Secure, Anonymous Web-Hosting With Community-Driven Censorship

Scott Cunningham
B.A. (Mod) Computer Science

Final Year Project April 2014
Supervisor: Prof. Stephen Barrett
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

______________________________  April 23, 2014

Scott Cunningham
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

______________________________
Scott Cunningham

April 23, 2014
Secure, Anonymous Web-Hosting with Community-Driven Censorship

Abstract

Private, anonymous web-browsing is now more important than ever due to recent allegations of mass internet surveillance by large government intelligence agencies. However, the solutions that currently exist in this problem space take extreme anti-censorship positions, and as a result are left largely un-moderated. While censorship is often imposed by third parties and mis-used against the will of the users, it is sometimes useful in removing legitimately harmful content from a network. This project aims to design and implement an alternative private, anonymous web-serving network which allows for censorship decided by members of the community.
Contents

Title Page ................................................................. i
Abstract ................................................................. iv
Table of Contents ....................................................... v
List of Figures ........................................................... viii
Acknowledgments ......................................................... ix

1 Introduction .......................................................... 1
  1.1 Background ....................................................... 1
    1.1.1 Hierarchical Model of the Web ............................ 1
    1.1.2 Internet Censorship ........................................ 2
    1.1.3 User Privacy ................................................ 2
    1.1.4 Centralisation of Hosting ................................. 3
  1.2 Project Goals .................................................... 3

2 Related Work ....................................................... 6
  2.1 Blanchfield ..................................................... 6
  2.2 FreeNet ......................................................... 7
  2.3 Tor, The Onion Router ......................................... 7
  2.4 Kademlia ....................................................... 8
2.5 BitTorrent ............................................................... 9
2.6 Elite ................................................................. 10
  2.6.1 Query Anonymity .................................................. 10
  2.6.2 Human browsing habits as a content-rating system .... 11
  2.6.3 Recommendation“Ripple Effect” .............................. 11
2.7 Stack Overflow ...................................................... 11

3 Design ............................................................... 13
  3.1 Design Choices ..................................................... 14
    3.1.1 Author Anonymity .............................................. 14
    3.1.2 Reader Privacy .................................................. 15
    3.1.3 Data Storage .................................................... 20
    3.1.4 Replication ...................................................... 23
    3.1.5 Censorship ...................................................... 24
  3.2 System Design Overview .......................................... 27
    3.2.1 Web-page Retrieval and Verification ...................... 27
    3.2.2 Web-page Publishing .......................................... 28
    3.2.3 “Down-vote” Content Censorship Algorithm ............. 29

4 Implementation .................................................... 33
  4.1 The Python programming language ............................. 34
  4.2 Kademlia Implementations ....................................... 34
    4.2.1 Entangled ....................................................... 34
    4.2.2 “Kademlia” Library ........................................... 35
    4.2.3 PyDHT ............................................................ 35
    4.2.4 Choice of Kademlia implementation ....................... 36
4.3 Cryptography ......................................................... 36
   4.3.1 Key Derivation Functions & MAC ......................... 36
   4.3.2 Hashing function ............................................ 37
   4.3.3 Cryptographic keys ......................................... 37
4.4 Code Structure .................................................... 38
   4.4.1 Kademlia-based Peer-to-Peer Network .................... 39
   4.4.2 User-facing Front-end .................................... 43
   4.4.3 Demonstration Shell & Python Scripts .................... 44

5 Future Work ....................................................... 46
   5.1 Censorship ..................................................... 46
   5.2 Cryptography .................................................. 47
   5.3 Exploits ...................................................... 47
   5.4 User Interface ............................................... 48
   5.5 Partial File Replication .................................... 48
   5.6 Other Applications Based on this System .................. 49

6 Conclusions ....................................................... 50

Bibliography .......................................................... 53

A Code .............................................................. 56
List of Figures

3.1 Diagram demonstrating the use of a multi-hop peer-to-peer network to avoid user traffic tracking. Since user 100 contacts user 300 via user 200, no direct correlation between user 100 and user 300 can be made by a third party analysing user 100’s traffic. ........................................... 16

3.2 Diagram of the derivation of the content-location and cryptographic key from a web-page UUID. ................................................ 19

3.3 Diagram demonstrating the usage of random sender ID replacement used to hide the origin of a data request. ............................... 20

3.4 A diagram showing the process through which an secure cipher-text (encrypted) message and Message Authentication Code for that message is derived from a plain-text (un-encrypted) message. ......................... 23

3.5 A diagram showing the system architecture. ................................. 27

3.6 Algorithm for retrieving web-page from network. .......................... 28

3.7 Algorithm for publishing web-page to network. ............................. 29

3.8 Algorithm for calculating a web-pages new TTL, given its current rating and TTL. ................................................................. 32

4.1 Code listing demonstrating how an item is selected for deletion based on its downvote rating. ...................................................... 42

4.2 A diagram showing the system architecture, including the user-visible front-end. ................................................................. 44
I would like to thank the following people for their help over the course of this project (in no particular order):

- Prof. Stephen Barrett, who was a constant source of enthusiasm while supervising this project.

- Michael Clear, who was very helpful in giving me advice on cryptography.

- My girlfriend Izabela, who was patient and supportive throughout the project.

- My family for their support throughout my entire degree.

- Ian Hunter, who listened to my complaints and suggested ways to put them into writing.
Chapter 1

Introduction

1.1 Background

1.1.1 Hierarchical Model of the Web

The current model of the world-wide web has a hierarchical topology. Users typically connect to the internet via an Internet Service Provider (ISP). This gives Internet Service Providers the ability to analyse and block traffic which passes through them. This ability can be used for censorship of the users. Internet Service Providers typically use this ability to prevent their users from accessing certain web-sites.
1.1.2 Internet Censorship

Web censorship is often put in place by Internet Service Providers in response to pressure or legal threats from organisations such as governments or large media establishments. It is often the case that these organisations that impose this censorship on web users act with their own best interests in mind, and not always that of the users. This is particularly true in the case of governments blocking external media sources in an effort to maintain political or social goals that could be interfered with by media from other countries. Research done by the OpenNet Institute shows that internet censorship is done for political, social, and conflict/security reasons [8]. Since the validity of this type of censorship is subjective, we argue that web censorship of this nature is not justifiable and that the web should be free from this censorship.

It is not the case, however, that censorship is universally bad. In some cases, censorship is a viable means of removing harmful content from the internet. However, it is difficult for any third party to accurately define what content can be deemed objectively harmful for the users of a system. Therefore, we argue that censorship should be decided only by community consensus by the users of a system.

1.1.3 User Privacy

Since all user traffic passes through them, Internet Service Providers also have the ability to record and analyse user browsing habits. By analysing the destination IP address of web-browsing traffic, ISPs can identify web-sites that their users are visiting even when the content contained within the request is encrypted.
SSL certificates, used for encryption when serving a web-site over HTTPS rely on a centralised certificate-validation system. Due to the expense of SSL certificates, much web traffic on the Internet uses the un-encrypted HTTP protocol rather than encrypted HTTPS. This means that users web traffic is transmitted un-encrypted: without HTTPS, user passwords and web content can be read by malicious users on a shared network.

1.1.4 Centralisation of Hosting

Web-site hosting is usually centralised. It is typical for a web-site to be served by one single web server which serves content to all of that web-site’s users. This leads to a single-point-of-failure in web serving. Hardware or software failure at this server will lead to a loss of service for this web-site. Attackers that can take this single web-server down will result in a global failure of this web-site. This is often done through an attack called a Distributed Denial of Service (DDoS).

1.2 Project Goals

This project aims to design an alternate model for web content delivery on the internet. The model should aim to address privacy and anonymity issues in the current model. The project goals are as follows:

1. Decentralisation

The system should aim to avoid the centralisation that web-site hosting cur-
rently uses. Decentralising the system means that no resources will be controlled by one centralised authority, putting the system in the hands of the users themselves.

2. **Author Anonymity**

Users who wish to publish content on this system must be able to do so anonymously. Web content published to the system must not be traceable to the user that created it. Users, then, can not hold their own content: it must be co-operatively held by other users of the system in exchange for holding their content.

3. **Reader Privacy**

Users who wish to consume content on this system must be able to do so with privacy. It should be impossible for malicious users who intercept traffic on this system (either via a shared connection or by being their ISP, etc) should not be able to determine what content a user is consuming.

4. **Community-Driven Censorship**

Censorship is often controlled by large, centralised powers such as governments. Due to this, it is not always the case that web users are in control of what content has been made unavailable to users of the system. Users of the system should be in control of what content is or is not acceptable.

5. **Increased Reliability and Redundancy**

Web content stored on the system should not be reliant on any one single node. Content should be spread across nodes to ensure that it remains on the system
6. Secure Data Storage

Users can not all hold their own data because this would violate a user’s anonymity. Other users must hold it for them. As a result, we must ensure that the users storing your web content can neither tamper with it nor read its contents. For them to do so could possibly breach the authors privacy. This necessitates secure data storage. Furthermore, users that hold data for other users must not be able to be held responsible for the contents that other users have requested them to store. Therefore, it must be impossible for a user to know the contents of the data that they store for other users and what this data represents.

In this project, we also want to address the following questions;

- Can censorship be built into our system design?

- Can we design a system to enable democratic censorship by the users?
Chapter 2

Related Work

2.1 Blanchfield

Blanchfield, in his paper “An Anonymous and Scalable Peer-to-Peer System” [2], describes a novel anonymous peer-to-peer network.

Blanchfield defines five types of anonymity:

1. Author anonymity:
   A user should not be able to be linked to a document which they have created.

2. Publisher anonymity:
   A user should not be able to be linked to a document which they have added to the network.

3. Reader anonymity:
It should be impossible to tell which documents a user on the network is reading.

4. Server anonymity:
   A user should not be able to be linked to documents they store.

5. Document anonymity:
   A user should not be able to know what documents they store.

We will borrow these definitions throughout the paper.

### 2.2 FreeNet

In his paper “A Distributed Decentralised Information Storage and Retrieval System” [3], Ian Clarke describes a system which he calls “FreeNet”. FreeNet is a distributed, peer-to-peer web serving system providing document anonymity, author anonymity, and reader anonymity.

FreeNet does not provide any means for censorship. Instead, content on the network is given a “time to live” value. After this time expires, the content is removed from the network. This serves to remove unused content from the network.

### 2.3 Tor, The Onion Router

Tor is a peer-to-peer web serving network which provides user privacy against web traffic analysis. It aims to protect its users from malicious third parties analysing their
web traffic and using it to profile users or make assumptions about their browsing habits. It does this through multi-hop routing, which they refer to as “onion routing” [5].

Tor takes a strong anti-censorship position. As a consequence, Tor is left unmoderated and is known to be used for various illegal activities.

\section{2.4 Kademlia}

Kademlia [15] is a decentralised Distributed Hash Table (DHT). It provides a peer-to-peer, distributed key-value storage network. Data (the ‘value’) can be stored at a network index (called the ‘key’). Subsequent requests for this key will retrieve this value from users (nodes) on the network. Users are identified by a numeric user ID, and these users store data with keys that are numerically close to their own user ID.

Kademlia supports automatic data replication. Data is replicated to a number of nodes on the network depending on a global network constant value, called the ‘replication constant’. As users join and leave the network, content is automatically distributed to the users that have user IDs closest to the numeric ‘key’ value for data to ensure that data is not lost.

Kademlia sends queries through its peer-to-peer network in parallel, according to a global network constant value referred to as $\alpha$ (alpha). When a user receives a request for a certain key, $K$, it forwards the request to the $\alpha$ number of closest user
IDs that it knows of. In this way, messages intended to be sent to a certain user ID are flooded through the network until they reach their intended destination.

2.5 BitTorrent

BitTorrent is a distributed, peer-to-peer file-sharing system [4]. While BitTorrent is not specifically a web-serving network, it is a useful example of a popular decentralised data storage system.

On a BitTorrent network, users voluntarily choose to download content from other users (peers) on the network. Once portions of data have been retrieved from the network, this user can then provide that content to other users of the network.

In recent BitTorrent implementations, information about content locations are found using a Kademlia-based network, called KAD [17]. Though content locations in a BitTorrent network are found through the decentralised KAD network, data transfers are done directly from user to user instead of over the KAD network.

There is no redundancy or reliability built into the BitTorrent protocol. Content only stays on a BitTorrent network while users interested in sharing that content remain on the network.
2.6 Elite

"Elite" is an anonymous, decentralised search engine described by Kilian Levacher in his paper “Elite, An Ethical Peer-to-Peer Search Engine” [12]. Elite aims to allow users to hold their personal data, avoiding user data being held by third party search engine providers. Levacher aims to enable users to “democratically make decisions about how information is managed on the system”. This ideal is related to our goal for community-driven censorship.

Elite has several interesting design ideas that are of interest to this project:

2.6.1 Query Anonymity

To ensure query anonymity, Elite users randomly change the source user ID number to their own. Then, when that user receives a response to this query, they then forward the message to the original user that requested it. This means that any user that wishes to monitor traffic passing through the network can not accurately determine the true origin of a request for data.

We can use this technique in our system to enhance query anonymity in the network.
2.6.2 Human browsing habits as a content-rating system

Elite uses the browsing habits of its users to determine the rating of a page in the system. Levacher argues that pages that are often accessed by users are of a higher quality than those that are accessed less often. This allows Elite to use its users to rate the quality of web-pages.

We can adapt this technique to determine the quality of web-pages stored in our system also.

2.6.3 Recommendation “Ripple Effect”

Levacher describes the “Ripple Effect” as a way that recommendations can “ripple” through the system. When users want to recommend a web-page, they send a “ripple” recommendation message to their peers. If their peers agree with this recommendation, they can choose to forward it to their peers also, thus causing a “ripple” of recommendations being flooded through the system.

It may be possible to take inspiration from this idea when designing our community-driven censorship model.

2.7 Stack Overflow

Stack Overflow is a technical questions-and-answers web-site that features community-driven content rating. Stack Overflow is a web-site where users can submit questions
about a topic, and other users can answer these questions. The quality of answers is rated by users through a simple “up-vote” and “down-vote” system, where an “up-vote” is used to express that a user believes that this answer is helpful, and a “down-vote” is used when a user believes that an answer is inappropriate or incorrect [16]. The quality of an answer is assessed by its overall down-vote count subtracted from its overall up-vote count. Answers are then sorted according to this quality metric, and the answers are ordered by their score. In this way, the Stack Overflow community decides which answers are appropriate and most worthwhile, and their consensus decides whether answers are shown or hidden from view.

We can borrow this “down-vote”-based content-rating metric and adapt it to be used in our own system.
Chapter 3

Design

Having looked at other related solutions in the problem area and mentioning their interesting points and weaknesses, we now turn our attention to the design of our own solution. It was decided that we would design an alternate web-serving system based upon the Kademlia Distributed Hash Table (DHT). We will take inspiration from parts of several of the systems mentioned in the Related Work section.

Our design will consist of several users in a distributed system, which we refer to as “nodes”, each with the same capabilities. Each node will contribute to the network by serving some of the content that other users publish.

We will now discuss the reasoning and arguments behind design choices made in the architecture of this system, and then provide a description of system design and architecture.
3.1 Design Choices

In this section, we will discuss the design choices made to enable the goals of the system that were set out in the project introduction.

3.1.1 Author Anonymity

In order for an author of some web content to remain anonymous, they must not have any connection to that content. This introduces a number of complications for web serving. If a user does not want to be associated with content that they wish to publish, they can not hold that content. This means that content must be held by other users in the system.

Peer-to-Peer

A peer-to-peer computer network is a network in which all nodes contribute some resources to the network as a whole. These resources can be, for example, disk space, computing power, or bandwidth. The nodes on a peer-to-peer network all have the same functional capabilities in the network.

A peer-to-peer network was chosen for the underlying network of this system. Users of the system will contribute disk space and internet bandwidth to assist in the running of the network. This disk space and internet bandwidth will be used to provide other users access to web-page data on the network.
Users in the network will be identified by a numeric “node ID”. Data stored on the network will have a corresponding “content location”, which will be a numeric identifier. This “content location” will serve as an index into the storage space provided by users on the network. Users will store content which is numerically close to their “node ID”.

**How to Enable Author Anonymity**

Author anonymity can be achieved by distributing an author’s content to be stored by other users in the network. This way, users that wish to read content written by a certain author do not make any direct connections to the author. This removes the direct connection between a content consumer and author.

We will distribute web-content over a peer-to-peer storage system. This system will provide users with an interface to publish and retrieve web-pages.

**3.1.2 Reader Privacy**

**Avoiding Traffic Analysis**

Traffic tracking by ISPs can be done through analysing the destination address in an IP packet. To avoid this, the system should avoid direct connections between a reader and the user which holds the data that they are requesting. This is achieved through a multi-hop peer-to-peer network. This is used to avoid direct connections between a user that hosts content and the user requesting that content. With no
direct connections between the content provider and reader, malicious third parties that can analyse a user’s web traffic can not make accurate assumptions about what content a user is viewing based on the other users that they connect to.

There is a possibility of a compromise of reader privacy when a user receives the contents of a web-page which they wish to view. If a malicious third party (such as their Internet Service Provider, or a rogue user on their network) were to intercept a user’s traffic, they could potentially view the web-content that a user has requested. The malicious user could then use this information to violate the user’s privacy. To prevent this attack, we can encrypt the web content sent in responses to request for web-page data. Thus, any attacker that can intercept user traffic could only see encrypted data, and not the actual content that a user has requested.

Cryptography

Each web-page in the system consists of three components apart from its content:
1. A randomly generated unique identifier (called a UUID) which identifies this web-page in the system.

2. A “content location” which determines this web-page’s storage index in the system.

3. Cryptographic keys which will be used to encrypt and decrypt web-page data.

We will now briefly discuss these terms and their meanings.

**UUID**

A web-site’s UUID is a unique identifier of this web-page, known only to the users that wish to retrieve it. These UUIDs are used in the same manner as web URLs, but instead of being in a form similar to a traditional web-site (for example, www.example.com), they appear as a random string of alphanumeric characters. These identifiers serve only to identify the web-site and play the same role as a URL in the web as it exists today. These UUIDs could be shared outside of the system, or they could be linked from another web-page in the system such as a search engine or web-page indexing site, similar to the case with traditional URLs.

A web-page’s UUID is used to derive both its content location and cryptographic keys. These transformations are one-way, and it should not be possible to derive a web-page’s UUID from either its content location nor cryptographic key.

**Content Location**

A web-page’s “content location” is a numeric value which determines the ideal node ID in the network at which this content should be stored. The content location
Chapter 3: Design

is derived from a web-page’s UUID through a hashing function which transforms this unique identifier into numeric index in the network at which the content should be stored.

It is not guaranteed that a user with any given Node ID will be present on a network. As a result of this, the content should be stored at the numerically closest node or nodes to the content location identifier.

Cryptographic Key

In order to avoid compromise of user privacy through traffic analysis by malicious third parties, web content being sent to users on the network should be encrypted. The encryption of this data requires a cryptographic key.

It would be inconvenient for users of the system to hold a database of cryptographic keys relating to web-sites. It would also be complicated for users to have to manually send cryptographic keys when sharing UUID links to other users. To avoid this, we can derive cryptographic keys from the web-page’s own UUID.

A Key Derivation Function is a cryptographic function that takes an arbitrary string and can deterministically generate a cryptographic key from it [10]. Using a Key Derivation Function, we can generate cryptographic keys to encrypt and decrypt web-page content using only a web-site’s UUID. When sending a request for a web-page, a user must only reveal a hash of its UUID (in order to find its content location). This means that the only information about user a request that is leaked to the network is the content location of that web-site. Since it is not possible to derive either the web-page UUID nor the web-page encryption key from this content location,
it is impossible for other users on the network to determine what content a user is consuming or what UUID this encrypted data represents.

Figure 3.2: Diagram of the derivation of the content-location and cryptographic key from a web-page UUID.

**Random Sender ID Replacement for Query Anonymity**

It is possible for a malicious user to join the network with the intention of analysing the sender and destination ID fields in packets that they route through the network. All content request packets routed through the network include both a sender ID and content ID field.

In Levacher’s paper on “Elite” [12], he introduces a novel way of addressing this issue which we will borrow in this system. All users, when passing a request on towards the destination, can randomly change the sender ID field in that message to their own user ID. When this user receives a response to this modified request, they can simply pass the response on to the user that originally requested it. This means
that users in the network can not reliably tell the real origin of a request, meaning that none of the users in the system can accurately record the activity of other users. This concept is demonstrated in figure 3.3.

![Diagram demonstrating random sender ID replacement](image)

Figure 3.3: Diagram demonstrating the usage of random sender ID replacement used to hide the origin of a data request.

### 3.1.3 Data Storage

In order to preserve document anonymity, a user holding some data must not know what the data that they are storing represents. In the case of this system, this means that a user must not know the UUID of said data. It is also necessary for a user to have no knowledge of the contents of this data, so that they can not be held responsible for the actions of other users of the system.

Since content can not be held by its author in order to provide author anonymity, it is important that the legitimacy of stored data can be verified by the users of the system. It must be possible for users to verify that the data which they receive has neither been tampered with by the user holding it nor the users in the system that delivered the content to them. We will refer to this guarantee as “content integrity”.
In this subsection, we will address and discuss these requirements as well as demon-
strate techniques to provide them.

**Document Anonymity**

When a user is requested to hold a piece of data, they are told the content ID, and the encrypted web-page data to store.

The content ID of a piece of data is derived from the UUID of that data. To ensure that the UUID of a piece of data can not be determined, it must be impossible to derive a UUID from a content location given to a user. Therefore, the hash function used to derive a content ID from a UUID must be sufficiently complex that it would be computationally infeasible to derive a UUID if given a content ID [6].

The data to be stored is encrypted by a cryptographic key derived from the UUID of that data. The only way to determine the contents of that encrypted data would be to decrypt it with the cryptographic key. This cryptographic key is derived from the UUID using a Key Derivation Function.

Since it should be computationally infeasible to derive the UUID of a piece of data from the content ID, it is impossible for a user to determine either the contents of a piece of stored data or the web-page UUID which it represents.
Content Integrity

In order to guarantee content integrity, the system must provide a means of verifying that content stored in the system has not been modified since being generated by the author. We can achieve this goal by using a cryptographic technique called a Message Authentication Code (MAC).

A Message Authentication Code is a cryptographic technique used for message authentication - that is, it is used to verify the authenticity of a message [11]. Given a message M and cryptographic key K, performing a MAC operation on K and M will generate an authentication code C. If a user modifies M, it is computationally infeasible for a user that does not hold the secret key K to compute a new MAC for this modified message. Thus, if a user receives a response to a data request and the MAC check does not verify the integrity of the contents of the received message, then the content must have been tampered with since its creation by the content’s author and is not reliable. This process is demonstrated in figure 3.4.
3.1.4 Replication

In order for the system to provide increased reliability and redundancy, data should be replicated in the network. This will provide resilience against single node failure due to network problems, disk failure, or other node failure. By adding replication, we aim to remove the single-point-of-failure for a web-page, providing increased reliability.

Kademlia networks provide automatic content replication \cite{15}. In a Kademlia network, there is a global replication constant among all nodes which determines number of replicas that are kept for any one piece of data. When some node $N$ leaves the network, the other nodes that hold the replicas of $N$'s data automatically share the data that $N$ held to the closest available nodes on the network.

We use Kademlia’s built-in replication system to provide full document replication.
in the network.

3.1.5 Censorship

To provide community-driven censorship, the system must allow the users to decide on the censorship of content on the system.

We will now discuss the difficulties associated with community consensus in a decentralised network and explain the design choices made in this system to provide community-driven censorship.

Decentralisation

Since in this system there is no centralised service providing access to web-pages, access to web-pages can not be controlled in one place. Censorship on a web-page can also not be decided by the author of some content as this would enable users to never have their content censored by the community. We propose that the censorship of web-pages should be done by the users that hold that web-page data.

User-based Content Evaluation

In “Elite”, Levacher claims that users tend to browse web content that they are generally interested in. He also argues that if a page is frequently viewed, then it must be of high-quality.
Levacher also makes an argument for a “Zero-Input Recommendation System.” Levacher’s argument is that evaluating the quality of content is most effective if users must exert no effort to do so. We argue then that if a user visits a web-page that they do not object to, then the web-page must be of high quality. This implies that they agree with the content of the web-page and do not deem it unsuitable for the community. Furthermore, if a user does visit a web-page that they object to, then they are likely to want it to be removed from the system. These users must be given a means to voice their objection to this web-page.

In light of this, we propose a mixture of Levacher’s “Zero-Input Recommendation System” combined with a channel for users to report web-pages as unsuitable. We will allow users to send a notice of objection to a web-page, which we refer to as a “down-vote”. This phrasing is in the style of Stack Overflow [16]. If a user views a web-page and does not send a ‘down-vote’ about this page, then the system assumes that a user does not object to the content and that the web-page is of high quality.

“Down-vote”-Based Censorship

In our system, we propose that censorship be decided by the aggregate “up-vote” minus “down-vote” value, where “up-votes” are implicitly made when a user does not object to some content. We will now propose a novel algorithm for the removal of content based on this concept of “down-votes”.

Data in FreeNet is automatically pruned from the system once is not accessed in some time [3]. This serves to remove old and unused content from the network. We
will borrow this idea in our system for the same reason. Web-pages will be given a time-to-live value (TTL), which determines how long until they will be deleted from the system. Web-page accesses (implicit “up-votes”) will increase this TTL, whereas “down-votes” will decrease this value, meaning that content will be removed from the network sooner.

Problems Caused by Kademlia’s $\alpha$-Parallelism

Since the Kademlia network sends messages in parallel to the $\alpha$ closest nodes to a particular destination user ID, it is entirely possible that a user may receive a single down-vote request twice. This is made worse by the random sender ID replacement that nodes on the network do to preserve reader anonymity. This makes it impossible for a user to tell whether they have received a down-vote request more than once. If we never performed sender ID replacement to down-vote requests, it would compromise reader privacy for the user sending the down-vote request.

This is mitigated by adding an extra field to down-vote messages: a “down-vote-UUID”, which is a randomly generated identifier with which a user can identify a particular down-vote request, and still perform random sender ID replacement on down-vote requests. We argue that these do not compromise reader anonymity as there is no user-identifying information present in the down-vote-UUID. If users receives several down-vote requests with the same down-vote-UUID, they can safely ignore it.
3.2 System Design Overview

![Diagram of system architecture]

Figure 3.5: A diagram showing the system architecture.

The system consists of several users (“nodes”) on a Kademlia network. There are three main actions that a user can perform, with the necessary parameters in brackets:

3.2.1 Web-page Retrieval and Verification

To retrieve a web-page for a given UUID, the user must derive the content location from the UUID using the hash function, and encryption key and MAC key from the UUID using the Key Derivation Function. They must then retrieve the encrypted
Algorithm 1 Retrieve web-page from network, given parameter “UUID”

\[
\text{content location} \leftarrow \text{hash}(\text{UUID})
\]

\[
\text{encryption key, MAC key} \leftarrow \text{key derivation function}(\text{UUID})
\]

\[
\text{actual MAC, encrypted content} \leftarrow \text{kademlia lookup}(\text{UUID})
\]

\[
\text{expected MAC} \leftarrow \text{generate mac(encrypted content, mac key)}
\]

\[
\text{if expected MAC} \neq \text{actual MAC then}
\]

\[
\text{return error}
\]
\[
\text{end if}
\]

\[
\text{decrypted content} \leftarrow \text{decrypt(encrypted content, encryption key)}
\]

\[
\text{return decrypted content}
\]

Figure 3.6: Algorithm for retrieving web-page from network.

data and MAC from the Kademlia network. They then calculate the expected MAC and compare it to the MAC of the received data. If they do not match, the data has been tampered with. Otherwise, the user can decrypt the data and read it. This is demonstrated in figure 3.6.

3.2.2 Web-page Publishing

To publish a web-page, the user must generate a random UUID. From the UUID, the user then derives the content location using the hash function, and the encryption and MAC keys using the key derivation function. They then generate a MAC of the data and the key, and store both of these values at the given content location in the
Algorithm 2 Publish web-page to network, given parameter “content”

\[
\begin{align*}
\text{UUID} & \leftarrow \text{generate\_UUID()} \\
\text{content\_location} & \leftarrow \text{hash\_function}() \\
\text{encryption\_key, MAC\_key} & \leftarrow \text{key\_derivation\_function}() \\
\text{encrypted\_content} & \leftarrow \text{encrypt(content)} \\
\text{MAC} & \leftarrow \text{generate\_mac(encrypted\_content, mac\_key)} \\
\text{kademlia\_store\_value(content\_location, encrypted\_content, MAC)} \\
\text{return } \text{UUID}
\end{align*}
\]

Figure 3.7: Algorithm for publishing web-page to network.

Kademlia network. This is demonstrated in figure 3.7.

3.2.3 “Down-vote” Content Censorship Algorithm

“Down-vote” messages are passed through the network in the same manner as data retrieval requests. This new request is not present in the Kademlia specification, and requires modification to any current Kademlia implementations. The “down-vote” Kademlia message simply contacts the nodes holding the data for a given UUID by using a modified version of the Kademlia content-retrieval remote procedure call (RPC). The user determines the content-location of this UUID by using the hash function, and then sends this “down-vote” RPC to the nodes responsible for this content location.
Content “down-votes” are handled by users storing the data on the network. Users hold three pieces of information about each piece of data they store:

1. The data itself.
2. The aggregate rating of that data, calculated from the number of retrievals minus the number of down-votes that the data has received.
3. A “time-to-live” (TTL) value, which represents the time in seconds until the content will be removed from the network.

When a user receives a down-vote request for a piece of data they are storing, they increment its down-vote counter. Once every second, every data item’s time-to-live value is decreased by a function of their aggregate rating. While an item’s rating is above zero (that is, that its number of up-votes is larger than its number of down-votes), its TTL does not expire, meaning that highly-rated and often-accessed pages are kept on the network. Conversely, low-rated pages will eventually be dropped from the network.

**Abuse**

Since down-vote messages are designed to be anonymous, it is possible for users to attempt to cheat the system and down-vote some content repeatedly until it is removed from the network. This can be stopped in the logic of the user’s application, but this would be an unsatisfactory solution as it would be trivial for the user to circumvent this by modifying the source of their application, or building a customised
client. As a result, there is a trade-off between absolute anonymity and accurate censorship. We aim to mitigate abuse somewhat in the down-vote algorithm that we use, but our choice was to prioritise anonymity over the possibility for abuse.

To counteract possible abuse by users of the system, our down-vote algorithm decreases a web-page’s time-to-live only logarithmically as its rating decreases. One per second, we decrease each web-page’s time-to-live by a function of its aggregate rating. This function takes the logarithm of the aggregate rating and subtracts that value, in seconds, from that value’s TTL. As a result, pages with few aggregate down-votes will decay from the network at a slower rate than those with very large aggregate down-votes. A listing of the algorithm can be found at figure 3.8 for clarity.

Since content is replicated across several nodes on the network by Kademlia, a malicious user would need to send sufficient down-vote requests to all replicas responsible for some content to completely remove it from the network. At every second, if a web-page’s aggregate score is below zero, then the TTL is decreased by a logarithmic function of the aggregate score of that web-page, given as:

\[ \text{TTL} \leftarrow \text{TTL} - \log_2(\text{absolute value}(\text{aggregate score})) \]

Thus, the number of down-votes required to remove this content from the network from one replica immediately (assuming that all requests hit the same replica) would be: \(2^{\text{TTL}}\), as \(\text{TTL} - \log_2(2^{\text{TTL}}) = \text{TTL} - \text{TTL} = 0\) Since data is replicated in the network by a replication factor, this number becomes \(2^{\text{TTL}} \times R\) where \(R\) represents the number of replicas serving that piece of web content. This means that content that was recently added to the network (thus having a high TTL), will be difficult for one
Algorithm 3 Calculate new TTL for a web-page, given parameters ‘rating’ and ‘TTL’

```plaintext
if rating ≥ 0 then
    return TTL
end if

score ← absolute_value(rating)

TTL\_delta ← log2(score)

return TTL − TTL\_delta
```

Figure 3.8: Algorithm for calculating a web-pages new TTL, given its current rating and TTL.

user to remove from the network by virtue of the large number of down-vote requests that they would have to make. This also assumes that no other user is accessing the content: indeed, if other users are accessing this web-page and not down-voting it, more requests will be required.
Chapter 4

Implementation

Now that we have discussed the design of our system, we will discuss its implementation. This design will consist of two main parts:

1. Identical Kademlia user nodes which will make up the Kademlia network. The code for these nodes will contain implementation for content look-up, publishing and down-voting and will provide a high-level interface for interfacing with the system.

2. A user-facing front-end which will have the ability to look-up content, publish content and down-vote content.

We will first discuss general implementation details, then turn our attention to the implementation of these parts.
4.1 The Python programming language

The project was written using the programming language Python. Python is a high-level interpreted programming language with light syntax and “duck-typing”, making it ideal for rapid development. Python also has many useful libraries available, many of which are open-source under permissive licenses. While none of these factors were essential to the development of the project, the availability of Python libraries helped with the implementation of the system.

4.2 Kademlia Implementations

While there are several available Python implementations of the Kademlia DHT, many are incomplete or badly documented. We will discuss the available implementations and their relative advantages and disadvantages.

4.2.1 Entangled

Entangled is a working implementation of the Kademlia DHT written in Python. It provides a high-level programming interface and a graphical interface for interfacing with the Kademlia network with the ability to demonstrate file-sharing graphically. Entangled also includes an extra “delete” RPC which could be easily modified to serve the purpose of the “down-vote” RPC that our system requires. However, Entangled lacks comprehensive documentation and has a complicated API. Entangled’s code-
base is complicated and difficult to understand in places. For this reason, it was
decided that Entangled was not suitable to be used for this system.

Entangled is released under the GNU General Public License, v3 (GPLv3).

4.2.2 “Kademlia” Library

“Kademlia” is a Kademlia DHT implementation in Python. “Kademlia” seems
to be well-documented and written, but as its first release was published after the
beginning of this project, it was decided that it would be too unstable of a platform
to base the project on. However, the library looks promising and it would be possible
for it to be used in future projects once the code-base has become stable.

“Kademlia” is released under the MIT license.

4.2.3 PyDHT

PyDHT is a pure Kademlia DHT implementation written in Python. PyDHT
does not add extra functionality to the DHT, and its code is simple enough to be
understood, despite its lack of documentation. The code-base of PyDHT is small and
modular, making it possible to easily change the implementation in places. PyDHT
includes simple code examples demonstrating how to use the library. As a result, it
was decided that PyDHT would be a suitable base to work on for this project.

PyDHT is released under the BSD 2-clause license.
4.2.4 Choice of Kademlia implementation

It was decided that PyDHT would be used as the base of this project due to its stability, simple API and permissive license. Since PyDHT is licensed under the 2-clause BSD license, we have forked the library and the modified library will be made available under the same 2-clause BSD license. The modified library will be made available in Appendix 1.

4.3 Cryptography

4.3.1 Key Derivation Functions & MAC

The implementation of key-derivation functions and MAC used in the project is written using the PyCrypto Python cryptography library [13]. The code for this implementation is based on PyCrypto code examples released in to the public domain by Brendan Long [14]. PyCrypto was chosen due to its proven stability and usage as well as it having functional examples of Key-Derivation Functions.

The key-derivation function chosen to be used for this project was PBKDF2. PBKDF2 was written by RSA labs to replace an earlier standard, PBKDF1, which could only produce keys with length of up to 160 bits [10]. PBKDF2 is suitable due to it supporting the generation of larger keys than 160 bits. This ensures that key-sizes can be increased beyond 160 bits in the future if necessary. PBKDF2 was chosen due to a readily available implementation being included in the PyCrypto
library. Other key-derivation functions that supported large key-sizes would also be suitable for usage in this system.

The MAC algorithm which was chosen to be used for this project was Hash-based Message Authentication Code (HMAC). HMAC was chosen due to a readily available implementation being provided in the PyCrypto library and due to it being well-known and proven in the field. Other MAC algorithms would also be suitable in its place.

4.3.2 Hashing function

The hashing function chosen to be used to derive content locations from UUIDs in the system was SHA-1. This could be replaced with any hashing function that provided a relatively large key-space. SHA-1 was chosen due to an implementation in the Python standard “hashlib” library.

4.3.3 Cryptographic keys

Cryptographic keys are necessary to provide encryption and decryption of data stored by nodes on the network. In order to ensure document anonymity, the encryption scheme used must be strong enough that it would be infeasible for a node on the network to break it in a reasonable amount of time. The user that publishes some data must be able to encrypt that data, and also allow users to decrypt it.

During the design of the system, we considered both symmetric key encryption
and asymmetric key (public-private key) encryption.

Asymmetric-key RSA encryption was initially used for encryption and decryption of content. In RSA, a private-key is used for decryption and a public-key for encryption [9]. Since the system requires that the author encrypts a piece of data and readers decrypt it, the private key would have to be given to users to decrypt web-page data. As a result, asymmetric-key encryption was decided to be unsuitable for the system.

Symmetric-key encryption uses one cryptographic key for both encryption and decryption. This fits the requirements of our design specification - the content author and content reader can both use the same cryptographic key for encryption and decryption. This implementation of the system uses Rijndael AES as its implementation of symmetric-key encryption due to its availability in the PyCrypto library [13]. Rijndael AES was arbitrarily chosen as it has been proven by its industry use, and had no obvious disadvantage compared to the other algorithms.

### 4.4 Code Structure

The code is split into two major components, as mentioned previously:

1. A Kademlia-based peer-to-peer network. Since the project uses the PyDHT Kademlia implementation library as a base, most of the code is contained in a modified version of PyDHT. This part of the code will provide a Python interface to publish, retrieve and down-vote web-pages as well as handling communication
with the underlying Kademlia network.

2. The user-facing front-end code. This part of the code will provide a user interface which will allow a user to publish, retrieve and down-vote web-pages on the network. This part of the code-base be composed of two parts:

   (a) A Python-based program which will relay messages from the user interface to the underlying modified Kademlia network.

   (b) A web-browser-based application which provides a user interface to the network. This will be written in HTML and JavaScript, using web-sockets to communicate with the Python program which in turn communicated with the Kademlia network.

We will now continue to discuss the implementation of both of these components.

4.4.1 Kademlia-based Peer-to-Peer Network

This part of the code is heavily based upon PyDHT, which provides a working Kademlia network as a base on which we build the extra features needed by our network.

Content Publishing - publish(content)

Content publishing is done according to the algorithm described in figure 3.6, in the Design section of this paper. This is done in the publish() function of the
DHT class in the source file pydht.py. The function generates a random UUID using
the standard Python “uuid” library, then derives the content-location using SHA-1
as the hash function. The function then uses the do_encrypt() function from the
key_derivation module to encrypt the data, using the UUID, then stores it in the
Kademlia network and returns the UUID.

**Content Retrieval - retrieve(uuid)**

Content retrieval is done according to the algorithm described in figure 3.7, in
the Design section of this paper. This is done in the retrieve() function in the DHT
class in the source file pydht.py. The retrieve() function takes the given UUID, and
uses the hash function to derive the content-location. The function then retrieves this
content from the Kademlia network, and tries to decrypt it using the do_decrypt() function from the key_derivation module, using the UUID. If either the decryption
or MAC checking fails, then the function throws an error. Otherwise, the decrypted
data is returned to the user.

**Encryption & Key Derivation**

The encryption and key-derivation logic of the system is contained in a key_derivation
Python module which was added to PyDHT. The encryption and decryption func-
tions written in this module are based on code examples released into the public
domain [14].

This module is comprised of the following functions:
• **make_key()**
  This function takes a UUID and uses a key-derivation function to generate a cryptographic key. This key can then be used to as an AES or HMAC key.

• **make_hmac()**
  This function takes a message and a cryptographic key and uses the key to create a HMAC verification code of the message. This HMAC verification code can then be used for message authentication.

• **encrypt()**
  This function takes a message and a cryptographic key and uses the key to encrypt the message using AES.

• **decrypt()**
  This function takes an encrypted message and a cryptographic key and attempts to decrypt the message using the key. An error is thrown if this is unsuccessful.

• **do_encrypt()**
  This is a simple wrapper function that takes a UUID and a message and returns an encrypted message and HMAC verification code. This function uses make_key to generate an AES key and a HMAC key using the UUID, then encrypts the data and generates a HMAC verification code.

• **do_decrypt()**
  This is a simple wrapper function that takes a UUID and encrypted message and attempts to decrypt and verify it. The function generates a HMAC key and an AES key using make_key, then attempts to decrypt and verify the encrypted
data using them. If the verification or decryption fails, the function throws an error. Otherwise, decrypted data is returned.

“Down-vote” Content Censorship Algorithm

Down-vote handling is done in the logic of the DHT class. The DHT class keeps a record of the TTL and aggregate down-vote count for each of the pieces of data that it holds. Once per second, the `tick()` function is called, which adjusts the TTL of items according to their rating. Once an item’s TTL is less than zero, it is deleted from the network. Figure 4.1 shows a code listing taken from the `tick()` function from the program source, demonstrating the down-vote algorithm’s implementation.

```python
for (uuid, rating) in self.webpage_ratings.items():
    if rating < 0:
        downvote_val = math.log(rating, 2)
        self.ttls[uuid] -= downvote_val

for (uuid, ttl) in self.ttls.items():
    if ttl <= 0:
        print "UUID", uuid, " past TTL - deleting"
        del self.data[uuid]
```

Figure 4.1: Code listing demonstrating how an item is selected for deletion based on its downvote rating.
4.4.2 User-facing Front-end

As mentioned previously, the front-end is made up of a JavaScript and HTML web-based front-end, and a Python program which interfaces with the modified Kademlia network. We will now discuss both of these.

**Browser Code - websocket-client.html**

The browser code consists of a simple HTML page which provides inputs for users to open pages for a given UUID, to publish pages, and to down-vote a page that is currently loaded. The page uses JavaScript to draw requested pages to a section of the browser screen, sending requests to a local Python interface to the Kademlia network using the JSON serialisation format over a web-socket connection.

**Local Node - websocket-node.py**

The Python “local node” provides the web-based front-end an interface to the Kademlia network. The node simply listens on a web-socket connection and then relays any requests to the underlying Kademlia network.

The web-socket Python code was adapted from an example in the source code of the AutobahnPython project [1], which is released under the Apache license.
4.4.3 Demonstration Shell & Python Scripts

In order to demonstrate the running of the program, a shell script was included with the project in the file `src/demo.sh`. The script does the following:

1. Creates an initial node that starts the Kademlia-based network

2. Opens ten instances of the xterm terminal emulator which start other nodes which then join the network. Messages received by these nodes from the network are printed in real-time.

Figure 4.2: A diagram showing the system architecture, including the user-visible front-end.
3. Opens a ‘front-end’ node, which listens on a web-socket connection for the
   front-end. This node also joins the Kademlia-based network as a normal user.

4. Opens the web front-end in the Firefox browser. A user can then store and
   retrieve web-pages in the network, and down-vote them.

This shell script is designed to run on GNU/Linux and has not been tested in
other environments.

As well as this shell script, two files create_network.py and join_network.py
are provided to create a network and join nodes to it.
Chapter 5

Future Work

We will now discuss possible future developments for this project, mentioning the most interesting areas for improvement.

5.1 Censorship

It was determined over the course of the project that there was a trade-off to be made between complete reader anonymity and accurate censorship. Indeed, if all requests made in the network are entirely anonymous, then it becomes difficult to prevent abuse by users that aim to force censorship on certain content. The design of the content-removal algorithm proposed in Chapter 3 of this report aims to mitigate this possibility for abuse by using a logarithmic algorithm for content removal. For future work in this topic, it would be useful to experiment with this algorithm and the values used by it on a large-scale, which was unfortunately not possible in the
short period in which this project was done.

5.2 Cryptography

Further research into the cryptography used by the system could prove useful. Homomorphic encryption, for example, is currently not a feasible approach to this problem due to the complexity involved in using it. As it is developed further and extended to work in real-time, it could be used to transform encrypted data in transit between nodes. This could allow for a more optimal encryption scheme for messages between users on the network.

5.3 Exploits

It would be particularly interesting to investigate possible exploits present in the system. One such exploit is related to the use of a symmetric-key for encryption, decryption and MAC calculations. Since this symmetric key is given to users that read some data on the network, it would be possible for arbitrary users to calculate new MAC values for data that they retrieve. This attack would only be useful if a user knew the UUID of a web-page that they wished to exploit, which means that it could not be performed by a user to exploit the data that they hold, simply because they would need to brute-force reverse a known content location to a UUID, which would require for SHA-1 an average of $2^{160}/2$ SHA-1 calculations. This potential exploit could be remedied by the use of some cryptographic scheme where readers
would decrypt with a public key, and authors would encrypt (and calculate MACs) with a different private key.

5.4 User Interface

For realistic use of the system, a more full-featured user front-end would be required. While the current prototype works well for single pages, a full alternative web-browser or browser plugin would provide a much better user experience and would allow for easy inter-page linking and the embedding of content within other pages (such as external CSS or JavaScript files). The current front-end only provides a prototype, and extra time spent developing a user-facing application would be useful. Still, the prototype front-end developed was not the main focus of the project, and the prototype works well.

5.5 Partial File Replication

More research info the possibility for partial file replication could prove to be very interesting. Some large, distributed data-stores that have been developed in the “Big Data” industry, such as Google’s GFS use a “sharding” technique, where stored data is split into “chunks” of a particular size and these chunks are distributed across storage nodes in the network [7]. These chunks could be partially redundant: one could imagine a system with $N+2$ chunks (for example), where only $N$ chunks would be necessary to re-assemble the original piece of data. This would also save on disk
space, and make exploits on the system more difficult as no single user holds an entire web-page.

5.6 Other Applications Based on this System

It would be interesting to research other possible uses apart from web-serving for the goals achieved in the design of this system. For example, it would be possible to adapt the system to serve as the base for an anonymous micro-blogging platform where user consensus was used to prioritise highly-rated content over poorly-rated content. In such a system, the global default “TTL” could be set to a low value, such as one hour, and thus only a small amount of the most popular content would remain on the network.
Chapter 6

Conclusions

As mentioned in the Introduction to this report, the main goals of this report were to provide a decentralised web-serving network providing the following guarantees:

1. Decentralisation
2. Author Anonymity
3. Reader Privacy
4. Community-Driven Censorship
5. Increased Reliability and Redundancy
6. Secure Data Storage

These goals were met, and a successful and functional prototype of a peer-to-peer network was developed. As well as this, a prototype user-facing front-end which would
allow users to easily interface with the network.

The project was also successful in these other ways:

1. The project implementation was written in a popular, cross-platform programming language, Python. Apart from the Linux-specific shell scripts used exclusively for project demonstrations, the rest of the client implementation should run on any platform which runs Python. This means that the application should run on all popular operating systems, widening the possible scope for further development and usage of the project.

2. The web-socket-based “front-end” node described in the Implementation section of this report allows for user-facing clients to be written in any programming language which supports web-sockets and JSON message serialisation. This could be used, for example, in an environment such as an Android mobile phone. In such a scenario, a local Python application can be run in the background, and a native Android front-end browsing application could communicate with the Python node over web-sockets to access the underlying peer-to-peer network.

3. While this project’s main focus was on web-serving, the general principles behind community-driven censorship could be used as a base for other types of systems that would benefit from community censorship. For example, in our Future Work section, we outline ways that the system could serve as a useful base for other applications, such as an anonymous micro-blogging platform.
Chapter 6: Conclusions

In answer to the questions set out in this paper’s introduction, we conclude that based on the results of this project that:

- Yes, it is possible to build censorship into our system design. Our system implementation has demonstrated that this is possible, and achieves censorship decided by community decision.

- Yes, we can enable censorship by community consensus, but in our system design this resulted in a trade-off between the accuracy of this censorship and the anonymity of the users. In our system design, we chose to compromise the complete accuracy of censorship in order to prioritise complete user anonymity, since user anonymity was a primary goal in the system design. However, the system design does make abuse difficult, so we argue that the system provides tolerably accurate community-driven censorship while still retaining full user anonymity.
Bibliography


Appendix A

Code

All source code for this project has been made publicly available under several open-source licenses. This source code will be made available on the author’s GitHub account, located at https://github.com/scottcunningham/fyp.

The PyDHT fork located in the src/pydht sub-directory of the project has been released under the 2-clause BSD License due to the original project’s licensing.

The source file in src/pydht/key_derivation.py has been released into the public domain since the code upon which it is based was released as such.

All other code with no explicit license mentioned either a LICENSE or LICENSES file in the root of that directory, or note at the top of the file stating otherwise, is released under the Apache license. This code is released under an open-source license so that any future work on the project can use this code as a base implementation.

The code for this project is also included on a CD.