and load balancing techniques. The body of the thesis will experiment with possible techniques and evaluate them. Finally I will indicate the aim of this research and possible future directions.

Keywords. Tor, Hidden services, Load balancing, Scalability, Resiliency.

Acknowledgements

I would like to thank my supervisors Steven J. Murdoch and Alec Muffett for their constant guidance and encouragement. I would also like to thank the people in the Tor development community that answered my questions though out the duration of my thesis.

Contents

List of Figures

Abbreviations

DH	**D**iffie **H**ellman
DHT	**D**istributed **H**ash **T**able
HSDIR	**H**idden **S**ervice **Dir**ectory
DNS	**D**omain **N**ame **S**ystem
HS	**H**idden **S**ervice
IP	**I**nternet **P**rotocol
MSc	**M**aster of **S**cience
OR	**O**nion **R**outer
OP	**O**nion **P**roxy
RP	**R**endesvous **P**oint
SSL	**S**ecure **S**ocket **L**ayer
TLS	**T**ranspot **L**ayer **S**ecurity

Chapter 1

Introduction

1.1 Motivation and goals

Tor is the leading software in anonymous communications[1]. It is a low latency overlay network which is designed to achieve anonymous communications by routing its traffic through a large pool of relay nodes, in such a way that the privacy of users is preserved. Many applications can be run through Tor, but for this project we are concerned with web services. Hidden services is a feature of Tor which allows both users and services to preserve their privacy.

Tor achieves anonymity by constructing a six hop route through a selection of relay nodes to a hidden service. We refer to this route as a circuit. The first three hops of this circuit are chosen by the client to a meeting point(rendezvous point) which is chosen by the hidden service. The last three hops are chosen by the hidden service. The first three hops are there to preserve the anonymity of the client, and the last three hops are there to preserve the service providers. However the three hops on the server side can be reduced to one hop, which would result in a shorter and faster circuit. Which would mean that the anonymity of the service provider is reduced yet still persevering the clients. The details of the Tor protocol will be explained in the next section, however it is important to note for the context of this thesis, that the thesis is concerned with non-anonymous communications,

that is to say that Tor is still useful when the anonymity of the service provider is not needed. An example of this could be a web service which is known and used by many, such as a social network. A service of this type may not need to protect their anonymity but may benefit from the other security properties Tor has to offer. Such as the anonymity of the client and end to end encrypted services. The reduction to this three hop circuit also means that the speed of the service is increased, as there is less hops on the circuit, there is also less latency.

Tor in its current states does not make use of modern multi-core systems, it runs as a single process on a single core. Some use is made of multi-threading for cryptographic operations, but for the most part it is only scalable vertically, thus to increase its throughput it would require a faster processor. However even this would not make much of a difference as modern processors are multi-cored, which Tor would not take full advantage of. Another issue of Tor in relation to hidden services is the effect a large web service would have on introduction points. A hidden services descriptor contains anywhere between three to ten introduction points. And as introduction points are regular relays with limited bandwidth capabilities, they will receive large amounts of stress for web services serving a large amount of clients.

Tor also does not provide any means of load balancing. With the modern web and DNS, there is DNS round-robin, where users get distributed across a number of IP addresses pointing to the same service. Tor provides no support for such a service, as all traffic gets push through to a single Tor instance. For Tor to scale, this issue of load balancing must be addressed as it will result in a major bottleneck for the system when large web services start to migrate to Tor.

The thesis will try to provide a deeper understanding of Tor hidden services, and look at some possible architectural changes and modifications which will address the issue of scalability while preserving the security properties of are threat model.

As opposed to the original threat model stated in Tor's design paper[2], The threat model this thesis considers is that under deploying a Tor hidden service for a large scale web service. This setting does not require the anonymity of the

service provider to be preserved, only the clients. Consider the setting of a user in a country who's censorship law and guards prevent that user from accessing a certain web site, or simply a user who wants the end to end encryption properties provided by Tor. The threat model is exactly the same as the previous one however, the anonymity of the service provider is not needed, only anonymity of the user and the privacy of the information shared between them.

1.2 Structure

This thesis is divided into six main parts. Chapter 1 looks at the motivation behind the project and some issues with Tor that need addressing. Chapter 2 Gives a overview of Tor and an in-depth look into hidden services. Then it goes on to look at relevant works in to hidden service scaling, firstly looking at works directly related to Tor and then looking at other multi-server architectures. Chapter 3 gives the methodology and looks at some of the important components which will be used during experiments and testing. Chapter 4 provides a glance at a possible approach that utilises using multiple Tor instances with the same hostname and private key, it looks at measuring the performance and behaviour in controlled experiments. Chapter 5 expands on this approach by providing a method which involves publishing different descriptors to different directories. Finally chapter 6 concludes with an evaluation on the method provided and looks at possible future directions.

Chapter 2

Background

2.1 Tor

Tor is a distributed overlay network, it maintains a large network of nodes called onion routers which we call the Tor network. In order for communication to be achieved over this network, a sequence of virtual tunnels are created through a random selection of relay nodes, which we call a circuit.

There are a number of different types of relays(onion routers), each severing different functionalities. Middle relays, relay traffic from one relay to another. Middle relays are publicly advertised to the Tor network. Exit relays are the final node on the circuit before the destination, thus as a result, any traffic being transferred through the circuit will have the source destination as the exit relay. They are also publicly advertised to the network as they need to be used by any Tor user. Bridge relays are relays that are not publicly listed on the Tor network, they act as alternative entry points. There purpose is to circumvent censorship or to hide the fact that a client is using Tor from its ISP. There are attacks that involve compromising the entry and exit nodes in a circuit. To reduce the likelihood of these attacks, a Tor client will maintain a list of entry nodes which it will use for all circuits, which are known as guard nodes. The general flow of a circuit is that

traffic gets routed through a guard to some middle relays and finally out an exit relay to its destination.

When communication is to be achieved over the Tor network, Tor must first create an circuit in which to communicate over. From a list of available relays on the network, a random path is chosen. The creation of a circuit involves step by step communicating with each relay on the path and establishing a symmetric key. Each symmetric key must be different, and is done using Diffie–Hellman key establishment. At each step, Tor uses a partially created circuit to establish keys with the next relay. This is done in order to preserve anonymity between other relays on the circuit.

Data that is to be transmitted over the circuit is constructed such that the data and the IP address of the next relay on the circuit, is encrypted multiple times. As a result every relay on the circuit will decrypt the packets to receive the address of the next relay on the circuit, until the packet reaches its destination. This way at any one time, a relay only knows the address of the previous and next hops on the circuit.

Packets sent over these circuits come in fixed sized cells of 512 bytes of the form of a header and a payload. The header contains the circuit ID which refers to the circuit in question and command flag which says what to do with the payload. All connections are maintained using TLS/SSLv3 connections, and each relay maintains a long-term identity key which is used to sign its descriptors, which is a summary of its state and any relevant information. Each relay also maintains short term keys which are rotated periodically. They are used during circuit creation in order to negotiate ephemeral keys with other relays and Tor clients[3].

2.2 The Onion Proxy

The onion proxy is the part of Tor which allows the user to connect to some service on the Tor network or to simply communicate over it. It essentially has

a bootstrapping phase which fetches information on available relays from a set of authorities, establishes some circuits beforehand and prepares the environment to communicate over the Tor network. It then allows the user to communicate to other services by handling connections over the Tor network[2].

2.3 Path Selection

Depending on the version of Tor being used there will be a slight variation to its circuit selection. The initial sequence of steps Tor takes is its bootstrapping. Firstly Tor must receive a consensus from the directory authorities, this consensuses contains a summary of available relays on the network. It then begins to build circuits. Circuits will be built ahead of time, in order to speed things up.

Tor will start to build internal circuits during the bootstrapping sequence. Internal circuits are used to connect to hidden services. For the newer versions of Tor (0.2.6.2-alpha and later), if the consensus contains exit relays, Tor will also build exit circuits which serve services outside of the network.

Once this initial sequence is complete, a client can then go ahead and make requests to access some hidden service or another service. When one is made a new stream is created, in which a circuit must be selected to handle the request, this can be selected from the previously constructed circuits which are rotated over time. Tor maintains a number of pre built circuits to serve streams quickly. As different circuits perform different purposes, Tor creates circuits which depended on what ports we have accessed in the past, in addition to some default exit and internal circuits. As time goes on Tor will adapt the circuits it builds depending of the requests it sees from the user. If no requests are made for an hour it will no longer do this.

If none of the circuits can handle the request, a new circuit is constructed. If the new circuit to be created is an exit circuit, an exit relay is selected which can

handle the request and a circuit is constructed to it. If it is a internal circuit an arbitrary path is selected and repeated as needed[4].

2.4 Directory Authorities

A small amount of trusted onion routers(a term describing the different type of nodes) act as authoritative servers, which track the networks topology. Currently there are nine authoritative servers[5]. The network is created by onion routers publishing signed descriptors to the directory authorities. These descriptors contain state and policy information. A list of directory authorities and there public keys is known before hand as it comes pre-packaged with Tor software. The authorities then combine the data and vote on a view of the entire network and publish a consensus which contains this information.

Authorities must vote on a view of the network and produce a consensus. The voting algorithm used is a simple one. Firstly authorities produce a signed vote which contains a composition of the entire network and information on the state of all routers. Then a number of steps is performed to ensure that every authority receives a signed vote. Each authority follows the same algorithm to produce a consensus, if each authority received the same set of votes, the produced consensus should all match up. The final consensus is then signed and published with the signatures of each authority.

In order for clients and routers to use Tor network they need a list of available routers from which to build they circuits from, this list comes in the form of a consensus. Routers can act as directory caches allowing them to cache consensus documents obtained from the authorities. The Consensus are obtained from the authorities when a Tor instances first bootstraps. As the consensus contains descriptors on all available routers on the network, these descriptors over time may be invalid, due to a relay changing its policies, or going down. If an onion routers list of routers is no longer up to date it can download missing descriptors from a directory cache which as a result reduces the load from the directory authorities[6].

2.5 Bridge Authorities

As bridges are not publicly listed, the way in which a bridge advertises its existence is through bridge authorities. Similar to directory authorities however the difference is that a directory authority will provide a list of all known relays but an bridge authority will provide a descriptor only if it is provided with the corresponding identity key. Meaning that you cannot just grab a list of all bridges, as that will defeat its purpose. But if you know about an bridge you can track its state. A bridges publishes its descriptors to bridge authorities in a secure manger in order to ensure that an adversary monitoring incoming connections cannot list them. The identity of the bridge authorities comes pre-packaged with Tor[7].

2.6 Hidden Services

Tor provides for two way anonymity, not only providing users with privacy but also service providers. Hidden services allow for TCP services such as instant messengers or web servers to be used over the Tor network without revealing its IP address. Communication is done in such a way that the user does not know the location of the service and the service provider does not know who the user is[2][8].

2.6.1 Protocol

Firstly the Tor generates an public/private key pair for the hidden service which is stored locally on the same machine as the hidden service. It then must select a number of introduction points, which are relays that act as a meeting point between the client and hidden service during communication. A single hidden service will generally have a minimum of 3 introduction points and a maximum of 10.

In order to establish an introduction point, Tor creates a circuit to the introduction point and sends it a command which signals that it is trying to establish an introduction point. The message also contains the public key of the service, session relevant data and signatures. The relay in question will then do some calculations to confirm the request, if successful it will reply with a similar type message.

The hidden service must then generate a descriptor. Once generated it is published to the necessary hidden service directory authorities. The user equipped with the .onion address of the hidden service queries the hidden service directories for an descriptor.

The clients onion proxy now equipped with the necessary information on the hidden service, constructs a circuit to it. It does this first by constructing a partial circuit which acts a meeting point. This is done by selecting an random onion router on the network known as a rendezvous point and building a circuit to it. The clients onion proxy sends a message which signals an attempt to create an rendezvous point and which contains an rendezvous cookie that is used to recognize the hidden service.

The clients onion proxy selects an random introduction point from the descriptor and build a separate circuit to it. It provides a message encrypted with the public key of the hidden service and some information about its self, the rendezvous point and the cookie. Then begins a Diffie–Hellman handshake. Now the hidden service builds a fresh circuit to the users rendezvous point. Provides it with a rendezvous cookie, a reply to the Diffie–Hellman handshake and a handshake digest. This information is now used to confirm the authenticity of the recipient and to produce a session key which they can both use for encrypted communications. [8].

There are two types of descriptors use. V0 is used for tor – 0.2.2.1-alpha and prior any thing after uses V2 descriptors. For this thesis we are only concerned with V2 descriptors [8]

- Descriptor id: identifier for this descriptor, this is a base32 hash of several fields

- Version: hidden service descriptor version
- Permanent key: long term key of the hidden service
- Secret id: part used to validate descriptor id
- Published: time in UTC when this descriptor was made
- Protocol versions: list of versions that are supported when establishing a connection
- Introduction points: A list of introduction points. An optional "descriptor-cookie" can be used to decrypt the introduction-points if it's encrypted, An introduction point contains the following:
  - Introduction point: hash of this introduction point's identity key
  - IP address: address of this introduction point
  - Onion port: port where this introduction point is listening
  - Onion key: public key for communicating with this introduction point
  - Service key: public key for communicating with this hidden service
  - Intro authentication: used to perform client authentication
- Signature: signature of the descriptor content

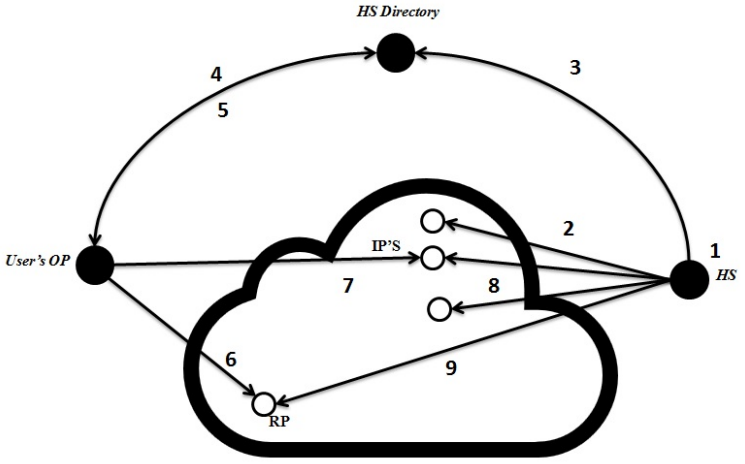


FIGURE 2.1: Hidden service protocol.

2.7 Hidden service directories and the distributed hash table

2.7.1 Overview

In order to connect to a hidden service the clients onion proxy must query some sort of lookup service. This lookup service comes in the form of a distributed hash table which allows for descriptors to be spread out over different relays across the Tor network[8].

Directory authorities will vote on relays that have been up for 24 hours to become hidden service directories, this is to ensure only highly available nodes serve as directories. These relays will be flagged with the HSDir flag, which means they act as hidden service directories and can handle v2 hidden server descriptors. The hidden service directories will contain routing tables for hidden services, allowing clients to fetch descriptors and hidden services to publish them

The process of calculating which hidden services directories are responsible for a given services descriptor is deterministic. This is done so a client that wants to connect to a hidden service can follow the same calculations in order to determine which hidden service directories the descriptors where published to. At any one time a hidden service has six directories responsible for its descriptors. And a client will maintain a rotating subset of V2 hidden service directories.

In previous versions, Tor had a dedicated set of hidden service directories(V2), however this approach came with a number of problems ranging from single points of failure, scalability, censorship, availability.

The current design provides for greater scalability and availability, as the location of the descriptors is no longer centralized. If a hidden service directory goes down, there is still six others to obtain an descriptor from. This directory design also makes it difficult for an attacker to censor or track a hidden service. More detail can be found from the original proposal for Tor's hidden service directory design[9].

2.7.2 Descriptor publication

The hidden service first produces two sets of the same descriptor, which are the replicas. The descriptors differ by the descriptor ID and the resulting signatures. The descriptor ID is calculated as follows:

$$Descriptor\text{-}id = H(permanent\text{-}id\ |H(time\text{-}period\ |descriptor\text{-}cookie\ |replica))$$

The descriptor ID as denoted above is produced by applying a secure hash function H to the concatenation of multiple values. The permanent-id is a hash of the hidden services public key truncated after the first 80 bits. The time-period is the number of the period since the epoch . The descriptor cookie is an optional secret value shared between an the client and hidden service for authorized access. The replica number denotes which of the two replicas is being created. Replicas are used in order to place the descriptors into different parts of the distributed hash table or hash ring, which is described below.

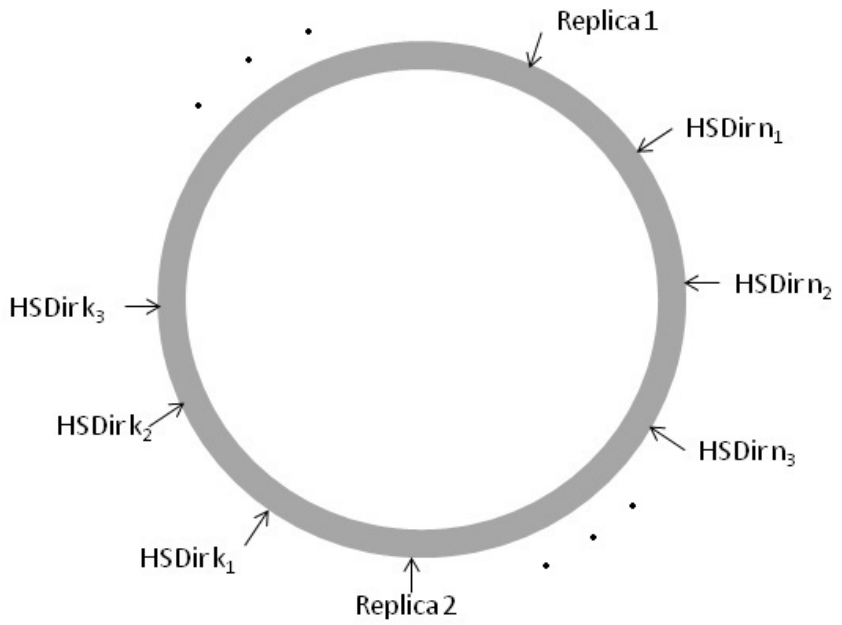


FIGURE 2.2: Conceptual view of the hash ring.

Once the replicas are created, the hidden service will publish each replica to a set of three hidden services. Which results in two sets of three descriptor publications. A

hidden service downloads the network consensus from the authoritative directories. It then filters out relays that do not have the HSDir flag and arranges them by the fingerprints. A relays fingerprint is SHA-1 digest of its public key. These fingerprints are then arranged in order to form a circle, or a hash ring. Going clockwise in the ring, the first three relays which have the closest fingerprint value to the replica's are chosen as the respective directories for that hidden service. This is done for both replicas. Figure 2.2 demonstrates this, from the first replica the directories n_1,n_2 and n_3 are chosen, where as the second replica gives the directories k_1,k_2 and k_3.

The reason why replicas of the descriptor are placed on the hash ring is to solve the issue of clusters on ring which result in large spaces before and between them. Having replicas on the ring which point to the same service has the effect of distributing servers more evenly across the ring. Which in turn results in greater availability.

Once the responsible directories are found, the hidden service creates a circuit to them and uploads the descriptor in the form of a POST message. If a directory already has a descriptor for the same service, it will only accept a new descriptor if its timestamp is not older than the previous cached descriptor. The directory must also receives verify the authenticity of any new descriptor before caching it. Verification comes in two forms, verifying the signature of the descriptor with the public key. Also generating the descriptor ID and comparing against the descriptor ID in question

Hidden services republish there descriptors in the following situations, incrementally every hour, whenever it detects that the responsible hidden service directories have changed and when parts of the descriptor are invalid, such as if an introduction point is no longer available[9].

2.7.3 Descriptor fetching

For a client to access a hidden service it must first obtain a valid descriptor. If it already has a valid descriptor cached for the service it will try it. If the descriptor is no longer valid, the client will fetch the descriptor from the responsible directories. The client first gets the network consensus, filtering out all relays that do not have the HSDir flag, then arranging the filtered relays by the relays fingerprint into a circle or hash ring.

The client will then locate the responsible directories by computing the descriptor ID's for the replicas and comparing them against the fingerprints on the hash ring. The free closest rings in the positive direction for a given descriptor ID are the responsible directories. As a result the client obtains six directories. The client chooses at random from the set of six, which directory it will fetch a descriptor from, this fetch comes in the form of a GET request. If for some reason a fetch fails, it will be removed from the set, and another directory will be chosen at random[9].

2.8 Hostnames in Tor

Tor uses the .onion host suffix for addressing its hidden services. It is a pseudo-top-level domain used only for hidden services. A site with an .onion address can be accessed through Tor browser software, or through a socks proxy. The domain format is a base32 encoding of the first eighty bits of the SHA1 hash of the identity key of the hidden service.

Other hostnames include .exit which refers to exit nodes and .noconnect which is a essentially a flag which signals tor to close a connection without attaching it to any circuit. However these won't be mentioned again as they are not relevant to this thesis[10].

2.9 Relavant work

2.9.1 Load balancing

2.9.1.1 Round Robin DNS

Round robin DNS, manages load distribution in relation to DNS queries. It supports load balancing for web services with multiple servers. A DNS server can maintain a list of IP addresses per service, in this fashion at each request, the next IP address in the list is given until the end of the list is reached, then cycled back around, thus distributing the load amongst all the servers[11].

Round robin has a few drawbacks that may be of relevance to us. Firstly, if one servers at a corresponding IP address on the list fails. The DNS system will still continue to serve it in its request, even if it is unreachable. Secondly, the scheduling is very primitive, it simply replies with the next address on the list, it does not take into account any other metrics, such as: bandwidth, response times, congestion etc.

2.9.1.2 OnionBalance

Here is a tool that tries to solve the same issue by providing randomized load balancing. OnionBalance relays requests to multiple Tor instances which points to the same hidden server. It works by providing a management server which modifies its descriptor to contain introduction points of the other Tor instances. Another benefit of having multiple Tor instances for the same hidden server is that reliability and availability is greatly improved[12].

The management server has its own Tor instance which essentially publishes a master descriptor. Several management serves can also be set up to better ensure reliability. This master descriptor is formed by combining the introduction points of the other Tor instances. The management server polls the responsible HSDir's for the descriptors of the back-end Tor instances. The introduction points can

then be extracted from the descriptors and combined and placed in the master descriptors. This master descriptor is then published.

Upon receiving a new descriptor, the management server will perform a number of checks to ensure the validity of the descriptor. These checks involve checking the signature and public keys against the Tor instance in question, checking the timestamps, to protect against replay attacks and confirming that the timestamp is at least not older than four hours, which might indicate that the tor instance is down.

Users can access the hidden service by using the hostname associated to the master descriptor. Which will provide them with the master descriptor which contains the mix of introduction points. The reason I referred to it as providing randomized load balancing is because the way in which the users onion proxy selects introduction points is random. Which results in a random distribution over the Tor instances.

Some clients could potentially not be able to reach the service if a malicious HSDir was to relay old instance descriptors that had introduction points that no longer exist. During a DoS attack an hidden service instance rotates the set of introduction points used faster, if this was to happen the management server may not be able to keep up with the rotation of the introduction points, which will result in invalid introduction points in the master descriptor. Also another limitation is that the onion address for the individual back-end instances is not hidden, so an HSDir can find it trivially.

2.9.2 Other possible approaches

2.9.2.1 Multitasking Architectures

The hidden server scalability of Tor can be solved in two ways, either we increase the hardware capabilities and resources, such as more memory, faster CPU's, or we can improve on its architecture which will allow it to make better uses of its

resources, resulting in a more cost efficient implementation. Below I will describe two common multitasking architecture used in web servers, which could possibly be used in Tor. Deciding what architecture would be an better fit depends on how much processing is required for a single client request and how much requests will have to be served.

2.9.2.2 Master Server Forking Architecture

The master server is the main server which will receive all the incoming connections. Whenever a new client request is made, the master server forks of a child process to handle the request. Clients make a request over a TCP/IP connection which the master server is prepared to handle. Upon receiving this connection request, the master servers establishes a connection and forms a new socket data structure.

After the master server will do a fork() system call to create a child process. The child process must know about the connection, so it must have access to the socket data structure created for that connection. The child process then serves the request until it finishes and then kills itself. At the same time the master process is still waiting for more request and following the same pattern of establishing connections and forking of child processes.

This approach is effective when requests do not end straight after they have been served. Meaning there is some sort of session state that needs to be kept alive. Which is the case with hidden services, Tor resets the connection every 10 minutes or in order to preserve security.

A issue with this approach is the potential bottleneck caused by the fact that the master server has to finish off creating a child process before it can accept any new requests. The master server creates a child process for every request, as a result this implementation can create large amounts of processes thus greater memory usage[13].

2.9.2.3 Pre-Forking Architecture

The master server architecture explained above forked child processes as requests came in. Pre-forking involves processes being forked ahead of time[14]. This removes the overhead of the using the fork command, as this can be an bottleneck for the entire system.

2.9.2.4 Apache Multitasking Architecture

Apache is an HTTP server, all its architectures are based on task pooling[15]. Task pooling as with pre-fork refers to the process of pre-initializing idle tasks (processes or threads) for later use. The master server is in charge of the amount of idle tasks and has control over the entire pool. Upon starting Apache creates a numbers idle tasks. Any request made to the server will be processed by one of these tasks.

2.9.2.5 Apaches Pre-forking architecture

Like generic pre-forking architecture, a master server creates a pool of child processes that sit idle, waiting to serve requests. With apache the master server doesn't serve requests itself but rather the child processes are registered with a TCP/IP communication server allowing the child processes to handle responses by it self[16].

The behavior and functions of the architecture can be categorized into the following phases:

- The start up phase
- The restart loop
- The master server loop
- The request-response loop

- The keep- alive-loop
- Deactivation phase

Apache has three start up phases. And each one has slightly different behavior. Which are When Apache starts up for the first time, when the server is restarted and when a child process is created.

The initialization steps generally involves: Memory management, pool management, Loading pre-linked modules, Loading command-line and set parameters, Loading configuration files, daemon related operations.

The restart loop, is initiated every time there is a force restart on the server. Its operations involve preparing the environment for the creation of child servers. The initiation of the master server main loop and additional operation that involve killing of idle child process and letting worker process complete.

The master server loop has two parts to it. The first part deals with the death of a child process. The second part is concerned with general pool management, monitoring child process. Killing of a process when there are too many and creating more when there is not enough.

The request response loop is run by child processes through the duration of its lifespan. Its role is to listen for and accept requests, the keep alive loop receives and responds to do those requests.

Chapter 3

Methodology

3.1 Overview

The following section will attempt to solve the issue of load balancing at the circuit level. The sections will observe running multiple Tor instances with the same hostname and private key, as well as how we can improve upon this. Described below are the main components used during the development, testing and experiment phases.

3.2 Testing and Experiments

3.2.1 Shadow

Shadow is a discrete-event network simulator that can run Tor[17]. It enables users to set up a private simulated Tor network, consisting of thousands of nodes. Shadow is used frequently in research purposes for controlled experiments, as it allows users to model network topologies, bandwidth and latency all all in virtual time. Shadow also allows users to create plug-ins that may interact with the specific application running inside the simulator. Due to these attributes Shadow is highly sought and required in regards to Tor research.

For Shadows Tor plug-in allows users to define network topologies and traffic generator configurations through XML files. Tor related files such as relay configurations can also be defined. At the end of the simulation Shadow produces detailed log files consisting of data on the activity of nodes such as CPU, memory usage, traffic generated and more.

The experiments demonstrated in the later sections of the thesis examine running multiple Tor hidden service instances with the same data. Performance is measured in these experiments by by monitoring the usage of hidden service relays. Shadow comes with useful tools that parse log data, which shows the bytes written and read by individual nodes across virtual time(ticks).All graphs generated in the following experiments are based on these files.

3.2.2 Chutney

Chutney is a tool used to bootstrap a private Tor network[18]. Chutney comes equipped with predefined templates for various types of relays used by Tor, such as directory authorities, hidden services and more. Chutney allows you to provide configurations that subsequently simplify setting up nodes on a network, by merely specifying the type of nodes needed and the quantity. Chutney also comes with some basic bandwidth testing tools, used mainly for small networks due to limitations. Chutney is utilized during some of the tests in the later sections for application testing.

3.2.3 Pros and cons

Shadow is used in the experiment as it provides a means of simulating Tor network as realistically as possible. It relies heavily on the amount of RAM available, whereas Chutney is relies on processing power. Furthermore, as a result , Shadow is more effective when needed to run large scale networks consisting of thousands of nodes. With up to 64 gigabytes of RAM , we can run experiments consisting

of hundreds of nodes within a few hours. Shadow also simulates latency, cryptographic and CPU processing delays. Making Shadow ideal for testing performance. A key limitation of Shadow is that it currently only supports up to Tor version 0.2.6.2-alpha, this is due to the way Tor handles threading in later versions.

One of the useful features of Shadow is that it allows you to create plug-ins that can be run within the simulation, however, these plug-ins have to be written in C. The program described in section 5 of the thesis utilises Pythons Stem library, which allows you to easily communicate with a Tor processes control port. Thus meaning it cannot run inside Shadow, as a result of this Chutney is used as it provides a means to test as well as monitor the behavior of the solution with relative ease. Chutney is used throughout the project for behavioral and functional tests.

3.3 Communicating with a Tor process

Tor's control port provides a means to communicate with a Tor process[19]. It relies on a message based protocol that provides a stream of communications between a subject and a controlling process. For it to work a Tor instance must have its control port set in its configurations, which specifies which port the process will listen for commands. Connecting to the port can also be authenticated through the control port, a subject can issue various commands to the process, such as to refresh its identity perform some descriptor fetches, publish descriptors etc. These commands are quite commonly used throughout the project, however, are not interacted with directly but done so through Python's Stem library.

Stem is a Python library that makes use of Tors control protocol, allowing a user to easily build against a Tor process[20]. A majority of the functionality of the program described in section 5 makes use of Stem for such tasks as downloading the networking consensus, posting/fetching descriptors and monitoring a hidden service.

Chapter 4

Multiple Tor instances with the same hostname/private key

4.1 Introduction

This particular section addresses the issue of scalability by running multiple Tor instances with the same host name and private key. This approach has been noted to work, it is made possible as the hidden services simply compete over which descriptors gets published by the HSDir's. The results show the hidden services each manage to serve requests.

This would address the issue of scalability on a multi core CPU, as each Tor instance is its own separate processes, the operating system will balance out the process across multiple cores as it usually does. This also means the hidden service instances can run at different locations. Setting up multiple Tor instances with the same hostname and private key is simple, both the hostname and private key files in the hidden service's directories must be the same.

4.2 Goals

The following section aims to present an understanding of the behavior, performance, and reliability issues that come with the approach described above. The experiments will be divided into two sections, load balancing and fail-over functionality. The load balancing experiments will measure how clients are distributed across multiple instances and how the number of instances used affects the performance of the system as a whole. The fail-over experiments will attempt to measure how traffic is redistributed when one instance fails, in order to measure reliability.

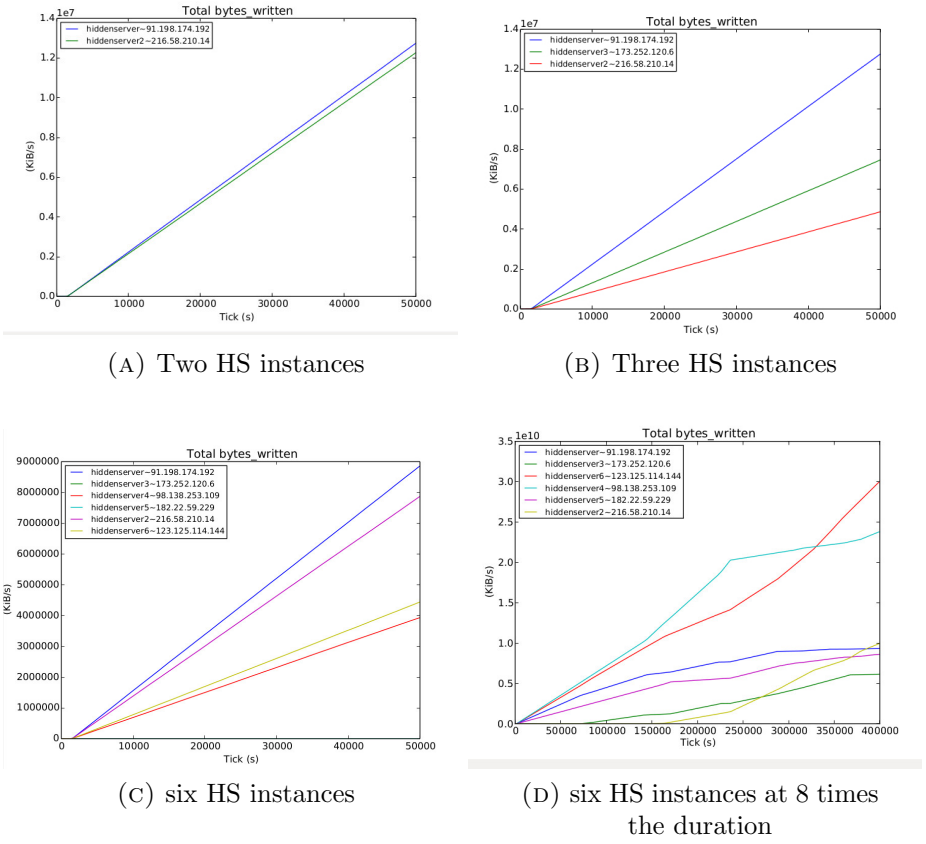
4.3 Experiments

4.3.1 Environment

The experiments will run under the Shadow simulator, the environment will be kept the same during all experiments and will run using an eight core machine with 64 gigabytes of RAM. All Shadow configurations contain the same amount of nodes except for where the hidden service nodes vary. Experiments run at a killtime of 50,000 ticks and take approximately six hours to complete. The network topology consists of one authority, 100 relay nodes and 300 hidden service client nodes. Client nodes are set up to start at random times across the start of the experiment.

4.3.2 Experiments Overview

Experiments will simulate a Tor network where clients repeatedly transfer a set amount of bytes to an .onion address. The load balancing experiments have all hidden service nodes running from start to finish, whereas the fail-over experiments will have the first hidden server fail at 20,000 ticks. For both cases, experiments



(A) Two HS instances

(B) Three HS instances

(C) six HS instances

(D) six HS instances at 8 times the duration

FIGURE 4.1: Load balancing experiments, cumulative bytes written

are run with two, three and six hidden server instances. The reason this is done is to see whether running more instances provides better performance and reliability or vice versa.

4.3.3 Load-Balancing

Figure 5.2 shows the cumulative total bytes written for the experiments with two hidden service nodes, it shows that the cumulative bytes written over time is roughly the same as the lines are close to each other. The first hidden services receives 51 percent of the traffic and the second receives 49 percent. Figure 2 shows the average bytes written every thousand ticks, the traffic split is constant throughout the experiment. This is promising in terms of performance, as it suggests that it is possible to attain some form of load distribution simply by running two hidden server instances.

However, with the experiments with three hidden server instances, the traffic split is not as even. The first hidden server gets the majority of the traffic at 50.8 percent, the second gets 19.4 percent and third hidden service get 29.7 percent. The final load balancing experiment as seen in figure 5.2 shows that when running six hidden service instances, only four receive traffic, the remaining two only receive a tiny amount at just above zero percent. Again a majority of the traffic goes to one of the instances. Comparing the times at which the instances publish their descriptors shows the first one will receive the majority of the traffic and the following will get the remaining traffic. However, increasing the duration of the experiment reveals that all instances start to receive traffic. Figure 5.2 D demonstrates this, the experiment runs at eight times the length, it shows that all instances can receive traffic at the same time, however the percentage of the traffic they receive is uneven.

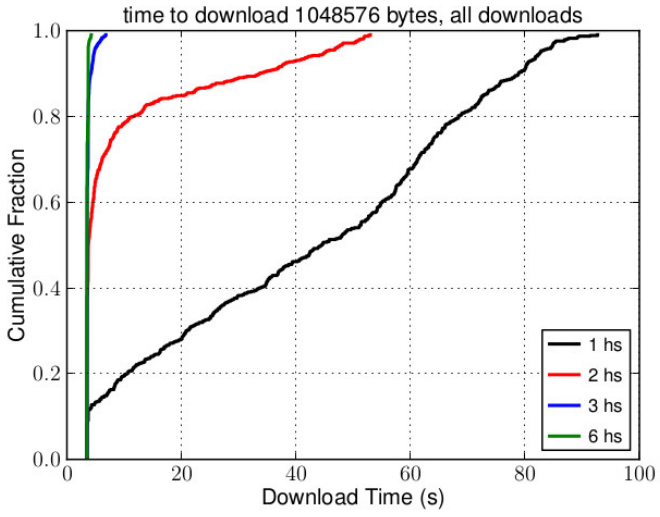


FIGURE 4.2: Time to download all downloads.

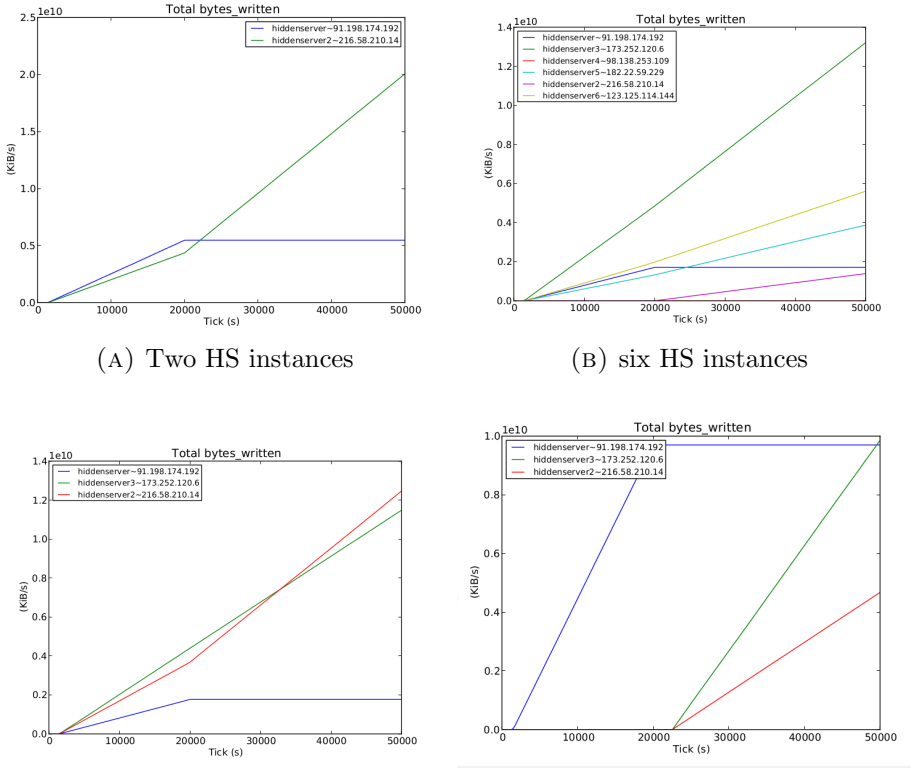
Figure 4.2 shows the results of another experiment which was conducted, where clients have a fixed number of bytes to transfer. The results show a comparison of experiments running one, two, three and six hidden services and how long it took to complete all downloads. When increasing the number of hidden service nodes the time it takes for all downloads to complete shortens. This therefore indicates running more hidden service nodes allows for clients requests to be served at a

greater rate. There is a significant performance increase when running two,three or six hidden service instances compared to running one.

4.3.4 Fail-Over

The fail-over experiments are configured in such a way that approximately half-way through the experiments one of the hidden services is shutdown. When this happens, in all three experiments, the clients which were originally connected to the hidden server that was shutdown, are left with an invalid route. After the hidden service goes down, the clients still carry on trying to communicate, but receives an introduction point failure. The clients then call a series of functions which check if any of the introduction points are usable, but none are as the hidden service was shutdown, as a result the descriptor is refetched. All previous clients of the hidden service that was shutdown are rerouted to another hidden service. The figures 4.3 shows the cumulative bytes written for experiments with two, three and six hidden services respectively, the flat line indicates that a hidden service has been taken offline.

However, if the hidden service instances had started at different times, the results would vary. Consider graph B at figure 4.3, this experiment was constructed such that, one of the hidden service instances started right after the other instances, as a result its descriptors are published right after the other, taking over the previous spots on the hash ring and receiving all the traffic. The responsible directories will all have the descriptors for that particular instance. When the instance fails, clients are unable to fetch a valid descriptor until one of the other instances republishes. This is made clear in graph B, as you can see that after the first instances fails, its takes around 2000 ticks before the other instances receive any traffic. In reality, this time would vary, as a hidden service republishes the descriptors every hour or when an introduction point fails.



(A) Two HS instances

(B) six HS instances

(C) Three HS instances

(D) Three HS instances with different timing

FIGURE 4.3: Fail over experiments

4.4 Explanation Of Results

This approach achieves load distribution due to a number of reasons. The first being that at any given time, the responsible HSDirs for an hidden service may not have the same cached descriptors. One HSDir may have the descriptor of hidden service instance A, whereas in comparison the other may have the descriptor for B. This is the case for the experiments explained above. Although all the hidden services start up and publish the descriptors at roughly the same time, the HSDir's will receive the descriptors in varying sequences, at each upload caching the most recent one. This is due to the process and order at which the hidden services uploaded the descriptors to the set of HSDir's which varies. The second is that clients decide at random which HSDir to query for a descriptor from a set of HSDirs.

Although the first point mentioned above may not hold true on a live network

or when the descriptors are not published at the same time. Consider the scenario where the hidden service instances were not started at the same time but ten minutes apart,which would mean that hidden service instances publish there descriptors at different intervals. As a result at any given time it is mostly likely that the set of responsible HSDir's will all have cached the descriptor belonging to the same hidden service instance, as there is no competing descriptor uploads. This will result in new clients going to the same hidden service instance until a new descriptor is uploaded. Furthermore, this approach will still work on a live network with real clients. Clients sessions may vary, some may use descriptors as long as they are valid, while others may reset the connection due to closing the browser, refreshing the identity (tor browser) etc. This means usage will still be somewhat split across different instances.

Chapter 5

Multiple Tor instances with mixed descriptor publishing

5.1 Method

5.1.1 Overview

Section four reviled that load distribution was possible while running multiple hidden service instances when the directories had descriptors belonging to different instances. This section improves on this by providing a solution that will help yield a more even load distribution and provide greater resiliency.

Load distribution is tackled by having a simple program sit in the background and republish the descriptors of each instance to the hidden service directories in such a way that descriptors are distributed across the six responsible hidden service directories as evenly as possible. This in effect provides a randomized load distribution for all new clients connecting to the service. This method is made possible as clients randomly pick of the six responsible directories, which one it will query for a descriptor first.

Providing resiliency is another important aspect of the method, section four revealed that when instances are publishing descriptors at different intervals, if an

instance goes down which was the last to publish its descriptors, all new clients will be unable to reach the service until the next descriptor publication. In the worst case this could be up to an hour. In order to achieve fail over functionality the program will regularly check the availability of each instance, if one appears to be down, a mix of descriptors will get republished to the directories containing only the descriptors of instances that are available.

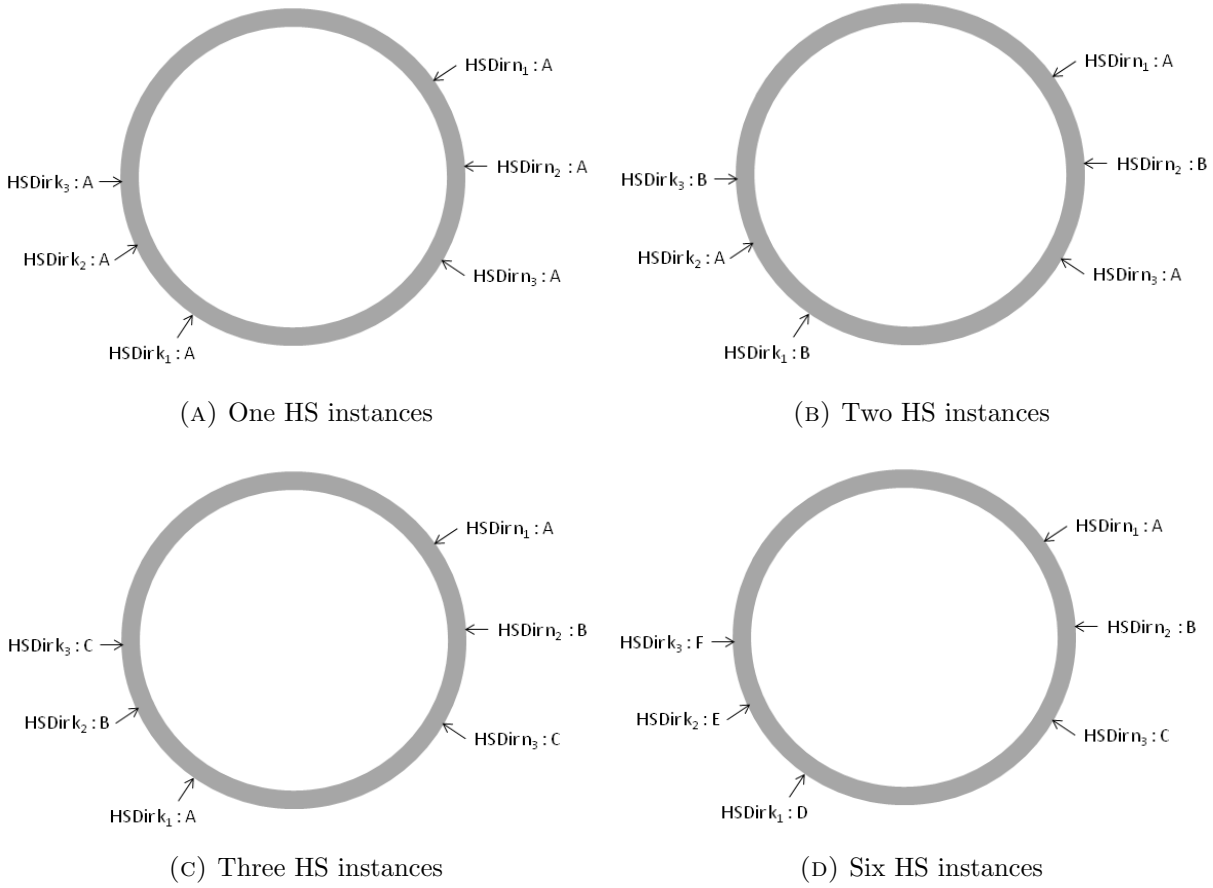
5.1.2 Changes to Tor

In order for the proposed method to work, the program must have access to the current descriptors used by each instances. The version of Tor used in regards to this thesis is, 0.2.7.2-alpha. This version does not have any features to accomplish this, however there is a proposal to-do so, that will be available in a later release, which can be viewed here. In order to make the program work, a simple modification was made to Tor where the descriptor is written to a file the program has access to, right before it is published. This modification was placed in the directory_post_to_hs_dir() function in rendservice.c.

5.1.3 Handling descriptors

Getting the responsible directories is rather simple, as described in the section 2.7, the program must first get the network consensus and filtering out all relays that do not have the HSDir flag. The remaining relays are sorted by the fingerprints and compared against the descriptor ID of the hidden service, the three closest relays in the positive direction are the responsible directories. This is done for both replicas, giving the program the six responsible directories.

Depending on the amount of instances being used, descriptors will be published as follows. If there is an even number of instances to directories, then descriptors are published evenly. With six hidden service directories, for a service running two instances A and B, half the directories will get A's descriptor and the remaining half will get B's descriptor. For a service running three instances, each descriptor



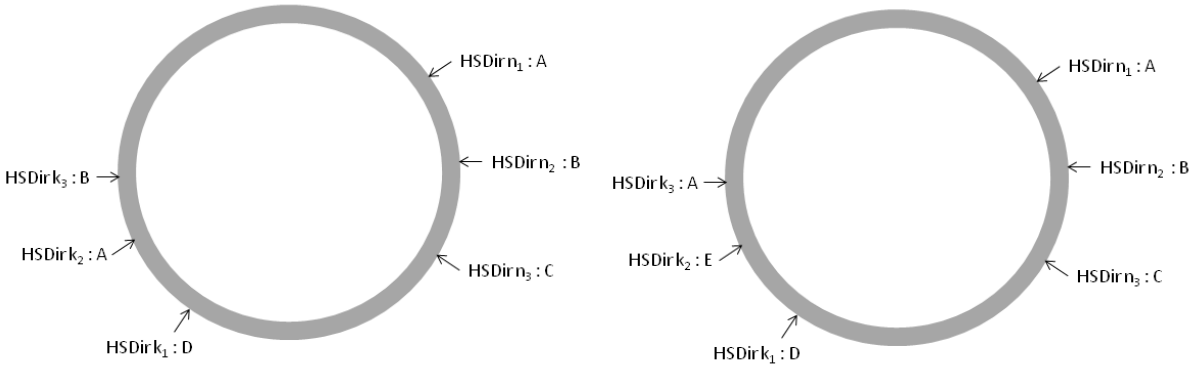
(A) One HS instances (B) Two HS instances

(C) Three HS instances (D) Six HS instances

FIGURE 5.1: Descriptor distribution

will get published to two different directories. For a service running six instances, each instances descriptor will be published to a different directory out of the set. Figure 5.1 shows how the descriptors will appear on the hash ring for a service using three instances compared to a service running one.

For services with an uneven amount of instances to directories the following applies. If there are four instances, each one will publish descriptors to different directories, and two of the instances chosen at random will publish a descriptor to the remaining directories. The same goes for five instances, one instances chosen at random will publish its descriptor to the remaining directory. This approach means there may be nodes that have a greater probability of receiving more clients. However for services running four or five instances, it would not be difficult to implement a method such that the program republishes descriptors at set intervals and at each time rotating which nodes or node will publish to the remaining directories.



(A) Four HS instances (B) Five HS instances

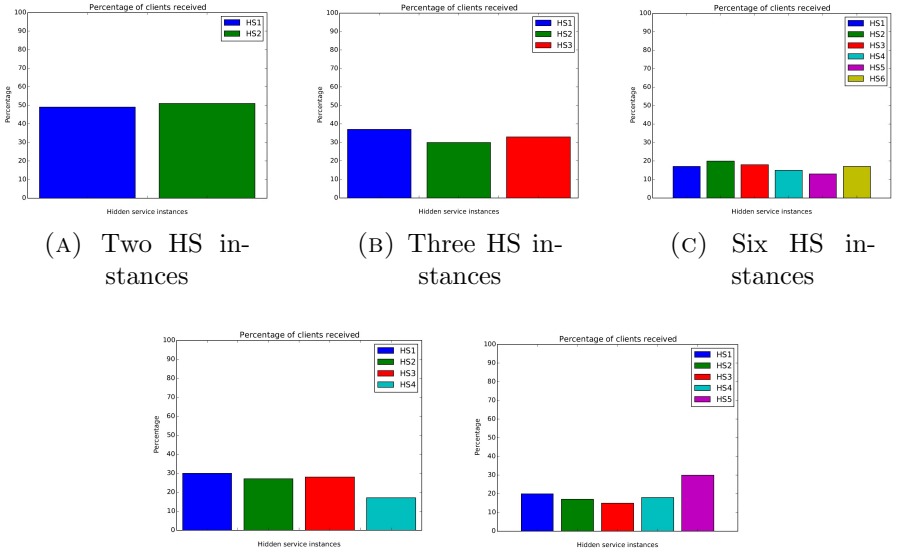
FIGURE 5.2: Descriptor distribution

Another important issue is when to republish descriptors. As this system does not involve modifying Tor to stop it from publishing it own descriptors, instances will still publish descriptors at set intervals, which will result in the instances taking over all the responsible directories with a new descriptor.

To tackle this, the program must publish the intended mix of descriptors right after any instances publishes its own, in order to bring the hash ring to its intended state. For this to happen the program must know when a descriptor is published, this is done by checking the files where the instances write there descriptors too every thirty seconds. If the descriptor in the file is different than the one cached by the program in anyway, it implies that a instances has just uploaded a new descriptor, as the only time the instance would write the descriptor to the file is right before it uploads it.

5.1.4 Fail over functionalities

As stated previously, an unreachable service occurs when none of the directories have valid descriptors. This will most likely occur in the situation where the last instance to publish its descriptors fails. However this would not be the case in this solution, as long as one instance is reachable then there exists a valid descriptor at one or more of the directories. But there may be situations where one or more



(A) Two HS instances (B) Three HS instances (C) Six HS instances

(D) Four HS instances (E) Five HS instances

FIGURE 5.3: Bar charts showing the percentage split between instances

of the instances fail, leaving directories with invalid descriptors for certain periods of time.

This is solved by having the program regularly check if the instances are up and running. Determining if a instance is running can simply be done by trying to connect to its control port. The program requires that the instances have the control ports set up with the necessary passwords provided to the program. Once the program determines that an instance has failed, it immediately republishes the descriptors of the active instances in the same manner as described above. The program attempts to restart the failed instance if possible, upon success republishing the next batch of descriptors containing the descriptor of the restarted instance.

5.2 Tests

For each experiment the hidden service instance was made to point to a unique web page. A client would then connect to the hidden service, depending on which web page was returned to the client, it was observable which instance the client was

forwarded too. In order to simulate a new client request, rather then creating many clients, the experiment used only ten clients, and refreshed the clients connection and identity before each new request, this would result in all cached descriptors being discarded and all circuits being reconstructed. In order to refresh the identity of a client a simple 'newnym' command was sent to the clients control port.

The experiments went on for a total of five hours and had each client perform a new request to the .onion address over a socks port every three minutes. Figure 5.3 shows the results of the experiments, all experiments resulted in a fairly even split between the instances. The most even being experiments running two, three and six hidden service instances, and the least even being running four and five instances. Experiments where also conducted to test the behavior under fail-over situations, however are not presented visually in this thesis. The tests simply shut-down random instances at random times, and observed the timing and behavior. The results showed the fail-over mechanism worked as expected, taking an average of a few seconds to fetch the new descriptor form the next HSDir successfully.

5.3 Evaluation

This method is limited by the amount of hidden service directories available, which means we can run up to six hidden service instances at once. It may be possible to use more than six instances by swapping instance descriptors in and out of the set of descriptors to be published but this may not yield as much of a performance benefit. However this is still quite a significant increase at it provides the possibility of scaling six fold, also providing for greater availability and reducing the likely hood of a single point of failure.

Another important point which was mentioned in the introduction, is the amount of stress which will be on the introduction points when larger services will migrate to Tor due to the amount of connections that will have to be throttled through them. This approach results in more descriptors with different introduction points being used and as a result the burden being spread across more introduction points.

The average descriptor allows for three to ten introduction points, simply using two instances could result in a maximum of twenty introduction points being used, with six instances that would be a maximum of sixty introduction points. This would make a significant difference, as the connections would be more evenly distributed across the introduction points, providing less stress on each individual introduction point.

Chapter 6

Conclusions

6.1 Future directions

The solution proposed looked at mixed descriptor publications, this solution was limited by the number of hidden service directories, allowing a service to use a maximum of six hidden service instances. In the background section we looked at OnionBalance, which provided load balancing by publishing a master descriptor containing a mix of introductions points of other back-end Tor instances. These two approach could be combined to produce a hybrid approach, where six different master descriptors could be published to different hidden service directories. This would result in the possibility of a maximum of sixty hidden service instances being utilised. It would be interesting to see the effects on performance for such an approach on large web services compared to a regular hidden service setup. It would also be interesting to measure the performance and reliability of such a setup against a realistic setup consisting of thousands of client requests. An interesting approach could be an modification to Tor which would allow for a master-slave setup. Where a master instance would be in charge of policing the creation and publication of descriptors. This would also bring up a number of questions such as, how would instances communicate with each other and how would private keys be shared. Another approach could be to modify Tor to allow

for a distributed hidden service set up, where multiple instances can be run, and the instances would vote on the creation of descriptors what introduction points they contain and how many different sets of descriptors are created.

6.2 Conclusion

Scalability can be addressed on two levels on the system hierarchy. One by modifying the Tor programs architecture, in the background section we looked at how multi-server architecture and more specifically how Apache handles multi-tasking by using a pre-forking approach, which would be one of many possible architectural changes that could be applied to Tor. The second being the main focus of this thesis, is by solving scalability at a circuit level. A solution was proposed where multiple Tor instances where ran with the same hostname and private key. An external program monitored the hidden service instances and republished a mixed batch of descriptors to the relevant hidden services directories. As a result randomized load balancing was achieved, allowing the use of up to six hidden service instances with the same hostname and private key. This approach also provides for addition resiliency properties.

Performance measurements where provided into the effects of using multiple instances, which showed a significant performance increase at the rate which clients where served. It was also explained that it was possible to potentially reduce the load on introduction points for large web services by allowing for larger amount of introduction points to be used which would result in traffic being distributed across a greater amount of introduction points. The solution was successful overall, providing significant yet limited amount of scalability, requiring little to no change to Tor itself. However for a more concrete and robust solution, there needs to be some significant changes to Tor architecture on both levels.

Bibliography

[1] The Tor Project. URL https://www.torproject.org/.

[2] Roger Dingledine, Nick Mathewson, and Steven J Murdoch. Tor: The Second-Generation Onion Router (2014 DRAFT v1). 1998. URL http://www.cl.cam.ac.uk/~sjm217/papers/tor14design.pdf.

[3] Roger Dingledine and Nick Mathewson. Tor Protocol Specification. . URL https://gitweb.torproject.org/torspec.git/plain/tor-spec.txt.

[4] Roger Dingledine and Nick Mathewson. Tor Path Specification. . URL https://gitweb.torproject.org/torspec.git/tree/path-spec.txt.

[5] Directory Authorties. . URL https://atlas.torproject.org/#search/flag:authority.

[6] Tor directory protocol. . URL https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt.

[7] Karsten Loesing and Nick Mathewson. BridgeDB specification. URL https://gitweb.torproject.org/torspec.git/tree/bridgedb-spec.txt.

[8] Tor Rendezvous Specification. URL https://gitweb.torproject.org/torspec.git/plain/rend-spec.txt.

[9] Karsten Loesing. Privacy-enhancing Technologies for Private Services. 2009. URL http://books.google.com/books?hl=en&lr=&id=JrEiIiaXjjUC&oi=fnd&pg=PA1&dq=Privacy-enhancing+Technologies+for+Private+Services&ots=gwh1RMdgVW&sig=zPvw9GFPRrqC91uj-BtwlLNw8oA.

[10] Nick Mathewson. Special Hostnames in Tor. URL https://gitweb.torproject.org/torspec.git/tree/address-spec.txt.

[11] T Brisco. DNS Support for Load Balancing. *RFC 1794*, 1995. URL http://freehaven.net/anonbib/cache/oakland2013-trawling.pdf.

[12] D O' Cearbhaill. Onion Balnace design documentation. URL https://github.com/DonnchaC/onionbalance/blob/develop/docs/design.rst.

[13] Multitasking server architectures. URL http://www.fmc-modeling.org/category/projects/apache/amp/4_3Multitasking_server.html.

[14] Michael Kircher and Prashant Jain. Pattern-Oriented Software Architecture, Patterns for Resource Management. 2004.

[15] Apache HTTP Server Version 2.2. URL http://httpd.apache.org/docs/2.2/mod/prefork.html.

[16] Munawar Hafizn. Secure Pre-forking - A Pattern for Performance and Security. 2005.

[17] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*, February 2012.

[18] Chutney. URL https://gitweb.torproject.org/chutney.git/tree/.

[19] Tor control protocol. URL https://gitweb.torproject.org/torspec.git/tree/control-spec.txt.

[20] Stem. URL https://stem.torproject.org/.