

YÅPS: Yet Another Anonymous Publication System

Christian Boesgaard

pink@diku.dk

Master's Thesis

Department of Computer Science

University of Copenhagen

December 9, 2003

Abstract

Yet another anonymous publication system (YÅPS) is an Internet-based system that makes it possible to publish and retrieve documents anonymously. YÅPS is designed to resist attempts by powerful adversaries to identify users and exercise censorship. The system can be deployed using commodity computer and network resources provided by untrusted individuals and organizations.

In this thesis I shall discuss and present the design of YÅPS. The emphasis on practical issues related to deployment sets the thesis apart from related work. The work on my thesis has resulted in a new rerouting-network design for anonymous communication and a block-based storage system, which can make it impossible for adversaries to link content on storage servers to a specific document. Furthermore, I have implemented a prototype to test parts of the design, which have not been tested by others. The prototype worked successfully. In the thesis I also provide a background on anonymous publication in relation to democracy and free speech.

In the exercise of his rights and freedoms, everyone shall be subject only to such limitations as are determined by law solely for the purpose of securing due recognition and respect for the rights and freedoms of others and of meeting the just requirements of morality, public order and the general welfare in a democratic society.

—Universal Declaration of Human Rights, Article 29 (1948).

Whoever fights monsters should see to it that in the process he does not become a monster. And when you look into the abyss, the abyss also looks into you.

—Friedrich W. Nietzsche, *Beyond Good and Evil*, 146 (1885-86).

Contents

1	Introduction	5
1.1	Project Summary	6
1.2	Thesis Overview	7
1.3	About This Document	7
2	Anonymous Publication	9
2.1	Democracy and Free Speech	9
2.2	Discussion	10
3	Goals	12
3.1	Anonymity and the Threat Model	13
3.2	Environment	14
3.3	Trust	14
3.4	Decentralization	15
3.5	Deniability	16
3.6	Availability	17
3.7	Document Placement	17
3.8	Updateable Documents	18
3.9	Discussion	19
4	Anonymous Communication	21
4.1	Anonymity	21
4.2	Overview of Anonymous Communication Solutions	23
4.3	Rerouting Networks	29
4.4	Mixminion	35
4.5	A Mixminion Based Solution	37
4.6	Summary	40
5	Secret Sharing	41
5.1	Overview	41
5.2	Exclusive-OR Scheme	42
5.3	A $(2,n)$ -Threshold Scheme	43
5.4	Discussion	44

6	Location	46
6.1	Overview	46
6.2	Location Based on <i>Loose Keys</i>	47
6.3	Location Based on <i>Tight Keys</i>	48
6.4	Chord	50
6.5	Discussion	52
7	Miscellaneous Design Issues	53
7.1	Identification	54
7.2	Redundancy	55
7.3	Anonymity	57
7.4	The Index Servers	58
7.5	Storage Servers	59
7.6	Rerouting Servers	62
7.7	Failing Servers	64
7.8	Updateable Document	65
7.9	Recipes	67
7.10	Summary	68
8	Yet Another Anonymous Publication System	69
8.1	Overview	69
8.2	Security	73
8.3	Requirements	75
8.4	Performance	77
8.5	Prototype	79
8.6	Summary	80
9	Conclusions	82
9.1	Future Work	83
A	Reply Block Example	84
A.1	Integrity Protection of Header	86
B	Communication Issues	87
B.1	Communication Layer	87
B.2	Payload Size for Anonymous Communication	87
	List of Internet References	89
	References	92

1 Introduction

To speak his thoughts is every freeman's right,
in peace and war, in council and in fight.
—Homer, The Iliad

Anonymous publication of ideas and critiques is an important part of democracy. The Internet is used to publish all kinds of material. Unfortunately, the Internet is not designed to preserve anonymity, which makes it easy for adversaries to exercise censorship. First, if it is easy to find the location of published material, for example a web server, then the material can be removed or the access to the server restricted. Second, if the identity of people who publish or retrieve material can be found, they can be punished or forced to remove the material. The Internet can on the other hand also be used to build systems that provides anonymity and censorship resistance. In this thesis I shall provide a design for such a system: an *anonymous publication system* (APS).

An APS is an Internet-based system where a person can publish a document by uploading a file to the system. The person can inform others about the published document and they will be able to download a set of files to make a copy of the published document. The APS must protect the privacy of the people using the system by making it difficult to identify them and the servers in the system must also be protected against censorship.

The primary goal of this thesis is to design an APS, which gives users the ability to publish and retrieve documents and protects the users and system against strong adversaries by providing the users with anonymity and the system with censorship resistance. A secondary goal is availability: the system must be able to make documents available even if a subset of the system fails and it must be possible to retrieve documents in a reasonable time.

The goals must be realized in an environment where the resources are provided by small organizations¹ in the form of **servers**: computers providing services. Such an environment is characterized by:

- *Low trust*: Participants cannot find out whether other participants are controlled by adversaries.
- *Unreliable servers*: The servers provided by the participants are not expected to provide reliable services and can be controlled by adversaries.
- *Commodity platforms*: The computing and network resources are expected to be limited to the platforms private people have access to.

Users can publish any kind of file as a document but as the system must provide protection against powerful adversaries it places limits on the effective bandwidth and storage capacity. These limits implicates that the system concentrates on the publication of smaller documents, such as books. Existing systems that provide users with the ability to search for documents, provide limited protection against censorship. Because censorship resistance is a primary goal, I do not expect to design a system that provides searching.

¹ I use organization in a broad sense, to denote an arbitrary people-controlled entity, for example a company, a university, or an individual.

A number of APS designs exist, but none of these meet the stated goals in the described environment. GUNet [4] and Freenet [11] do not support strong protection of users and servers. Eternity [1] and Publius [38] require central servers that are exposed to adversaries. Free Haven [21] does not allow efficient retrieval of documents. Furthermore, I shall provide a stronger legal defense for server operators than existing systems provide: I shall design a system where it is impossible for adversaries to link the content placed at storage servers to any specific documents.

I will base the design on a distributed storage system, which uses the participants' servers to store and serve blocks, which are pieces of a document. Censorship resistance will be based on anonymity of the participants providing the blocks and a means for a legal defense.

The problems I have to address include:

- Providing anonymous communication.
- Finding a way to divide documents into blocks, such that:
 - participants can deny knowledge of the content of the blocks they store.
 - documents can be recreated even if some of the servers fail.
- Making it possible to locate and retrieve the blocks needed to recreate a document within reasonable time.

In this thesis I shall concentrate on a practical design, which works in the envisioned environment. I shall provide solutions to the specified problems, but not present a specification or discuss low-level implementation issues, for example communication protocols or design of client and server software.

1.1 Project Summary

... the ethics most important for the survival of the true, free, human individual would be: ... [to] build improved electronic gadgets in your garage that'll outwit the gadgets used by the authorities.

— Philip K. Dick

Anonymous publication systems are hard to build because usually they are distributed systems, which have to work in low-trust environments with limited resources and must provide resistance against powerful adversaries.

I have combined recent research in the different problems that must be solved in an APS. I provide a rerouting network for anonymous communication based on Mixminion [15] (an overview of the research I did on anonymous communication was presented at a workshop at ECOOP 2003²). Furthermore, I show

² European Conference for Object-Oriented Programming, Workshop on Communication Abstractions for Distributed Systems, <http://www.ecoop.tu-darmstadt.de/workshops/03.phtml> (November 2003).

how *secret sharing* can be used with a block-based storage to make it impossible to link the content placed at storage servers to any specific documents.

I have designed an APS, which is both practical and resistant to attacks by powerful adversaries. The design I present consists of a number of different servers, which:

- store pieces of documents.
- index the pieces.
- provide anonymous communication.

Users of the system publish documents by dividing them into pieces. The pieces are used to create a set of blocks and the blocks are stored in a redundant way on storage servers. The publishing process results in a *recipe*, which can be distributed and used to recreate the document. The system is decentralized and is designed to work in a low-trust environment with a large set of participants whom provide few resources. The design supports schemes for redundancy as part of the publishing process, but does not provide any guarantees on the documents stored. An early version of the proposed design was presented at the conference at *Chaos Communication Camp 2003*.³

I have implemented a prototype to test parts of the design, which have not been tested by others. The prototype worked successfully.

1.2 Thesis Overview

Out of the crooked timber of humanity, no straight thing can ever be made.
—Immanuel Kant

Anonymous publication is closely related to free speech and democracy. In Section 2, I shall place the goals of this thesis into a greater perspective by discussing political and ethical issues related to anonymous publication.

The rest of the thesis is structured as follows: in Section 3, I shall begin with a discussion of the goals of the thesis, as well as a discussion of related work. This is followed by discussions about *anonymous communication* in Section 4, *secret sharing* in Section 5, and *location* in Section 6. Remaining problems and issues of YAPS are discussed in Section 7. The resulting design is described and discussed in Section 8. I conclude and discuss future work in Section 9.

1.3 About This Document

In this document, I have followed the guidelines on writing given by Lyn Dupré in [24]. I emphasize keywords with **boldface** when they are explained.⁴ Citations referring to unpublished work, which does not represent research,

³ Chaos Communication Camp 2003, Conference, <http://www.ccc.de/camp/2003/conference/index.en.html> (November 2003).

⁴ There are some systems I do not explain in one place, the names of these are not emphasized anywhere.

are put in footnotes. For web pages I have noted in parentheses the month I verified the content of web pages.

I use B to denote a byte of 8 bits and prefixes—such as k or M—accordingly to the International System of Units (SI).⁵ This implies that I use kB to denote 10^3B , and MB to denote 10^6B .

I have found inspiration to this thesis in many places, including the works of Karl R. Popper, Roger Dingledine et al.⁶, The Cypherpunks, and Hacktivism.⁷

I would like to thank the people who have helped me during the work on the thesis, especially my adviser Jørgen Sværke Hansen, Allan Beaufour Larsen, and Eric Jul—all from Distlab⁸, as well as my wife Christina Carlsson, Kenneth Andersen, Per Leslie Jensen, and Martin Røpcke.⁹

I made the *Day of the Dead*¹⁰ drawing on the front page based on a figure found on the Hacktivism web page.⁷

⁵ International System of Units (SI) prefixes, <http://physics.nist.gov/cuu/Units/prefixes.html> (November 2003).

⁶ Roger Dingledine is involved in The Free Haven project, Mixminion, and Tor.

⁷ Hacktivism, <http://hacktivism.com> (November 2003).

⁸ Distributed Systems Lab., University of Copenhagen, <http://www.distlab.dk> (November 2003).

⁹ I also wish to thank all the people who have provided me with the software I used in the work with the thesis (for free). The tools I used include: Aspell, Emacs, gcc, gimp, Ghostscript, Common Lisp, L^AT_EX, Linux, Mozilla, Perl, T_EX, Xfig, and miscellaneous GNU tools.

¹⁰ *Day of the Dead* (Día de los Muertos) is a Mexican holiday that honors people's dead ancestors (November 1st and 2nd).

2 Anonymous Publication

Anyone in a free society where the laws are unjust has an obligation to break the law.

—Henry David Thoreau

In the following, I shall present my thoughts on why anonymous publication is important. In Section 2.1, I will demonstrate why democracy is important, show that free speech is a necessary part of democracy, and that anonymous publication is an implication of free speech. This discussion is based on Karl Popper's *Open Society and its Enemies* [45]. I shall end with a discussion of the benefits and problems of anonymous publication in Section 2.2.

2.1 Democracy and Free Speech

Many forms of Government have been tried, and will be tried in this world of sin and woe. No one pretends that democracy is perfect or all-wise. Indeed, it has been said that democracy is the worst form of government except all those other forms that have been tried from time to time.

—Sir Winston Churchill, Hansard, (November 11, 1947)

Democracy is a form of government in which the supreme power is retained and directly exercised by the people. Alternatives to democracy are autocratic government forms such as dictatorship or oligarchy. One argument for democracy is freedom: a government with rules can only coexist with freedom of its people if the rules are made by the people. Another argument is that democracy is a prerequisite for an **open society**, a society where opinions and values can be discussed in the open and used to transform the society. The open society is the opposite of the closed society where decisions are taken by authorities, for example by dictators, priests, or technocrats.

Against democracy, it can be argued that people are not ready to govern themselves, that they lack the necessary knowledge. Surely, some people are marionettes of strong organizations, manipulated, blinded, or controlled in another way. And a benevolent dictatorship might be as good or even a better solution than the democracy—in theory. However, the strength of a democracy is not that its people choose the best government, but that the people are allowed to discuss and evaluate the chosen government, and have the power to remove it. When a dictator fails, he might continue his malpractice until removed by force (probably causing suffering) and it might end with a violent revolution. When a democratic government fails, it can be removed at the next election. Democracy is not an infallible solution, but nothing indicates that such government form exists.

Democracy allows reshaping of a state in small steps based on open discussions and critique. This evolutionary transformation shall be contrasted with the violent revolutions of **utopian engineering**, which tests an ideology using dictatorship. In practice, utopian engineering have resulted in humanitarian

catastrophes, even when the dictatorship was introduced with the best intentions.¹¹

I believe one of the key elements of democracy is the feedback mechanisms of the open society, which provides a way to evaluate the existing government and rules of society. A natural enemy of open society is censorship, a useful means for governments and organizations to silence critics. Free speech can be defended as a human right but it is also a necessity of the open society. Since limits on free speech can be used to silence critics, so it follows that free speech must be provided with few limits to support the mechanisms of democracy.

Censorship does not need laws, force can be used directly against people who do not speak decently or truly in the eyes of a government or organization. In effect, censorship can be exercised if the speakers are known by an organization. This possibility of censorship can be prevented if speakers can be protected with anonymity and allowed to publish their thoughts. Anonymous publication is therefore necessary to protect free speech against censorship.

I conclude that free speech is an essential part of a democracy that seeks to provide the benefits of an open society, and that anonymous publication is a protection of free speech.

2.2 Discussion

They that can give up essential liberty to obtain a little temporary safety
deserve neither liberty nor safety.
—Benjamin Franklin, Historical Review of Pennsylvania, 1759.

There is no conflict between liberty and safety. We will have both or neither.
—Ramsey Clark

Anonymous publication is an essential part of a democracy but the benefits of an open society can also be used to help the ending of dictatorships. The Soros Foundations¹² is an example of an organization that works to advance the democratic development, inspired by the idea of the open society. One of the means of the organization is to provide communication infrastructure in closed societies.

Internet-based anonymous publication can be used to advance a democracy by providing system critics and political parties with a means to work against an oppressive regime (assuming they have access to the Internet). Furthermore, anonymous publication systems are important because the Internet is easily controlled and censored.

¹¹ Karl Marx' socialism is an example. Marx reacted on a horrible and unfair world by designing a theory on society that where used as basis for utopian engineering. The lesson here is that even though Marx worked for a better world, the result was some of the worst societies in modern times. Dictatorship lacks the feedback mechanisms needed to correct inevitable mistakes. Coincidentally, the requirements, Marx envisioned for a better world, were fulfilled, not by any socialistic revolution, but by gradual changes in democracies, for example in Scandinavia and Canada.

¹² The Soros Foundations Network, <http://www.soros.org/> (October 2003). Curiously, the foundation was founded by George Soros who is infamous for his speculation against the British Pound.

It is clear that in most countries over the past two years there has been an acceleration of efforts to either close down or inhibit the Internet. In some countries, for example in China and Burma, the level of control is such that the Internet has relatively little value as a medium for organized free speech, and its use could well create additional dangers at a personal level for activists. [2, p. 7]

The APS Freenet [11] is currently used in China.¹³ The United States *support* Iran by providing the Iranians with anonymous access to information, and circumvention of the Iranian censorship system.¹⁴

Because anonymous publication provides people with the ability to publish material anonymously, such systems can easily be abused. Often discussed examples of abuse include: terrorism, people spreading material showing child abuse, so-called *child porn*,¹⁵ and copyright infringement, so-called *piracy*. Anonymous publication cannot itself be limited to provide protection against such misuse, which implies that the benefits of anonymous publication must be compared with the misuse.

It can be argued that even if anonymous publication might be important to help destabilize dictatorships, it is not an essential part of a democracy, where free speech is a given right, and where a free press exists. I think the current changes in laws in many countries, following the terrorist acts of September 11th 2001, shows that many governments have an interest in increased control of its citizens, for example in the United States where the USA PATRIOT Act¹⁶

broadly expands law enforcement's surveillance and investigative powers and represents one of the most significant threats to civil liberties, privacy and democratic traditions in U.S. history.¹⁷

I believe such encroachment hurts a democracy, and that it is a reason to support anonymous publication as a means to discuss and criticize the actions of a government.

Anonymous publication has drawbacks. This is not different from other tools, which can be misused, for example airplanes or fertilizer,¹⁸ which can be used by terrorists or photographic equipment, which child abusers can use to document their actions as child porn. The question is whether the benefits outweigh the drawbacks—I believe they do.

¹³ See *China News on Freenet* (in Chinese), <http://freenet-china.org/> (November 2003) or GrepLaw: *Ian Clarke on Freenet and his Decision to Leave the USA*, <http://grep.law.harvard.edu/article.pl?sid=03/09/02/0125236> (November 2003).

¹⁴ Kevin Poulsen, *US sponsors Anonymiser – if you live in Iran*, <http://www.theregister.co.uk/content/6/32922.html> (November 2003).

¹⁵ There are indications that the APS Freenet [11] has been used to spread child porn, Slashdot Discussion, <http://yro.slashdot.org/article.pl?sid=03/09/03/226243> (October 2003).

¹⁶ *Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism Act*.

¹⁷ Electronic Frontier Foundation, <http://www.eff.org/issues/usapa/> (October 2003).

¹⁸ Timothy McVeigh used a fertilizer bomb against Alfred P. Murrah Federal Building in April 1995 and killed 149 adults and 19 children, <http://www.infoplease.com/ipa/A0882060.html> (October 2003).

3 Goals

The sea is dangerous and its storms terrible, but these obstacles have never been sufficient reason to remain ashore.
—Ferdinand Magellan

In the following I shall discuss the goals of this thesis and present related work.

An APS is an Internet-based system where a person can publish a document by uploading a file to the system, which consists of a number of servers. The person can then inform other people about the document and they will be able to download a set of files to make a copy of the published document. In practice, the users have a program at their computers, which is used to publish or retrieve a file. In the following I shall use **publisher** to denote users who publish documents.

When users have published a document, they can inform the intended audience about the publication. A publisher must obtain information from the system, which can be used to retrieve the document again and the publisher can distribute this information to the audience with the announcement of the publication. If users could search the system for documents or watch for certain publications, this would be an improvement of the previous scheme.

The research community have proposed a number of APS designs, including: GNUnet¹⁹ [4], Freenet [10, 11], Publius [38], Eternity [1], and Free Haven [21]. In the following, I shall evaluate these systems and other related work.

Anonymity, censorship resistance, and availability, are goals any APS is expected to meet. APS designs differs in how the goals are achieved and in how they handle other issues. I shall divide all the goals and issues into the following categories:

- *Security*: How difficult is it for adversaries to identify users? What threats are the system designed to resist?
- *Functionality*: Does the system give users any guarantees on the lifetime of published documents? Is it possible for users to update documents? Can users search for documents?
- *Efficiency*: How many resources does the system require of servers and users?
- *Deployment*: Under what circumstances can the system be deployed?

My goal is to design an APS, which emphasizes security and ease of deployment. The system must be resistant to attacks by powerful adversaries and the system must be able to run on the computers and network connections available to ordinary people. These goals have higher priority than efficiency and functionality. I believe an APS must be decentralized to provide adequate protection against powerful adversaries.

¹⁹ In [4], the authors describe a way to implement an APS on the top of GNUnet. In this paper I use GNUnet to refer to the design presented in [4].

There are also a few goals related to functionality, which I shall try to integrate into the design. As I mentioned previously, it would make the system easier to use, if users are able to watch for certain publications. I shall provide this ability by using updateable documents. Furthermore, the system should allow documents to be placed on servers chosen by publishers and provide users and servers with the ability to negotiate contracts and place documents based on trust.

Another goal is simplicity: simple systems are easier to analyze, implement, and deploy. I will strive for simple solutions.

In the following I shall discuss the goals and other issues in more detail:

- *Section 3.1*: The threat model and anonymity of users and servers.
- *Section 3.2*: The requirements created by the environment.
- *Section 3.3*: The role of trust.
- *Section 3.4*: Why decentralization is necessary to protect the system.
- *Section 3.5*: Deniability as a defense for storage servers.
- *Section 3.7*: Why arbitrary document placement is important.
- *Section 3.8*: What updateable documents provide and the problems with updateable documents.

I shall finish with a summary in Section 3.9.

3.1 Anonymity and the Threat Model

The primary security goals of an APS are to make it hard for adversaries to reveal the identity of the users and prevent censorship by making it difficult for adversaries to identify servers where a specific document is stored. An adversary, who wishes to exercise censorship must locate the offending material and stop people from sharing it. In oppressive regimes it might be possible for an adversary to turn the problem around and only allow the sharing of approved material on carefully controlled systems. Censorship can be enacted in many ways. First, there are technical means, for example filtering traffic, blocking traffic, or attacking servers with denial-of-service attacks. Second, through legal or repressive means, for example an adversary can make the use of a system illegal or use threats against users and participants (who provide servers).

Users are anonymous in APSs but if the servers that store documents are provided with the same protection, then it will be difficult for an adversary to exercise censorship by attacking these servers. If the servers are anonymous, the information users need to retrieve a document must not contain information, which adversaries can use to identify the servers that store the document. To reach these goals, it is necessary for a system to provide *sender anonymity* as well as *recipient anonymity* (these and other terms related to anonymity are discussed in Section 4.1). Publius [38] and Eternity [1] are examples of APSs that protects the anonymity of users. GNUnet [4] and Freenet [11] protects the

anonymity of both users and servers, but not as strongly as the other systems. Free Haven [21] protects both users and servers.

I have chosen to design a system that provides sender and recipient anonymity and resists attacks on the anonymity from an adversary whom:

- can observe or change the traffic between 50% of the participants
- might control 50% of the servers and users in the system

Such adversaries might be able to successfully expose some endpoints (Section 4.3.1) but the anonymity of users and anonymity-protected servers shall be preserved against other attacks. The location of documents must also remain unknown to such adversaries.

I believe that even if a system can in theory resist adversaries more powerful than the 50%/50% model presented, the system will be vulnerable to attacks by other means, for example attacks using traditional intelligence. To design a system that will work under stronger adversaries, for example under oppressive regimes with strong control of the communication infrastructure, is not a goal.

3.2 Environment

I envision a system that demands few resources from users, who publish and retrieve documents. Low requirements make it possible for more users to use the system, but also make it easier for users in less free areas to cover their activities.

Red Rover [6] is a system, which emphasizes the use of the system under restrictive regimes. Red Rover uses resources in free societies to provide documents to users in more restrictive societies. The system I design must be able to support access like Red Rover does, using HTTP and SMTP to hide activities (I discuss these issues further in Section 8.2.1).

I believe the system must be decentralized (Section 3.4) and I require that the system works on servers provided by many small organizations. This implies that the system must be able to run on and be used on commodity computing platforms. In practice, the resource requirements should be in the same order as existing widespread peer-to-peer systems, such as Gnutella [35] or Kazaa²⁰. However, I expect at least some participants to provide servers with a decent availability—greater than 80%, but the design must not fail if this is not the case (though performance might degrade).

3.3 Trust

Some servers in an APS must be trusted. First, anonymity requires that servers are trusted not to reveal the identity of an anonymous party. Second, users

²⁰ Kazaa Homepage, <http://www.kazaa.com> (November 2003).

must trust the servers in the system to make their published documents available. Third, servers might need to trust users to publish certain content, for example servers donated by organizations or individuals for a certain cause.

Services that provides anonymous communication are difficult for users to verify, as such services are designed to be hard to analyze to prevent adversaries from learning about the communication.²¹ Furthermore, servers that provides anonymity can use covert channels [40] to leak information about their operation.

Users are required to trust the servers, which they use to store a published document on (to keep the document). If users can choose between a set of servers and can obtain information to base their choice on, they can choose the servers they regard as most trusted. Free Haven [20] provides a decentralized infrastructure to keep scores of the servers accountability. I believe, getting this to work in a decentralized system is difficult. Furthermore, I do not believe accountability benefits much from being integrated into the system and it can help adversaries if they can direct attacks against the servers with the best accountability records. I believe accountability should be implemented as a service outside the system, used and updated by a group of users or participants that trusts each other.

Servers must be able to authenticate users to restrict access to users that are trusted. For example, such restrictions can be used to donate storage to certain political parties in oppressive regimes.

As I shall discuss in Section 7.5.2, trust can be based on a *web of trust* [26, Chapter 1] strategy. The requirement to a system to support the use of trust, is to let authentication be integrated into the communication protocols. Furthermore, if the system must provide users with the ability to choose servers or restrict access for users, the system must also provide arbitrary document placement (Section 3.7).

3.4 Decentralization

When I discuss centralization and decentralization in relation to APSs, I shall concentrate on two related but different aspects:

- *Technical*: Is the system working in a centralized or decentralized fashion?
- *Organizational*: Is the system distributed among different organizations?

In a decentralized system, there are no central components, and if some parts fail, some content or functionality might disappear or performance degrade. However, the system must not completely fail. In a centralized system, there are crucial components, which if they fail, make the system unusable. It follows that decentralized systems are potentially more reliable than centralized systems. Decentralized systems require a more complicated infrastructure compared to centralized systems, which is a disadvantage. For example, it is easier

²¹ This situation refers to most rerouting solutions. See Section 4.2.2 and Section 4.3.1.

to keep a centralized index consistent compared to a distributed index. Furthermore, an APS with centralized components might be easier to attack if crucial components are exposed to adversaries.

A system can be controlled by few organizations or spread out among many organizations. Adversaries can attack a system directly by trying to control the resources or indirectly by trying to control the organizations behind the system. Another aspect is whether information about users and participants are available at central points or is distributed. If it is available at central points, these can be attacked. A centralized APS, where a few organizations control the system, is more vulnerable to adversaries than a decentralized system.

An example of the problem of centralization is the cases of the centralized system Napster²² and the decentralized system Gnutella [35]: Napster was successfully closed down by the music industry,²³ Gnutella is more difficult to attack, because there is no central system or organization. The Java Anonymous Proxy (JAP),²⁴ a service for anonymous web browsing, is another example. JAP was forced by the police to provide a bugged client to allow logging of visits from users to certain web-sites. JAP was bugged without informing users, it was discovered accidentally by a user looking at the changes in the new client. The first system providing anonymous email, the Penet Remailer²⁵ is another example of a centralized system that was forced to reveal the identity of a user.

Publius [38] and Eternity [1] are examples of centralized systems, based on few servers. In these systems, the problems discussed in this section are not solved. GNUnet [4], Freenet [11] and Free Haven [21] are decentralized systems.

In order to be resistant to powerful adversaries (Section 3.1), I believe it is necessary for an APS to be distributed over many organizations. As a technical centralized system would still leave few organizations with crucial components, it would be necessary to trust these organizations and protect the components. I do not believe that this is acceptable.

The goal is to provide a decentralized system, which is possible to distribute over many organizations, and where a single organization have limited information about users and participants.

3.5 Deniability

As I mentioned in Section 3.1, to let the servers, which store documents, be anonymous is a goal. The servers are anonymous in Free Haven [21]. Anonymity protects server operators against some attacks, but not against an adversary who search an operator's computer for documents. Even if the mere operation of a storage server is not a crime, a server operator can be punished if illegal content is stored on her server. It follows that it must be hard to show what content exists on the storage server.

²² The design behind Napster is described in [51].

²³ Find Law, Legal News and Commentary, <http://news.findlaw.com/legalnews/lit/napster/> (October 2003).

²⁴ Java Anonymous Proxy: http://anon.inf.tu-dresden.de/index_en.html (October 2003). The case was discussed at The Register Website, <http://theregister.co.uk/content/55/32450.html>, (October 2003).

²⁵ http://www.wikipedia.org/wiki/Penet_remailer (October 2003).

In most existing APSs the operator of a storage server does not know what content she stores. This limited knowledge provides the operator with the ability to **plausible deny** that she knows which documents she stores. If the operator is accused of storing some specific content, she can defend herself by stating that she did not know what she stored. The efficiency of this defense depends on how hard it is to find out what is stored on a server (if it is easy to find out, the operator might have to explain why she did not check what was stored).

Most APSs encrypt the document, or parts of it, and let the encryption key be part of what is needed to retrieve the document. This scheme makes it possible to store an encrypted document. But if the encryption key is known by an adversary and the encrypted document is found on a storage server, then the adversary can show what is stored on the server.

Using a scheme called *secret sharing* (explained in Section 5) it is possible to divide documents into pieces, in such a way that it is impossible to know what document a piece belongs to. Secret sharing makes it impossible for an adversary to search a number of storage servers for a specific document. I call this **strong denial**. I have not found any existing systems that provide this.

To provide secret sharing to protect the server operators is a goal. The use of secret sharing requires that documents must be divided into pieces, which are stored on different servers.

3.6 Availability

There are two aspects of availability: reliability and efficiency. First, the system must be able to make documents available even if a subset of the servers in the system fail. Second, retrieval of documents in reasonable time must be possible.

As I have put few requirements on the availability of the servers in the system, the design is required to provide redundancy. Free Haven [21] uses an information dispersal algorithm to store pieces of a document in a redundant way and Freenet [11] provides caching-based replication.

The goal is to make it possible to retrieve small documents in less than 60 minutes. For huge documents, only the bandwidth in the system should limit on the retrieval time. It is unclear how efficient related APSs are, as few have been evaluated in theory or practice. In Free Haven [21], location of pieces to recreate a document is based on flooding (over a remailer network), this is inefficient. Freenet [11] is based on routing that optimizes the placement of documents, Freenet is expected to be efficient.²⁶

3.7 Document Placement

In some APSs, documents are stored on servers determined by criteria that are beyond the users' control. The placement might instead be determined by the content of the document. Publius [38] provides restricted placement: a document is encrypted and divided into pieces and each piece is stored on a server

²⁶ Practical tests of Freenet indicates some problems, they might be caused by problems incompatible versions.

determined by a secure hash of the content. In Freenet [11] documents are not placed on a fixed server, but cached at different servers on the path to the requesters of the document.²⁷ Freenet shows that schemes, which require control of document placement, can make it possible to move documents around in a system to support faster retrieval or better redundancy.

Another strategy is to allow arbitrary placement and movement of documents or pieces of documents. Free Haven [21] is an example of such an APS. Most file sharing systems also provide free placement, for example Gnutella [35]. The free placement can be used to place documents on trusted servers or to make contracts with entities responsible for some servers. Contracts require a trust scheme where users and servers can remain anonymous. Another benefit of free placement is that it makes it possible to move documents between servers by any scheme, this can for example make it possible to let well-behaved servers transfer their documents to other servers if they leave the system.

Arbitrary placement makes it possible to store and move documents using any strategy, which makes the system more flexible. It is a goal to let the system provide arbitrary document placement.

3.8 Updateable Documents

The main advantage of documents that can be updated is that it can reduce the problem of announcements of publications. If documents are static, a publisher has to use an out-of-band channel to announce new publications. With updateable documents a publisher can announce an overview document, which is kept updated with location information on new publications. The publisher has to make only one announcement with this scheme.

A fundamental problem with the idea of updateable documents is that they can be used to exercise censorship by tracking down the publisher and force the person to delete or modify the document. Another aspect of this problem concerns the safety of the publishers (not the potential censorship option): the possibility of revocation can be an incentive for an adversary to find a publisher and use force to make the publisher revoke the document, for example torture (this problem is discussed in [20]).²⁸

A central goal of an APS is to keep the publishers anonymous, this prevents the mentioned problems. Furthermore, a system can provide both static documents and updateable documents to reduce such problems. Even if a publisher of a static document identified himself in a published document (for example by mistake), an adversary could not force the publisher to revoke it.

Free Haven [21] and Eternity [1] are examples of systems that do not provide updateable documents. Freenet [11] and Publius [38] are examples of APSs that provide updateable documents. In both systems updateable documents

²⁷ However, in Freenet it is also possible for operators of servers to force the servers to store specific documents.

²⁸ The use of terror against users is not a problem only for systems that permits updateable documents. I expect, that an adversary, who uses terror to obtain control with a published document, also would punish or prevent a publisher from further publications.

are implemented by indirection. In Freenet an asymmetric key pair is used to restrict access to update a document. In Publius access to updates is controlled by a shared secret (a password). In Freenet documents are replicated on multiple servers, beyond the publishers control, and after an update there will be different versions of the document available for at a certain time.

The system must provide both static and updateable documents (using asymmetric keys as in Freenet [11]).

3.9 Discussion

In the previous sections I have discussed the goals I shall work toward, as well as presented related work and how it deals with the presented goals. An overview of some of the observations is given in Table 1.

System	Decentralized	Anonymity	Arbitrary Document Placement
GNUnet	✓	(✓)/(✓)	÷
Freenet	✓	(✓)/(✓)	÷
Eternity	÷	✓/÷	✓
Publius	÷	✓/÷	÷
Free Haven	✓	✓/✓	✓

System	Efficient	Strong Denial	Updateable Documents
GNUnet	?	÷	÷
Freenet	✓?	÷	(✓)
Eternity	?	÷	÷
Publius	?	÷	✓
Free Haven	÷	÷	–

Table 1 Overview of anonymous publication systems. The two parts of anonymity refer to *user/server* anonymity.

Freenet [11] does not fulfill the goals I have stated, but is the most used APS. Freenet has a large user base, but it is still a research project under development, and there are too many changes in the network to provide a stable system.²⁹

The Free Haven project [21] is the related work, which most closely resembles my goals. However, there are notable differences: Free Haven does not provide strong deniability or updateable documents. Free Haven also concentrates on integrating accountability into the system, which I consider a wrong decision (Section 3.3). Moreover, there are problems with the current³⁰ Free Haven design. First, Free Haven uses unreliable and high-latency mixmaster remailers for communication. Second, Free Haven does not have an efficient method to locate documents. These two issues result in an unacceptable slow system (Free Haven is not currently used).

²⁹ See *Fracturing P2P Networks* about Freenet, on Slashdot, October 6, 2003. <http://slashdot.org/articles/03/10/06/1058250.shtml> (November 2003).

³⁰ Free Haven is under heavy development, the next generation seems more promising, information can be found on the project homepage <http://freehaven.net/> (November 2003).

I conclude that a new design is required to reach the goals I have stated. I have therefore decided to design *Yet another anonymous publication system* (YÅPS³¹). I will base YÅPS on a distributed block storage system, which will consist of *storage servers*. I will try to make a clean design where the different functionality is separated into different services, I shall treat these services as different servers. A computer provided by a participant can run one or more of such servers.

In the following sections I shall discuss the major issues that needs to be solved in YÅPS:

- *Anonymous communication*: I shall present existing solutions that provides anonymous communication in Section 4 and present a new solution for YÅPS.
- *Secret sharing*: I shall discuss how secret sharing can be used to implement strong denial in Section 5.
- *Location*: I shall discuss how blocks and documents can be located in Section 6. Location is needed to recreate documents.

I shall discuss remaining issues in Section 7, including updateable documents and redundancy issues. In Section 8 I present an overview of the design and discuss security, requirements for users and servers, and performance.

³¹ The letter *å* is the 29th and last letter in the danish alphabet. *Å* was officially introduced in danish writing in 1948 to replace the use of the digraph *aa*. The digraph *aa* has been used in danish to denote the sound of a long *a* since 1534 where it was used in Christiern Pedersen's *Karl Magnus-chronicle*. The symbol *å* was introduced in Sweden in 1526 (source: Dansk Sprognævn, Nyt fra Sprognævnet, <http://www.dsn.dk/nfs/2002-3.htm>). To replace two *a*'s in an acronym with *å* is definitely not good style. I pronounce YÅPS with a so-called short *å* sound to make it rhyme with *Hobbés*—the last name of the philosopher Thomas Hobbes or the name of Calvin's tiger. Note that YÅPS is also an acronym of YÅPS—*an anonymous publication system*.

4 Anonymous Communication

But words are things, and a small drop of ink,
Falling like dew, upon a thought, produces
That which makes thousands, perhaps millions, think.
—Lord G. G. Byron

To provide anonymity for users and storage servers is a goal. YÅPS is a storage system and the primary traffic in the system is publication and retrieval of blocks, this means that anonymous communication is needed for large amounts of data. Because the resources are limited and efficient retrieval is a requirement, the anonymous communication solution must be efficient.

The choice of anonymous communication solution does not have a significant influence on the rest of the design. First, all the solutions allow arbitrary protocols. Second, all the solutions require infrastructure where participants can find information about the anonymous communication system (for example current servers or encryption keys).

In the next sections, I shall discuss anonymous communication. I begin with a definition of anonymity and related terms in Section 4.1, this is followed by an overview of solutions, which provides anonymous communication in Section 4.2. I shall then describe and discuss *rerouting-based* solutions in detail, including a presentation of Mixminion [15] in Section 4.4. Subsequently, I discuss the requirements for YÅPS and a solution based on *Mixminion* in Section 4.3. A summary is given in Section 4.6.³²

I shall use the usual setting where a subject, a **sender**, sends **messages** to another subject, a **recipient**, using a communication network. A message is the payload of one or more **packets**, that in addition contains routing information. The discussion can be generalized to describe other communication scenarios by abstracting away the terms sender, recipient, message, and packet—for example bidirectional real-time connections. I use these terms to follow the custom of most papers on this subject.

4.1 Anonymity

Before I can discuss anonymous communication I have to specify what I mean by *anonymity*. My terms and definitions are inspired by the definitions given in [43].

Anonymous communication is realized when two subjects can exchange messages and their identities are not revealed to the other subject or any third party. In a communication, the two active subjects—the sender and the recipient—are linked to the events of sending and receiving packets to communicate a message. In anonymous communication the subjects cannot be linked to certain events of sending or receiving packets. I shall therefore define anonymity in terms of **unlinkability** between subjects and events:

³² Furthermore, I shall discuss implementation issues related to communication in Appendix B).

Anonymity is a state where a specific subject, within a set of subjects, the **anonymity set**, is unlinkable to an event.

Example: Given an anonymity set of users and the event that one of the users in the set sent a message. The sender is anonymous if it is not known who sent the message.

The anonymity sets for senders and recipients can be disjoint, equal, or have a common subset. The sets can vary over time.

Anonymity can be expressed—or measured—as the probability of a specific subject being linked to an event. Given an anonymity set with the size n , the probability can vary between $1/n$, the maximum possible anonymity, and 1, where there is no anonymity.

The definition of anonymity as unlinkability is symmetric and can be extended to:

- **Sender anonymity:** The property that a particular message is not linkable to any sender *and* that to a particular sender, no message is linkable.
- **Recipient anonymity:** The property that a particular message cannot be linked to any recipient *and* that to a particular recipient, no message is linkable
- **Relationship anonymity:** The property that the sender and recipient in a communication cannot be linked.

Relationship anonymity is a weaker property, than the two others, because it might still hold even if it was possible to link all sent messages and received messages to subjects. Relationship anonymity is the property that is usually expected to hold in systems that provide anonymous communication.

Anonymity deals with the relations between subjects and events. Another related, but stronger property is:

- **Unobservability:** The strong property that it is not known whether an event took place.

In an anonymous communication system that provides unobservability of all events, it is not even possible to find out that subjects are communicating.

Unobservability can also be expressed as a probability, but is customary used to indicate that nothing is known about an event.

Anonymity and unobservability are closely related to information theory, and it is possible to express the probabilities as entropy. Some ideas on the use of information theory to describe anonymity are presented in [18], where a model for measuring anonymity in connection oriented communication is presented. Another model is presented in [17]. The ideas are interesting, but hard to apply to practical systems.³³ There are many missing parameters in the models, and it is hard to model real adversaries.

³³ Usually, theoretical models for security are difficult to apply to practical systems.

I require sender and recipient anonymity in YÅPS (Section 3.1). Unobservability can provide better protection in theory, but it is hard to support in practical systems and the benefit is small. As long as the users and servers are anonymous, I do not believe it changes security in a significant way if an adversary can learn that the system is used or how much it is used. To limit exposure of users or make the system resistant to attacks against the anonymity is of greater importance.

4.2 Overview of Anonymous Communication Solutions

In the following I shall shortly mention attacks against anonymous communication, and then give an overview of solutions for anonymous communication.

That an adversary cannot get information by analyzing the content of the messages exchanged (the payload of the packets) is an assumption. Messages can be protected by encryption to prevent adversaries to learn from the content as the messages is communicated.

Adversaries can direct attacks against the traffic between the set of users and servers, such attacks are directed against communication links, and are usually denoted as **link-based** attacks. I shall use the term **server-based** to denote attacks that concentrate on servers or requires control of servers. I use the term **passive** to describe attacks where traffic or environment is not altered, for example eavesdropping. **Active** attacks are attacks where changes are made to packets or where packets are blocked or inserted. Active attacks are often considered harder than passive attacks, but I believe active server-based attacks are easier to mount compared to attacks on communication links, because today's operating systems are more vulnerable than most network connections. Attacks against a small subset of servers or links are described as **local** attacks in contrast to **global** attacks, where an adversary controls most servers or communication links.

Any anonymous communication solution requires that the computers of anonymous senders and recipients are uncompromised, otherwise the attacker can eavesdrop at these endpoints.

Anonymous communication solutions falls into one of two groups **broadcast based** solutions and **rerouting based** solutions. David Chaum was the first to come up with an scheme based on broadcast: DC networks³⁴ [8]. DC networks are based on a simple protocol where all the users are broadcasting bits in steps. a user can send one bit anonymously in each step. Before Chaum came up with the idea of DC networks, he proposed a solution [7], which provides anonymity by the use of a set of servers, which **reroutes** messages to make it hard to correlate sender and recipient. Such a server is called a **mix** or a **rerouter**.

³⁴ DC is both a contraction of *David Chaum* and *Dining Cryptographers*.

4.2.1 DC networks and Other Broadcast Solutions

DC networks were introduced as an unconditional secure solution to provide anonymous communication. The scheme was introduced to solve the **dining cryptographers problem**:

Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. [8].

The problem in this scenario is that one cryptographer needs to be able to anonymously communicate exactly 1 bit to the others if she paid for the dinner. If all the cryptographers could commit 1 bit anonymously, and a paying cryptographer committed a 1, the problem was solved. David Chaum proposed in [8] that each cryptographer flips a coin (with a 0 and a 1) in secrecy with the cryptographer sitting to the right side (imagine a round table). Each cryptographer can then tell if the two coins she sees are equal. If a cryptographer paid the dinner, she should lie.

The following interpretation of the protocol based on exclusive OR is my own (Chaum does not use exclusive OR in the explanation). I shall use \oplus to denote the exclusive-OR operator. Assume each cryptographer says 1 (for true) if the two coins observed were equal and 0 (for false) otherwise. The result of an exclusive OR of all the statements by the cryptographers, would result in 0 if no cryptographer paid the dinner, and a 1 if one of them paid. The result would be 1 because for n cryptographers there would be n sub results from coin flips and each sub result would be \oplus 'ed twice times to the final result and as $P \oplus P \oplus Q = Q$, the sub results would cancel themselves out from the final result. A liar however, would negate a sub result, and this would result in a negation of the final result. Chaum's solution can be generalized to any number of participants. The requirements for the solution is that all the subjects have a shared communication channel, and that each pair of participants have a communication channel (to use for coin flipping).

An adversary who observes all the statements of the cryptographers, can only tell that someone submitted a bit—not who. This results in sender anonymity and because the information is broadcasted, the recipient can also be anonymous. The anonymity is unconditional against global passive attacks on the traffic. No information sent as part of the scheme can help the adversary to identify the sender or receiver. An active attacker can reduce the anonymity set only by controlling some of the participants. If the attacker controls the two neighbors to a participant s , the attacker would be able to see the two coin flips, and be able to tell whether s committed a bit. Chaum's scheme is strong and the security is better than the requirements for YÅPS (Section 3.1): an attacker who controls 50% of the servers will be able to identify only the participants of whom both neighbors are controlled.

The disadvantage of DC networks is that only one bit can be committed at a time on the shared communication channel, which makes the system inefficient. Also a protocol is needed to let a recipient know when a message is sent to her. Few practical systems have been based on DC networks. However, *Herbivore* [31] and \mathcal{P}^5 [56] are two examples. I shall discuss those in the following. \mathcal{P}^5 provides a system based on the broadcast idea in DC networks. The system is an improvement of Xor-Trees [23]. In \mathcal{P}^5 subjects are ordered in a binary tree structure defining different broadcast channels, with a trade off of anonymity and efficiency. The system has been simulated: the analysis shows that in a system with 8192 subjects and anonymity sets in the order of 100 subjects, each participant must constantly handle the decryption of about 200kB of asymmetric encrypted data per second, and use 2 Mbps of bandwidth. In this setting, a subject will be able to send about 200 kbps of anonymous communication with a packet loss of around 40%³⁵. While a sender is sending a message, she can use 6% of the available bandwidth—the traffic needed to send a messages is increased with a factor of 17. If a sender is not using the ability to send messages at all times, then the overhead increases accordingly (as the same computations and traffic must be handled). For larger anonymity sets much less than 1% of the bandwidth needed can be used to transfer content (I speculate that this is a high estimate as the overhead grows fast ($O(n)$) with the size of the anonymity set).

Herbivore extends DC networks by providing protocols for dividing all participants into smaller subset and protocols for bandwidth reservation. The overhead, *excluding* the traffic needed for reservation *and* the traffic wasted on collisions, is at least $2(n - 2)$ for each bit sent, where n is the number of subjects in the anonymity set. I believe that the anonymity sets in YÅPS will be larger than 1000. An anonymity set of 1000 would amount to an overhead of at least 99.94% of the bandwidth (or increase the traffic with a factor greater than 1996). *Herbivore* has been implemented and tested with small anonymity sets (10–40 subjects). For large anonymity sets much less than 0.01% of the bandwidth needed can be used to transfer content.

As the users in YÅPS are expected to provide a limited amount of resources, including bandwidth and computation, the solutions presented in this section are not feasible.

4.2.2 Mix Networks and Other Rerouting Solutions

David Chaum’s first proposal for anonymous communication was based on the concept of rerouting servers [7]. A rerouter, which Chaum calls a mix, is inserted between the sender and recipient, and its purpose is to make it hard to correlate the communicating subjects.

In the original design, a mix had a public and private encryption key. The sender of a message uses the public key to create a packet by encrypting a message with information about the recipient and send it to the mix. The mix stores a number of such packets in a pool, reorders them and decrypt and send

³⁵ These numbers have been taken from the paper: it is not clear from the paper what the prefixes exactly denote.

them to the recipients. The effect of a mix is shown in Figure 1. The reordering done by a mix prevents a passive attacker from correlating the in- and outgoing packets to correlate senders and recipients. The result is relationship anonymity. A side effect of the reordering is that packets passing a mix will be delayed, which introduces a latency.

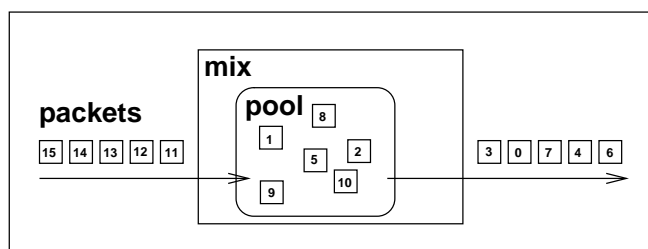


Figure 1 A mix randomizes the order of incoming packets using a pool. An adversary who observes the communication links cannot link senders and recipients because the incoming packets cannot be correlated with the outgoing packets.

In the following I will collectively refer to senders and recipients as **endpoints**.

A set of mixes are used instead of just one mix because a single mix cannot provide adequate security. In a system with a single mix an adversary can identify all endpoints, by eavesdropping on the communication links to a single server. And just one server must be compromised to break the system completely. Instead packets are sent through a subset of mixes from the total set of mixes, following a path. A user chooses a set of mixes to use as a path for a message. Packets are created by encrypting the content with the public keys of the mixes, starting with the last mix on the path, then the public key of the mix before the last, and so on. The use of layered encryption is depicted in Figure 2. Each mix on the path knows about only the next and the preceding mix (or an endpoint). This encryption scheme guarantees that even if some mixes on the path are compromised by an adversary, anonymity can still be preserved. Also an adversary must monitor more communication links to identify endpoints. The use of a set of mixes is shown in Figure 3.

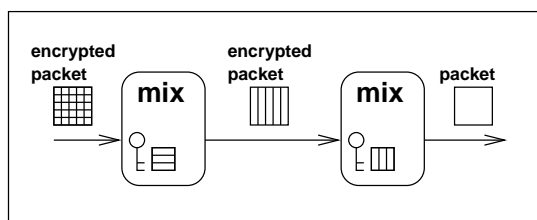


Figure 2 Multiple layers of encryption is used to prevent mixes from learning about the path of a packet. This figure shows how a packet encrypted with the keys of two mixes are decrypted, layer by layer, as it travels along the path.

Mixes are built to provide sender anonymity, but it is also possible to support recipient anonymity using **reply blocks**. A recipient can create a reply block,

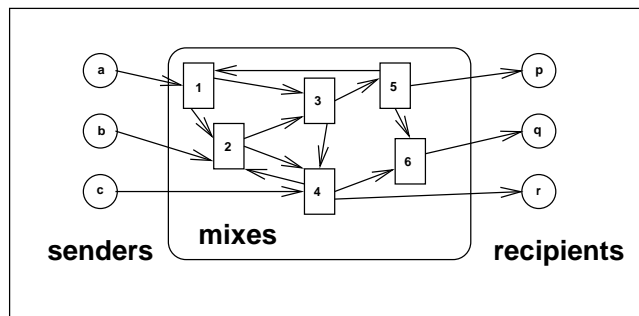


Figure 3 A group of mixes require an adversary to observe a set of mixes. In this example an adversary must observe the mixes 1, 2, 4, 5, and 6 to be able to identify the endpoints of the communicating parties. Furthermore, if an adversary wants to know exactly who a sender is communicating with, more than one mix must be attacked, as a chain of mixes is used by all the senders.

by encrypting a header as if it were a packet, but with herself as the recipient. The reply block is distributed to senders. Senders can attach the reply block as a header to a message and send it through mixes to the recipient. A reply block makes use of both the public keys of the mixes on the path and a number of symmetric session keys generated by the recipient. Usually, the public keys are used to encrypt the header and each mix gets access to a session key when they decrypt the header. The session key is used to encrypt the payload of the packet before it is resent. The use of a reply block is shown on Figure 4 and the creation and use is described in detail in Appendix A. As the message is unknown at the creation of the reply block, the message cannot be integrity protected by the reply block. Reply blocks are also prone to attacks because messages sent using a reply block always use the same path.

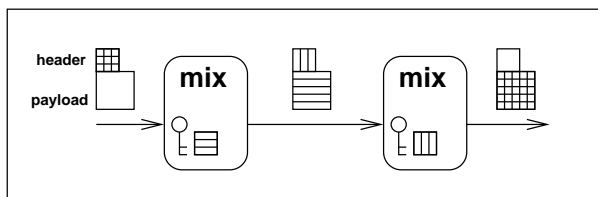


Figure 4 A reply block is used as a header in a packet to send a message as payload to an anonymous recipient. As the layers of encryption are removed from the header, other layers are added to the payload. The recipient knows all the keys used to encrypt the payload and can remove them. Compare to Figure 2.

Mixes have been used for anonymous email since the original Cypherpunk

remailer³⁶ was deployed in the early nineties by the Cypherpunks.³⁷

The high-latency mix networks cannot be used to provide anonymous real-time connections, such as TCP connections. Other rerouting solutions, have been proposed to provide low-latency connections, for example Onion Routing [32], Tor [22], and Tarzan [28]. These systems work by setting up a circuit using a path like the original mix design. The path is used for all the communication in a session. As in mix networks only the two neighbor rerouters are known by any rerouter on a path. Packets are typically not rearranged at the rerouters as they are in a mix, but instead resent immediately. The missing reordering reduce security. The problem is that the in- and outgoing packets of a rerouter can be correlated by timing and all the packets in a communication are traveling along the same path. The system does still provide anonymity against local attacks because it is still difficult to correlate a packet observed somewhere on the path with a sender or recipient. I shall use the term **OR networks** to denote low-latency rerouting solutions, such as Onion Routing and Tarzan.

Crowds [49] is a variant of rerouting, which only provides sender anonymity. In Crowds all participants are rerouters. Rerouting is done by creating a circuit based on a simulation of the flipping of a biased coin at each rerouter that receives a packet. The rerouters either forward the packet to the destination or to another rerouter, based on the coin flip. This scheme provides a circuit with variable length path. The problem with this strategy is that it requires a packet to contain information of the destination unencrypted (because all rerouters on the path must be able to forward the packet to the destination), and the packet is exposed to all rerouters on the path. This exposure can be used to correlate sender and recipient, as in- and outgoing packets can be correlated at each server (based on the destination information).

GNUnet [3] provides another variant of rerouting: queries are routed using a flooding strategy. If a server stores something that can answer the query, a response is sent back to the sender. If a server cannot answer, it queues the query, and resend it to its neighbors, but it might change the sender address to its own address. A similar strategy is used in Freenet [11]. This strategy does not provide strong anonymity because packets can be traced (an attack on GNUnet is described in [36]).

Rerouting networks have been used in practical settings and they support a large number of users. Usually, the number of users scales well with the number of rerouters. In many systems there is a small network-traffic overhead for the users, to allow space for routing information. Typically the overhead is between a factor of 1.1 and 2 (1.14 in Mixminion [15]). The rerouters have to reroute the messages, and assuming paths of 5 to 10 rerouters, the total amount of overhead will be a factor of 5.5 to 20, corresponding to 80%–95% of the total network traffic required to send messages. But this overhead does not grow with the number of participants, it is only dependent on the amount of payload sent and affects the rerouters—not the users.

³⁶ There are no published material written by the creators. A collection of emails on the Cypherpunk mailing list describes the development of the Cypherpunk remailer: <http://cryptome.org/zks-v-tcm.htm> (November 2003).

³⁷ *The Cypherpunks Home Page*, <http://www.csua.berkeley.edu/cypherpunks/Home.html> (November 2003).

A weakness of most rerouting solutions is the use of a path created before a message is sent. This scheme requires all the rerouters on the path to function properly, otherwise the message cannot be delivered.

4.2.3 Discussion

DC networks and related systems provide unconditional security against passive attackers, even global passive attackers. The problem is that existing systems require that all subjects provide resources, even when they do not communicate. This requirement combined with a huge traffic overhead makes it infeasible to use these solutions to provide anonymous communication in YÅPS.

Mix networks and related systems can provide relationship anonymity and sender and recipient anonymity against local attackers. These solutions are also inefficient, but not as inefficient as DC networks and rerouting solutions scales much better with the number of endpoints in anonymity sets.

I shall base the anonymous communication in YÅPS on rerouting.

4.3 Rerouting Networks

In a rerouting network, anonymity is provided by using a network of rerouters to **reroute** the communication. An anonymous sender chooses a path of rerouters to send a message through, which requires the sender to know a set of available rerouters. Recipient anonymity requires the recipient to provide senders with a reply block.

I shall now discuss rerouting networks in more detail. I start by presenting attacks against rerouting networks in Section 4.3.1 and path selection in Section 4.3.2. I shall then discuss different rerouting solutions in Section 4.3.3.

4.3.1 Attacks Against Rerouting Networks

The function of a rerouting network is to make it hard for an adversary to correlate the incoming and outgoing packets to and from the rerouting network. An adversary can break the anonymity that the system provides, if she can correlate the incoming and outgoing packets to and from the rerouting network. A problem with the functionality of a rerouting network is that it is hard to verify that the network and the rerouters behave correct. This problem is difficult to solve, because it is the purpose of the rerouter to obfuscate the rerouting process. The problem makes it impossible for users of the rerouting network to identify compromised rerouters and packets manipulated by an adversary.

In the following I shall describe different kinds of attacks against rerouting networks and discuss countermeasures. Generally, an active attacker can flood a system with packets, or try to bring down or make some of the rerouters unstable. Both attacks can be used to make the system unusable (a denial-of-service attack) or to force users to use a specific set of rerouters—in effect making other attacks easier. I speculate that against the most powerful adversaries—for example a nation like the United States—attacks are impossible to prevent but

large rerouting networks makes the attacks hard. Like for other security systems a practical goal is to make it more expensive to break the system, than the value of revealing the identity of the communicating parties for the attacker.

Users of a rerouting network must place complete trust in at least one of the routers on the path. There are two aspects of this trust. First, users must trust the operator of a router not to reveal information about the communication. A router operator can be attacked with court orders³⁸, threats, or bribery. Second, users must trust an operators ability to defend her router against attacks to compromise the server (for example, an adversary can use Trojan horses or vulnerabilities in the operating system of the rerouting server to mount server-based attacks).

Another concern is the users themselves, as the users can use the rerouting network in a wrong way. Case studies of this in real systems are provided in [12]. I shall try to avoid mistakes of this kind in YÅPS by clearly describe how the system is expected to be used to provide anonymity.

Server Based Attacks

If a server in a rerouting network is compromised by an adversary, the effect of that server as a router is eliminated. All communication using a path including the compromised server is protected as if the path did not include that server (against the adversary who controls it). If all the servers on a path are controlled by an adversary, the rerouting does not provide any anonymity. The effects of attacks directed against the servers have varying effects dependent on the number of servers compromised and the path lengths. Such attacks must be taken into account by a path selection scheme and servers and operators must be protected against such attacks. A rerouting design cannot by itself protect against such attacks.

Replay Attacks

A replay attack works by observing the in- and outgoing packets to a router. If a packet is resent to the router, this results in the sending of a packet by the router, which has already been sent, and an earlier incoming and outgoing packets can then be correlated.

Replay attacks can be countered by caching packet identifications or hashes of packets and reject packets that are duplicates of packets already seen. If it is a problem to store enough entries in the cache, timestamps can be introduced, and packets older than the oldest in the cache rejected. However, this solution allows attackers to use the information in the timestamps. Another solution is key rotation as introduced by Lance Cottrell.³⁹ Using this strategy, a router

³⁸ Examples of court order attacks include: *United States District Court for the District of Columbia, Civil Action No. 98-116*, http://www.loundy.com/CASES/McVeigh_v_Cohen.html (November 2003) and *American Health Scan v. Technical Chem. and Prods., Inc.*, http://www.cybersecuritieslaw.com/lawsuits/cases_corporate_cybersmears.htm, (November 2003).

³⁹ Lance Cottrell, *Mixmaster & Remailer Attacks*, <http://www.obscura.com/~loki/remailer/remailer-essay.html> (November 2003).

only uses its public key in a period of time corresponding to the size of the cache. When the cache is full, a new key is made, and replays using the old key is be rejected (the packets cannot be decrypted correctly). A disadvantage of key rotation is that it create a key management problem, as users and servers need access to a new public key when a key is rotated.

Another possible solution is to allow packets to be replayed, if they are not in the cache of packets already seen, but make it difficult for an attacker to determine if an outgoing packet is a replayed packet. An encryption scheme, which do not encrypt the same packet to the same ciphertext twice,⁴⁰ can make it difficult to combine link attacks with replay attacks: an attacker who replays packets and observes traffic is not able to make a direct replay attack by observing duplicated packets (but it might still be possible to learn from the duplicated path the replayed packet travels). However, adversaries who controls a number of rerouters, can still identify replayed blocks that passes one of the controlled rerouters, because they can observe the decrypted content (that remain unchanged).

Blending Attacks

An active attacker can block or delay packets and insert bogus packets to manipulate rerouters to correlate packets. This attack is called a blending attack because the adversary blends his own fake packets with the real packets in a rerouter. If messages are blocked, the set of incoming packets would be reduced, which would make it easier to correlate incoming packets with outgoing packets. A countermeasure is to require rerouters to stop resending if they only receive few incoming packets. However, this countermeasure does not work because the attacker can just insert bogus packets to keep the number of ingoing packets high. The adversary can remove the bogus packets again from the set of outgoing packets (for example, the adversary could make sure the bogus packets where resent to rerouters she controlled).

To make countermeasures against blending attacks, is hard, specially against global attackers. A detailed analysis of the effect of blending attacks against different types of mixes are given in [54]. One countermeasure is to use random cover traffic. This cannot prevent a blending attack, but make it impractical because an attacker would have to find out which packets are real packets and which are just cover traffic. Another strategy is to use a verification scheme to authenticate servers or time stamp messages. This might be able to prevent blending attacks, but is impractical, because it introduces complicated and resource demanding protocols.

Edge Attacks

In a rerouting network where the rerouters can be distinguished from the endpoints, it is possible for a passive attacker to identify senders or recipients if the

⁴⁰ One solution is to use a block cipher in CBC-mode with a random IV and send the IV with the packet [39, Section 7.2.2].

communication between the endpoints and the closest rerouter on the communication path is observed. Such an attack is called an **edge attack**. If endpoints cannot be distinguished from rerouters, this attack is impossible. Tarzan [28] is an example of a system where endpoints cannot be distinguished from rerouters, because all participants operate rerouters. A problem with such systems is that all participants must provide resources to work as a rerouter. Another problem is that it can make it easier for an adversary to find users if they all operate rerouters (specially if a list of rerouters is published).

If edges are open to attacks, for example if the network connections of users are observed, the only solution is to try to hide the traffic to the system (I shall discuss this in more detail in Section 8.2.1).

Attacks Based on Reply Blocks

A reply block is used to create a packet that can be sent to an anonymous recipient. If a mix network allows reply blocks to be reused, then an attacker can use a reply block to send a number of messages to a recipient and they will all travel along the same path. For example, an attacker can send a message to the recipient, and the attacker will know that the next mix on the path will be from the set of mixes that the mix forward packets to. This can be repeated and the attacker can reduce the set of possible mixes, because she knows the next mix on the path is in the intersection of all the sets of mixes she have learned about. The attack is hard to apply because the mixes can potentially store messages for a long time unknown to the attacker. The attack can however be combined with other attacks.

This attack, which uses reply blocks, can be prevented if reply blocks are only allowed to be used once, this is the solution provided in Mixminion [15]. The disadvantage is that recipient must create and distribute at least one reply block for each message they receive.

Miscellaneous Correlation Attacks

If the packets have characteristics that can be correlated, a local passive attacker can break the security provided by a rerouter by correlating the packets coming to and leaving from the rerouter. For example, a correlation attack can be based on the size or content in a packet. The Cypherpunk Remailers did not pad packets to a fixed sizes, which made it possible to correlate the packets. If part of the content of a packet is unchanged at the rerouter, this can be used to correlate packets. If packets are re-encrypted, passive attackers cannot use the content to correlate packets.

An active attacker can try to alter parts of incoming packets and if the alteration can be correlated to changes in outgoing packets, the packets can be correlated. Such a **tagging attack** is possible only if the attacker can identify the tagged packet at later on the path of the packet on either a communication link or a controlled rerouter.

A tagging attack can be countered if the tagging can be revealed at the first rerouter the tagged packet arrives at. Integrity protection will reveal such an

attack. Integrity protection can be implemented using a MAC [39, Section 9.5]. In Mixminion [15], the integrity protection is provided indirectly by encrypting part of the header with a key based on the payload and some of the information in the header. If a single bit is changed in the content the key is based on, the result is a random-looking header after decryption. A random packet can be discarded by a mix (it would be impossible to forward the packet anyway, because the routing information would be unreadable).

Another way to correlate packets are timing. Correlation by timing is usually not a problem in modern mixes, but OR networks, which does not rearrange packets before resending are prone to such attacks. Such OR networks allows adversaries to correlate the sender and recipient of a communication, if both are observed during the communication. In Tarzan [28], the communication is based on pairs of rerouters. Each pair exchanges a constant amount of data, either anonymous communication or random data. This makes it impossible for an adversary to learn anything just from watching traffic.

Other Attacks

The attacks presented previously are used to directly identify users, but other attacks are also possible. If the objective for an adversary is to prevent anonymous communication, this can be done by blocking access to rerouters or by denial-of-service attacks against the rerouters. Such attacks can also be combined with other attacks to reveal the identity of users, as users can be forced to use a subset of the rerouters.

4.3.2 Path Selection

In this section I shall concentrate on path-selection issues related to security.

The path selection is an important aspect of rerouting networks. If all the rerouters on a path are controlled by an adversary, the endpoints are not anonymous against this adversary. The path is usually chosen at the endpoint, or at the first rerouter. In some rerouting networks the path is freely chosen as a subset of the trusted and available rerouters, this is called free-route rerouting networks. Another possibility is to have a fixed path for all communication or let endpoints choose between a number of fixed paths constructed from the set of rerouters. These paths are called **cascades**.

In [5] it is argued that against a global attacker cascades are the strategy, which provides the highest amount of anonymity, because fewer attacks are possible. The general idea is that in a cascade with one fixed path the anonymity set consists of all the participants (for example all senders), and this does not change even if all but one rerouter is controlled by an adversary (ignoring endpoint attacks and assuming the mixes forward the packets correctly). In a free-route rerouting network, the anonymity set for ingoing packets to a rerouter is a union of the anonymity sets of the rerouters that sends packets to a rerouter, this anonymity set will usually be a small subset of all the senders, because the senders might use different paths and only a subset of all rerouters.

Cascades introduce other problems. First, to make edge attacks against cascades is easier, because all the rerouters placed first or last in the cascades are

known, and this set is smaller than the total set of rerouters. Second, blending attacks (Section 4.3.1) and other attacks based on the sets of rerouters a rerouter communicated with are easier against cascades, as the paths are fixed. Third, cascades makes a system more vulnerable to denial-of-service attacks or blocking, because they create crucial components. Fourth, cascades prevents untrusted or malfunctioning rerouters from being excluded by a user from a path.

In practice, the disadvantages of cascades outweighs the added security against a global attacker. Furthermore, I speculate that the power a global attacker represents, can be used to break any rerouting network using other means. I shall use a free-route path selection for the anonymous communication in YÅPS.

The path length is essential for the security. The optimal path length is studied in [33]. In that paper, formulas are provided that can be used to calculate optimal path lengths or length distributions, given a number of rerouters and the expected number of rerouters controlled by an adversary. In the paper it is shown that variable path lengths are most secure and that security does not monotonically increase with the path length. There is a maximum length, and the use of paths longer than this length decrease anonymity. Longer paths reduce security because they can include a larger number of rerouters controlled by adversaries. The decrease in anonymity is of theoretical nature and I speculate that other attacks will be more of a concern in scenarios where long paths reduce anonymity.

The path length must be determined using the number of rerouters and endpoints, and the number of rerouters an adversary is expected to be able to control, as well as the anonymity expected of the system. As an example I would—based on [33]—propose variable path lengths between 5 and 10 rerouters for systems with 50 to 100 rerouters. These are numbers that I would expect of the anonymous communication in a practical deployment of YÅPS.

4.3.3 Discussion

I have introduced the idea of mixes and rerouting networks, presented attacks against such systems, and discussed path selection. I shall now discuss a possible solution to provide anonymous communication in YÅPS. First, I have to identify my requirements. I need a solution that provides both sender and recipient anonymity and there must be a limit on latency to support document retrieval in reasonable time. Another consideration is reliability, YÅPS must be able to work even if a subset of the rerouters fail, I shall return to this in Section 7.6.1. I also have to consider whether rerouters in YÅPS must be indistinguishable from endpoints.

Systems like Crowds [49], do not provide recipient anonymity, so they fail to meet the requirements. Solutions like those in GNUnet [3] and Freenet [11], provide only limited anonymity, GNUnet is analyzed in [36]. Furthermore, these systems provide a limited variant of recipient anonymity, because they are based on queries, which are spread in the system.

Real-time connections are not required, so mixes are a possibility as long as the latency can be limited. Mix networks are harder to attack than OR networks (Section 4.3.1), I shall therefore base YÅPS on a mix-based solution.

If endpoints cannot be distinguished from rerouters, edge attacks are prevented, but the solution introduces problems: all users must operate rerouters, which requires resources and it makes users more exposed. I shall not require that users are participating as rerouters (but it should be possible for users to participate with a rerouter, to provide security against edge attacks).

Different strategies exist on how a mix collects and resends packets [16, 54]. These strategies are based on the number of packets the mix currently holds or timing. In systems where low latency is a requirement, timing is used to guarantee low latency, an example is the simple **timed mix**, where all packets are sent at each timeout. Another simple strategy is the **threshold mix**, where all packets are sent when a certain number of packets are reached. The more secure strategies hold a large pool of packets, resulting in high and unpredictable latency. An example is the **Cottrell mix** [13], where randomly chosen packets are sent at each timeout, but only if the number of packets in the pool is higher than a fixed threshold. The Cottrell mix was introduced with the Mixmaster remailer [13], and the reordering in Mixminion [15] is also based on it.

In Mixminion [15], the rerouting is made more resistant against blending attacks by adding dummy traffic between the mixes.

Some mixes are based on theoretical work that can provide strong guarantees, including zero-knowledge proofs [39, Section 10.4]. These include flash mixes [34], provable shuffles [29], and hybrid mixes [41]. The advantage of these approaches are the guaranteed properties, but the problem is that they are impractical because they require synchronization and are ineffective. These solutions are impractical for YÅPS.

Most work on mixes has ignored the practical problems that must be solved to deploy the system—the general exceptions being remailers. Practical remailers for anonymous email have been tested and proved to work in practice. The Mixminion remailer [15] is the latest remailer design, and I have chosen to study this as a basis for a mix-based anonymous communication solution for YÅPS. I shall present Mixminion in the next section and in Section 4.5 a solution based on Mixminion.

4.4 Mixminion

Mixminion [15] is an attempt to design a modern mix-based rerouting network for email communication. Mixminion is based on the mixmaster remailers as well as on current research on anonymous communication.

I shall use the following terminology to describe the different types of packets in Mixminion⁴¹:

- *F-packet*: Forward packet, used to provide sender anonymity.

⁴¹ The terminology I use differs slightly from the terms used in the Mixminion paper [15].

- *R-packet*: Reply packet, used to provide recipient anonymity. The sender uses a reply block provided by the recipient to create the packet.
- *FR-packet*: Forward-reply packet, used to provide sender and recipient anonymity. The sender uses a reply block provided by the recipient to create the packet.

Mixminion is the first remailer to drop SMTP for packet transport, this is replaced by TLS (Transport Layer Security) [19] over TCP. TLS is used to authenticate the next rerouter on a path, and transferred packets are encrypted with a session key to make passive or active attacks on the link impossible.

As I have mentioned earlier (Section 4.3.1) replay prevention is done with a combination of a cache of already seen packets and key rotation. In Mixminion, each mix must announce its keys and key rotation scheme to a directory service, this service is used by the senders to create packets. Mixminion is a free-route mix: senders create paths randomly from the set of current mixes.

To provide recipient anonymity it is necessary to use reply blocks. In Mixminion reply blocks are not allowed to be reused. Single-use reply blocks require recipients to produce and distribute a number of reply blocks corresponding to the messages that they expect to receive. Reuse of reply blocks can be prevented the same way replay of packets can be prevented. In Mixminion, single-use reply blocks are denoted SURBs, I will also use that acronym.

To provide as large anonymity sets as possible, Mixminion [15] makes it impossible to distinguish between F-, R-, and FR-packets. The use of a dual header scheme makes it impossible to distinguish between the different packets. There are two headers in a packet, which are built depending on the packet type (*hops* refers to the number of mixes on a path):

- *F-packet*: (1) A sender generated mix-route (4 hops) (2) a sender generated mix-route ending with the final recipient (2 hops).
- *R-packet*: (1) SURB (recipient generated mix route, 4 hops) (2) empty.
- *FR-packet*: (1) A sender generated mix-route (4 hops) (2) SURB (recipient generated mix route, 4 hops).

The packets are routed along a path and at a crossover-point (the last mix on the path in the first header), either (1) the packet is forwarded to the final recipient, or (2) the headers are swapped, and the rest of the path defined in the new header. Because the headers are encrypted, they cannot be distinguished.

The use of reply blocks in mix networks usually allows tagging attacks, because the message cannot be integrity protected (the reply block is created independent of the message). Because packets must be indistinguishable in Mixminion, it is not possible to provide integrity protection directly on any of the packets. Instead Mixminion provides indirect integrity protection using a block cipher with a large block size (see Section 4.3.1), and by the use of a cross-over mix, where the headers are swapped. The protection is implemented by using the first header and the payload to encrypt the second header at the cross-over mix, this results in an unreadable second header if there have been made changes in

the second header or the payload. If a packet is tagged, the packet is destroyed at the crossover-point (by the decryption), and will never reach the recipient (thus preventing an adversary from identifying the recipient). R-packets do not pass a crossover-point, but they are not open for tagging attacks because the payload is encrypted by the parameters specified by the recipient in the reply block (this idea is shown on Figure 4, p. 27).⁴²

The reordering strategy used in Mixminion [15] is based on the *Cottrell mix* [13] (Section 4.3.3). To enhance resistance against blending attacks, the mixes add a random number of dummy packets each time they resend packets. This method is presented in [54]. The dummy packets are generated to travel a path of 1–4 mixes, and are discarded at the last mix on the path.

4.5 A Mixminion Based Solution

The Mixminion design [15] does not meet the requirements for YÅPS. There are two main problems related to the use of reply blocks. First, the use of SURBs requires all servers that are anonymous, to create and distribute a huge amount of SURBs. Second, even if I assume reply blocks can be reused, the use of key rotation to prevent replays creates a problem if the key rotation happens more often than reply blocks are changed, as these are based on the keys at the mixes.

YÅPS requires that reply blocks of servers have a long lifetime. Users can send a SURB to anonymous servers, and would therefore not need reply blocks with a long lifetime. In the following I propose a variant of Mixminion [15] that meets the requirements of YÅPS.

What are the space requirements for a cache and key-rotation scheme to prevent replay attacks? If I assume the anonymous servers update their reply blocks each month, key rotation can also be done monthly. If I furthermore expect that it might be necessary to keep the two latest keys at each mix, to reduce problems when the keys are rotated, each mix would need to cache two months of messages. If I assume an average of 100,000 packets a day, and an 8B hash value⁴³ is saved for each packet, about $61 \cdot 100,000 \cdot 8 = 48.8MB$ is needed for the cache. This space requirement is reasonable and I suggest a solution like this with key rotation and caching.

The use of a hash made from a packet header to prevent replay attacks also prevents servers from reusing reply blocks. It is therefore necessary that a recipient marks a reusable reply blocks, to make reuse of the reply block possible. For reusable reply blocks, a mix should not save a hash of the header in its cache, but rather save a hash of the message, as this prevents replays of packets with reusable reply blocks, but allows reuse of reply blocks.

The main problem with reusable reply blocks is that they can be used to mount replay attacks. The problem is reduced against link-based attacks if the communication between the mixes are encrypted—as I will suggest. In Mixminion this is done with the use of TLS, later I will provide a simpler solution (in Section 7.6).

⁴² An attacker that tags an R-packet will not be able to verify the tag later, as it is encrypted at the first rerouter it passes.

⁴³ An 8B hash value is a sufficient large hash value, as the change for a collision would be less than 1 to $4 \cdot 10^{13}$, and the damage of a collision limited.

In the most recent work on Onion Routing an approach named *rendez-vous* [22] is presented as a solution to recipient anonymity. In this approach, a recipient is listening for connections at certain points (on other servers), using anonymous communication connections, thus gaining recipient anonymity. These communication points must be advertised to potential senders, and senders can contact the recipient through the points to set up an anonymous communication path (using other rerouters than those used to make the contact).

The rendez-vous strategy can be modified to work with the messages based communication in mixes. A recipient announces an ID (these are explained in Section 7.1.1), and a list of *middleman-mixes*. I shall denote middleman-mixes as **m-mixes**). Users can send messages to a recipient by creating a packet with the message and the ID of the recipient, which is sent to one of the listed m-mixes listed by the recipient (for example, as an FR-packet to provide sender anonymity). Each m-mix is given a number of reply blocks by the recipient, these reply blocks are used to forward packets, which the m-mix receives for the recipient. The packets include the recipients ID, and the m-mix creates a new packet, using the payload and a randomly chosen reply block. The function of the m-mix is shown in Figure 5.

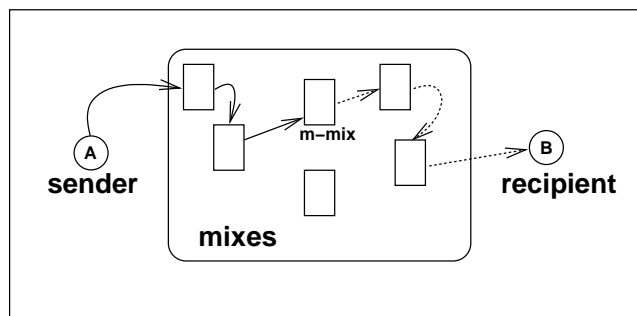


Figure 5 The m-mix. The recipient B have placed a set of reply blocks at the m-mix, and announced the m-mix to senders. A can send messages to B by sending them to the m-mix with the ID of B. At the m-mix the message is recovered, and a new packet is created and send to B using one of B's reply blocks.

In the following I refer to the path between the sender and the m-mix as *stage 1*, and the path from the m-mix to the recipient as *stage 2*.

The main advantages of the m-mix approach is that the recipient does not need to announce a reply block to senders and the sender can choose an arbitrary path for stage 1. The recipient can update the reply block at the m-mixes as often as she wishes. Furthermore, it is possible to make attacks harder by limiting the use of reply blocks (for example, by setting the maximum number of uses). However, this approach can only provide a security advantage over normal use of reusable reply blocks if the m-mix is not controlled by an adversary (as the m-mix possesses the reply block), but the window of misuse is still limited by the key rotation scheme.

If an attacker tries to mount an indirect attack based on the reply blocks by sending packets to the m-mix with the recipients ID (as presented in Section

4.3.1, p. 32) this would be made more difficult than an attack based on the possession of a reply block, as the m-mix might use different reply blocks for each incoming packet.

An attacker that can identify a reply block on stage 2 is still able to mount replay attacks as in other systems with reusable reply blocks. An attacker can identify reply blocks on stage 2 if she controls a mix on the path: as reusable reply blocks are marked, they can be identified.

When a sender, *Alice* needs to communicate with an anonymous recipient, *Bob*, and Bob expects an exchange of messages, Bob can send SURBs back to Alice to use for the next message to limit the use of Bob's m-mixes.

An alternative to the use of packets containing routing information is to set up a circuit to transport messages through (as in Onion Routing [48]). The initiator can send a *hello* to an anonymous recipient, who sets up a path, which ends with an *n-mix*. The ID of the n-mix is sent along with the hello to the server, using the m-mix. The recipient can then send an answer back, which contains information to set a path up along the way to the n-mix. The n-mix is then able to forward messages between the two communicating parties using the paths. The setup can consist of a symmetric key for each mix on the path, as well as an identifier. Layers of encryption would be removed on the path until the n-mix, and added on the path after the n-mix. The parties would need to exchange a symmetric key to use to encrypt the content, to have it protected at the n-mix. The advantage of this scheme seems to be that only minimal routing information must be sent along with the packets that use the circuit, which can save traffic. Furthermore, symmetric encryption can be used, which requires fewer computation resources than asymmetric cryptography. However, the advantage is however limited, as a non-circuit solution also requires information for routing and a circuit design is more complex. At each rerouter both solutions require information on where the packet should be forwarded to. The circuit solution has a state compared to a non-circuit solution where each packet is routed independently. I shall not propose a solution that uses circuits.

Another problem with Mixminion [15] is the potential unlimited latency in the mix strategy. This strategy can easily be changed by putting time limits on incoming packets and force a mix to forward the packets after the time limit has expired. This time limit reduces security⁴⁴, for example, the limit makes it easier for attackers to mount attacks based on reply blocks (Section 4.3.1, p. 32). I believe that to provide an acceptable latency, the average time a packet is pooled must be limited to 20 seconds on average. This limit results in an average latency of 150 seconds for paths between 5 and 10 mixes. In practice the time limit shall be chosen randomly to reflect a normal distribution. If a mix receives few packets, it must add cover packets (like in Mixminion) to its pool to prevent adversaries from mounting correlation attacks. I believe more analysis is needed to determine exactly how much cover traffic a mix should add.⁴⁵

⁴⁴ Exactly how much it reduces security is not clear. I believe, that the paper *Stopping Timing Attacks in Low-latency Mix-based Systems* by M. Wright, B. Levine, M. Reiter, and C. Wang (to appear in Financial Crypto'04), will provide some answers to this question.

⁴⁵ I speculate that at least 10-20 packets must be kept in the pool at all times.

In short I propose a mix design based on Mixminion [15], but with the following changes:

- Limited key rotation, the key rotations should be done no more often than anonymous servers can update the reply blocks at the m-mixes.
- Reply blocks can be reusable and it is possible to distinguish packets with a current header based on a reusable reply block from other packets.
- Reply blocks are distributed to m-mixes and potential senders are given storage server IDs and m-mix IDs.
- M-mixes forward packets to anonymous recipients by creating a new packet using a randomly chosen reply block for the given storage server ID.
- The time a mix can hold a packet is limited.

These changes introduce changes into the different types of packets, created by a sender to an anonymous recipient using an m-mix:

- *FR-packet by m-mix*: (Stage 1) a sender generated mix-route to the m-mix (stage 2) storage server ID.
- *R-packet by m-mix*: Special case of FR-packet, where stage 1 is only one hop.

The protection of these packets against tagging works as follows: if there have been changes to the payload or headers in a packet before the m-mix, the packet cannot be delivered, because the mix-route or ID is unavailable. After the m-mix the packet is protected like in Mixminion [15]. The m-mix resends the incoming packets as a normal R-packet.⁴⁶

4.6 Summary

I have presented a number of theoretical and practical solutions to the problem of anonymous communication. I found that rerouting based solutions provides the most efficient solutions and that a modified mix-based solution can provide better security than an OR rerouting solution.

Mixminion [15] is a promising practical mix-based design, but it does not fulfill the requirements of YÅPS, I therefore proposed a variant of Mixminion. The variant allows reply blocks to be reused, but allows easy updates of reply blocks by using m-mixes as an indirection when recipient anonymity is required.

The proposed solution requires a number of servers that can function as mixes, as well as infrastructure to let users of YÅPS and the servers in the system know about the mixes.

⁴⁶ Another possibility is to resend the packet as a normal FR-packet. This makes it harder for an attacker to make indirect attacks before the m-mix, using the reply block. More analysis is required to reveal the overall effect of this regarding security.

5 Secret Sharing

You're setting up a launch program for a nuclear missile. You want to make sure that no single raving lunatic can initiate a launch. You want to make sure that no two raving lunatics can initiate a launch. You want at least three out of five officers to be raving lunatics before you allow a launch.
—Bruce Schneier, *Applied Cryptography* [53, Section 3.7].

I shall now discuss how strong deniability can be implemented using secret sharing. I shall start by giving an overview that states the requirements of YÅPS and introduces secret sharing (Section 5.1), then present schemes for secret sharing in Section 5.2 and Section 5.3. This is followed by a discussion in Section 5.4.

5.1 Overview

The goal is to provide a storage-server operator with the ability to deny knowledge of the content stored on the operator's server. YÅPS protects storage servers with anonymity (Section 3.1), this protects against adversaries who tries to find the identity of servers where something is stored. Another consideration is adversaries searching storage servers for illegal material. To defend against such attacks, server operators need the ability to deny knowledge of the content on the server. Most APSs provide this protection by encrypting the content, but this approach can fail if the adversary can get the decryption key.

I propose another approach, using secret sharing, which makes it impossible to link the content on a storage server to any document. Even if an adversary can publish a document D , by storing a block B , and later finds B on Bob's server, the adversary cannot show that the B on Bob's server probably originates from D . If the adversary can prove that the publication process *did* store B on Bob's server, this defense would probably not work—the anonymity of Bob's server should however prevent such an attack.

This approach, using secret sharing, makes it impossible for Bob to know anything about the documents that the blocks on his server originates from—even if Bob is told what documents they can be used to recreate. If I have information to recreate a specific document X , from content stored on Bob's storage server, I will not know whether the content on Bob's server did originate from that document. For example, someone could have produced my information, as anyone would be able to create information, which showed that part of document X is stored on Bob's server.

In YÅPS the documents must be split into blocks of a fixed size. These blocks are transformed by secret sharing, and the resulting blocks are stored at the storage servers. I shall use the term *share* to describe the transformed blocks in some of my explanations, to clarify the difference from the original blocks.

Secret sharing is used in Publius [38] to distribute encryption keys for the published documents. Secret sharing is also used in *Tangler*[61], a centralized APS, to provide **entanglement** of documents. The entanglement in *Tangler* prevents censorship by removal of blocks, because removal could affect a number of

documents. The idea of Tangler has been discussed in relation to plausible deniability, because it removes the relation between retrieving a specific share and a specific document,⁴⁷ but to protect the users (not the server operators).

I shall also propose the use of entanglement, but with the intention of making it hard to show what content the blocks on a storage server originates from.

I shall now present two schemes for secret sharing, and show how they can be used in YAPS to implement strong deniability. The exclusive-OR scheme I present in Section 5.2 is a simple scheme, sometimes referred to as **secret splitting** [53, Section 3.6], this belongs to a subset of (k, n) -threshold schemes (it is a $(2, 2)$ -threshold scheme), as I present in Section 5.3. (k, n) -threshold schemes are more general than secret splitting, they support a more efficient way to allow a random subset of shares to be combined to recreate a secret. Threshold schemes are a subset of generalized secret sharing schemes. Generalized secret sharing schemes provides a set of shares where it is possible to recreate the secret only from specific sets (this functionality is not relevant for the requirements of strong denial).

5.2 Exclusive-OR Scheme

A simple way to transform a block into shares is to use bitwise exclusive OR (XOR) with a random block. This scheme is a variation of the **one-time pad** [39, Note 6.1], an encryption algorithm that provides unconditional security (confidentiality). Given a plaintext P , this is transformed into a ciphertext C , by xor'ing a random key k , the same length as P , to P (I use \oplus to represent this operation):

$$C = P \oplus k.$$

Decryption is trivial because the XOR operation with k is its own inverse:

$$P = C \oplus k.$$

The security is unconditional because given a ciphertext, there exists a key for any possible plaintext (the key space is the same size as the plaintext space, and each different key maps to a different ciphertext).

I have no requirement for confidentiality, but require that the possession of a block cannot be linked to the possession of a piece of a specific document. This requirement is fulfilled by an exclusive-OR scheme: given a block B , and a random share s_1 , another share s_2 can be created:

$$s_2 = B \oplus s_1.$$

⁴⁷ Roger Dingledine of the Free Haven project, discusses entanglement on the development mailing list, April 15 2001, <http://archives.seul.org/freehaven/dev/Apr-2001/msg00007.html> (November 2003).

B can be discarded, and the two shares s_1 and s_2 stored on different storage servers.

The shares are unlinkable with any specific block, because for any share s , and any block, b , there exists a share s' , that links them:

$$\forall s \forall b \exists s' : s' = B \oplus s.$$

Note that s' can easily be generated.

This unlinkability implicates that a share cannot be linked to a specific document. Given a share, nothing can be said of the document (or block) the share originated from. Even if s_1 is not random, s_2 can still not be linked to a document. It is therefore easy to create shares that are entangled with other blocks: an existing share can be obtained and used to create a new share.

This scheme is **perfect**, as no knowledge of a block can be deduced from the shares of the block (I proved this). Furthermore, each share are the same size as the original block, making the scheme **ideal**. The total amount of space needed for recreation of a block is two times the size of the block, this is an absolute lower bound for a perfect secret sharing scheme. This is a lower bound because (1) the share resulting from the XOR must be able hold the same information as the original block, and (2) the XOR is a mapping defined by a share, and the mapping must be able to produce all possible shares, for blocks of length n bits, as there are 2^n possible shares, at least n bits are necessary to represent them all.

5.3 A $(2,n)$ -Threshold Scheme

In the following I shall use Shamir's threshold scheme [55] as an example of how a threshold scheme can be used to provide strong denial. In the end of the section I shall briefly discuss the used of threshold schemes in general.

A (k,n) -**threshold scheme** is used to divide a piece of data into n pieces ($n > k$), such that the data can be recreated from only k pieces. For my purpose there is no reason to require more than two shares to recreate a block. But there might be a reason to create more than two shares to support redundancy, which is why I need a $(2,n)$ -threshold scheme.

Shamir's threshold scheme is based on polynomial equations in a finite field \mathbb{Z}_p . Given a block B , of size N bits, interpreted as a number N_b , a random prime $p > N_b$ is generated, as well as a random coefficient $a < p$, to create a polynomial of first degree in \mathbb{Z}_p :

$$y = ax + N_b$$

Any number of shares ($< p$) can be obtained by evaluating the polynomial for different values of $x > 0$. The share is then the two values (x,y) . a is discarded after the generation of the shares. Using two arbitrary shares and p , N_b can be recreated, because the shares provides two distinct points of the polynomial.

The two points makes it possible to recreate the coefficient a , and this can be used to get N_b by evaluating the polynomial for $x = 0$.

Example: Given the block with $N_b = 42$, I choose $p = 43$, and $a = 7$, and get the polynomial $(7x + 42) \bmod 43$. I can then create a number of shares: $(12, 40)$, $(13, 4)$, $(14, 11)$, $(15, 18)$. Given $p = 43$, and two shares: $(12, 40)$, $(13, 4)$, I can create two equations: $(a12 + N_b) \bmod 43 = 40$ and $(a13 + N_b) \bmod 43 = 4$, and calculate $N_b = 42$.

One way to introduce entanglement is to create the polynomial as described, and then obtain an existing random share and use this to calculate the x value corresponding to the share. The problem of this method is that it would result in an x value in the range $0 < x < p$. On average x would be of size $N - 1$ bits. Because the x value must be stored somewhere, this approach requires significant additional storage. A better solution is to start by choosing p and use the share to calculate the coefficient for a polynomial. The polynomial can then be evaluated for other values of x as usual.

Example: Given $N_b = 18$ and one of the shares from the previous example, but without the x value (as it is not stored, this is explained below) 40 , I choose $p = 53$ and $x = 3$, and get the polynomial: $(a3 + 18) \bmod 53 = 40$, resulting in $a = 25$. The polynomial is then $(25x + 18) \bmod 53$, and I can use this to create the additional shares.

There is no requirements for the x values. For example, the x values could be fixed to a (small) number.

Shamir's threshold scheme is both perfect and ideal. That it is ideal is easily seen. A proof that shows it is perfect is given in [55], it is obvious for the $(2, n)$ -threshold scheme I have explained: if each shares is seen as a point in a two-dimensional coordinate space and the secret as the slope of a line—a single point cannot reveal anything about the slope of a line running trough the point. Even though Shamir's threshold scheme is ideal, it uses more space than the exclusive-OR scheme because the x values and p are also needed. As the p value is about the size of a block, the scheme requires three times the space as the original block in practice. Other schemes might in theory reduce the size of the parameter corresponding to p , but not the size of the shares.

There seems to be an advantage of the proposed threshold scheme, as it has built-in support for redundancy: any number of shares can be created, and two arbitrary shares can be used to recreate a block. However, the support for redundancy uses as much space as replication as shares have the same size as the original blocks. Furthermore, the redundancy support still requires p to be replicated.

5.4 Discussion

I have presented an exclusive-OR scheme and a $(2, n)$ -threshold scheme to implement secret sharing. The two solutions do not influence the rest of the design in any significant way, they provide the same security, but the $(2, n)$ -threshold scheme uses more space because space corresponding to an additional share is needed to store p . The p -values might be reused randomly to lower the space requirements, but users would still need to retrieve 50% more

data with the $(2, n)$ -threshold scheme, compared to the exclusive-OR scheme. This difference as well as the simplicity of the exclusive-OR scheme makes me choose the exclusive-OR scheme for YÅPS.

Strong denial is realized using secret sharing to generate the blocks, which are published. Because the publisher is responsible for the secret sharing, there is a possible attack on this method: what if a user does not secret share the blocks she published? The storage-server operator can still use the defense outlined in Section 5, but if the share in itself was a piece of a document, for example *The Holy Bible*, the defense will probably not work well (because it will be considered a strange coincidence that a random block is identical to a piece of a document). A storage-server operator can refuse to store shares that seem to have a low entropy, but this can be countered by an adversary by encrypting the share before it is published. An adversary can then use the key to demonstrate what the share contains. Even a multi-level secret sharing scheme, with secret sharing done by the user as well as at the storage servers, will not work. The storage servers need to store their shares on other servers, which again would need to use secret sharing, and so on. This problem indicates that strong denial is a second-line defense after anonymity (this is also mentioned in Section 3.5), and that storage-server operators, whom relies on the strong denial as a defense operators must restrict the access to trusted publishers.

6 Location

If you don't find it in the index, look very carefully through the entire catalogue.

—Sears, Roebuck, and Co. Consumer's Guide, 1897

In Section 3.9 I described that YÅPS will be based on a distributed system of storage servers. A publisher can generate a set of blocks using secret sharing as described in the previous section (5), and distribute these to a number of storage servers. In this scenario, the question is how the document can be retrieved, and more specific how can the blocks be located?

In the following I shall discuss different solutions to this problem. I give an overview in Section 6.1, where I also define two categories of location, based on the identifiers. In the following two subsections I discuss these: location based on *loose keys* in Section 6.2, and location based on *tight keys* in Section 6.3. I present Chord [57] in Section 6.4 and end with a discussion in Section 6.5.

6.1 Overview

If I want to retrieve a document from YÅPS, I need (1) access to information about how I can recreate the document from a set of blocks, and (2) information on how I can retrieve the necessary set of blocks. In the following I shall refer to the information a user needs to retrieve a document as a **recipe**. In short, the two location problems are:

- How can blocks be located?
- How can documents be located?

A simple solution to the problem of locating blocks is to let the publisher save the information about the servers that store the blocks in a recipe. This information can then be announced to users that must be able to recreate the document.

There are two disadvantages of this solution. First, the location is fixed, this makes movement of blocks between storage servers impossible. Second, the publisher must distribute the information to the people who wants to retrieve the document. The last problem can be reduced if the recipe and a description or a list of keywords is saved together, this would enable users to search the system for interesting documents.

The general location problem can be described as follows: values are identified by a key, a name for example, and the problem is to map keys into values. In the following I shall discuss solutions to the problem of locating recipes and blocks.

A solution for YÅPS is required to be decentralized and efficient, and also it must be scalable and secure. This implies that anonymity and the ability to provide strong denial must be preserved. Furthermore, the solution must be able to work in the expected environment, with unstable servers and limited resources (Section 3.2).

The location problem is traditionally solved in two different ways. I shall discuss those based on the way keys are used:

- **Loose keys:** Arbitrary keys that can be chosen freely. For example a list of keywords.
- **Tight keys:** Keys that are fixed and bound to content or location.

In systems with loose keys, searches can be made without having an exact key, for example, users can be allowed to search on substrings of keys or lists of keywords. Loose keys can solve the problem of locating recipes (as well as blocks). Systems that provide only searches on tight keys, require an exact key and location. Location by tight keys can be solved with lookup tables, which maps keys to locations. Tight keys can be used locate blocks, but cannot make it possible to search for recipes or documents.

6.2 Location Based on Loose Keys

Gnutella [35] provides a simple decentralized solution to the location problem using flooding of requests to all servers. In Gnutella all the storage servers are connected (directly or indirectly through the direct connections), and searches are done by flooding all neighbors with requests. Each server stores files indexed by filenames and keywords. A server, which receives a request for something it stores, issues a reply. Gnutella provides a robust solution, as no index separate from the storage servers exist, and searches for anything that is currently available is expected to be successful. The problem with location in Gnutella is that the system does not scale well [37], [51].

In [37], attempts are made to optimize Gnutella, it is shown that location based on multiple random walks resolves queries slower than flooding, but also reduces the traffic significantly. Gia [9] is another alternative, which provides a more advanced infrastructure than the random Gnutella network. In Gia the concept of using random walks is combined with (1) a dynamic topology adaption protocol, that strives to let high-capacity servers handle most random walks, (2) flow control to avoid overloading, (3) replication of location information to a server's neighbors. These optimizations takes advantage of the spreading and replication of files in Gnutella-like systems.

GNUnet [4] provides a more secure version of searching with keywords. In GNUnet documents are stored under a number of keywords, but only a secure hash of the keyword is saved together with the document. Instead of keywords, a search contains the hashes corresponding to the keywords. While this makes it hard to produce a list of the keywords stored at a server, it is still easy to verify if documents (or parts of documents) stored by specific keywords are present. The queries in GNUnet is done with limited flooding.

As I shall discuss in the next section, Freenet [11] is also an optimized variant version of flooding based solution.

The advantage of location based on loose keys is the possibility of searching. Searching makes it possible for users to retrieve documents from the system

without using other channels to obtain recipes. There are however two disadvantages. First, in a decentralized system, searching is inefficient and does not scale well. Second, searching based on loose keys are based on the servers knowledge of the content they store (as the servers needs to verify searches on freely chosen keys), this is a problem because it weakens the servers defense based on denial.

The optimized systems based on Gnutella assumes a wide replication of data to work, and does not support plausible denial. GNUnet [4] proposes a partial solution to the second problem, but it prevents defenses based on denial. Current solutions, which provides loose keys, do not fulfill the requirements of Y&A for security and efficiency.

6.3 Location Based on *Tight Keys*

In Free Haven [21], all the blocks belonging to a document is identified by the same ID. Users flood the system with requests for blocks matching a document ID to recreate a document.

Freenet [11] is based on caching of documents, and limited flooding. Freenet servers are organized in a network where each server keeps track of a number of neighbors. Each Freenet server keeps a routing table of recent requests, and also caches recently requested documents. Each document is identified by a key, which is a secure hash of the content. A search for a document is done by a modified flooding strategy, using the keys. The request is forwarded to the neighbor, which has earlier found documents with a key close⁴⁸ to the key in the current request. The *closest* neighbor is tried first, but if the request is unsuccessful, that neighbor is removed from the set of neighbors to try and the request is send again to the closest neighbor. The insertion of documents is done by a search, and the document is inserted on a server with similar keys. When a document is found it is returned by the path of the request, it is also cached on the servers on the path. This caching entails that Freenet is a self optimizing version of the flooding strategy. The scalability of Freenet is an open question, experience⁴⁹ reveals a high latency system with limited bandwidth. The caching and routing in Freenet does not help much if files are requested at random from random entry points, which I speculate is part of the explanation for the slowness of Freenet. In short, location in Freenet is based on tight keys and restricted flooding. Freenet keys are mapped directly to documents, but the location strategy can also be used to map keys into locations (this would however cause problems with updates of location, because Freenet uses a caching strategy).

An alternative solution to flooding-based strategies, is to provide a lookup table that can map keys into values, for example, IDs of blocks to locations. The location information can then be updated by the storage servers to reflect changes in the storage. In a centralized system this solution can easily

⁴⁸ Closeness is based on a distance function on keys. If the key is interpreted as a number, the distance can be expressed as the absolute numerical difference between keys.

⁴⁹ I have found no studies on the practical performance of Freenet, but personal experience and studies of other users experiences suggests Freenet have problems with speed. See mails on the *freenet-support* mailing list (November 2002), <http://www.mail-archive.com/support@freenetproject.org/msg01955.html> (November 2003).

be implemented by the use of a tree-based data structure or a hash table. For a decentralized system the solution must scale well with the number of keys and servers. The number of operations to complete a lookup must not exceed $O(\lg n)$, for n lookup servers (I use \lg to denote binary logarithm). The system must also support servers leaving or joining the system, and the system must be able to recover if servers fail or disappear.

In the following I shall present designs that can be used to provide the functionality of a distributed lookup table.

One of the first routing algorithms that could be used to implement distributed lookup tables was developed by Plaxton, Rajaraman, and Richa [44]. I shall denote their approach as **Plaxton routing**.

Plaxton routing [44] is based on the use of so-called **PRR-trees** (due to the researchers last names). In Plaxton routing all the servers are connected by letting each server have a list of neighbors. A request received by a server is routed toward the neighbor with an ID that most closely matches the ID in the request. A server implements this routing by forwarding requests to the neighbor with the longest matching prefix of the ID. In a system with n servers, each server must know $O(\lg n)$ neighbors. The routing path is $O(\lg n)$ servers, which means that it is possible to reach any ID after visiting $O(\lg n)$ servers. I have not found an implementation of a system with Plaxton routing. Plaxton routing is complicated and does not handle concurrent joins of servers or failures. Plaxton routing is designed to work in a static environment, with a fixed set of servers—it is therefore not suitable for YAPS.

Pastry [50], is another prefix-based routing scheme. Each server is given a random ID that is in the key space. The key space is considered to be a ring. Each server is responsible for the keys numerical closest to the servers ID. Each server keeps two sets of neighbors: a set of the closest neighbors (smaller and larger, the highest ID is neighbor to the lowest ID), and another set—based on PRR-trees—with servers spread out in the key space. Only the closest neighbors are needed for correctness, the other set is used to make searches faster. Routing is done by forwarding a request to the server that best matches the key in the request. Pastry, like Plaxton routing, requires each server to keep routing information concerning $O(\lg n)$ other servers, and the routing path is also $O(\lg n)$.

Tapestry [62] is another scheme based on PRR-trees. Tapestry works almost like Pastry [50], and provides the same bounds, but is more fragile and more complicated.

Content-addressable networks (CAN) [47] uses a d -dimensional Cartesian coordinate space to implement a distributed lookup table. Each server is required to maintain $O(d)$ state and lookup requires $O(d \cdot n^{1/d})$ operations. Compared to the other presented systems, the states in CAN do not grow with the size of the network, but the lookups scales worse than $O(\lg n)$ (for the special case $d = \lg n$, lookup requires $O(\lg n)$ operations, and CAN is comparable to the other systems presented). The value of d is fixed, but can be adjusted to a specific design. CAN requires a remapping protocol to keep (key, value)-pairs on the right servers.

Chord [57] is another system, where the servers are organized in a ring in an ordered way—like in Pastry. The ring is an ordered list, sorted on the IDs of the servers and each server knows its successor as well as $\lg n$ other servers. The concept of the ring is shown on 6. Each server is responsible for the keys close to its ID. Given this ring, any key can be found starting from any server. Like Pastry [50] and Plaxton routing [44], each server keeps routing information concerning $O(\lg n)$ other servers, and lookups can be done by a path of $O(\lg n)$ servers. Chord is based on a simple infrastructure and supports a dynamic environment within the bounds on state and lookup.

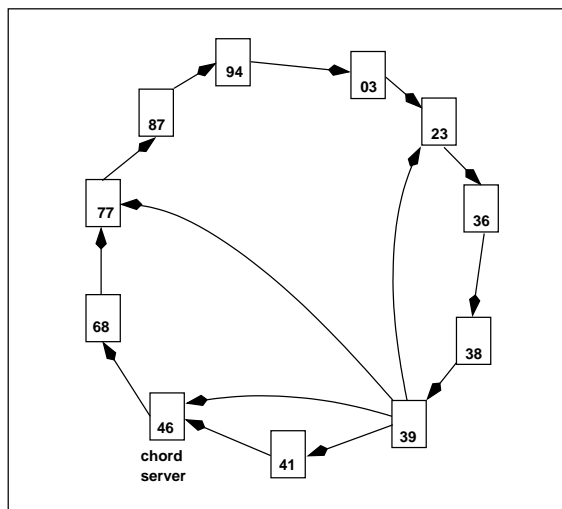


Figure 6 In Chord each server knows its successor, shown with arrows in the figure. Furthermore, each server keeps a table with information concerning $\lg n$ other servers, these are shown on the figure for server 39.

Chord [57] is as efficient as the other proposals (with the exception of CAN in special cases) and it supports a dynamic set of servers and failures well. Furthermore the design is simple, and have been proved in theory and tested in practice [58] and is used in the implementation of *The Cooperative File System* (CFS) [14]. I shall use Chord in YÅPS. Chord is described in detail in the next section.

6.4 Chord

Chord [58, 57] is based on an ordering of the servers in a ring. The ring is sorted on server IDs. If the size of the IDs are m bits, servers have an ID in the interval $0, \dots, 2^{m+1} - 1$. Each server knows its successor. The successor of $2^{m+1} - 1$ is 0, which makes the ordering a ring. Each server is responsible for the keys in the interval between its own ID and the successor nodes ID (including its own ID). Given this ring, any key can be found starting from any server, because there is a link to all servers through the successors. A lookup, which only uses successors to locate a key, requires a path of 0 to $(n - 1)$ servers for a ring with n servers. If each index server knows all the 2^p th successors, (successors with

a distance of 1, 2, 4, 8, 16, . . . servers) and a request is routed to the server with the highest ID less than the requested key, then the path is reduced to $O(\lg n)$ servers.

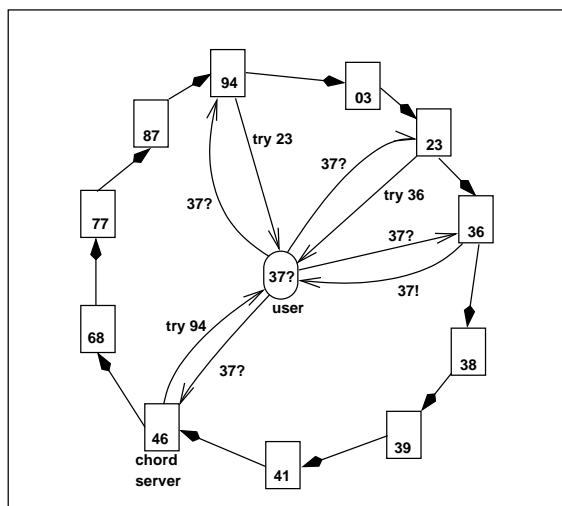


Figure 7 A lookup in Chord is fulfilled by letting each server either: (1) return a value or an error message if the server is responsible for the requested key or (2) return the address of the server closest to the key (the current server knows of). In the figure the user knows Chord server 46 and requests key 37. The request is routed closer to the correct server for each request, as each server responds with the server nearest to the key, until it is found.

When a new server enters the ring, the keys in its key space is transferred to the server. The protocol for joins is straight forward: a new server r finds its successor t , and copies the keys belonging to its own key space from the successor. The only update left is for the predecessor s to update its successor from t to r . The successor updates are done by having all servers periodically run a stabilize algorithm. The stabilize algorithm checks the successors information about its predecessor (the join operation updates this) and the server will find out if a new successor have joined. Join operations cannot result in failed requests, but can degrade the efficiency of lookups.

If a server fails, its predecessor will have wrong information about its successor. Chord reduces this problem by requiring each server to keep a list of about $O(\lg n)$ successors. As n can change, this can be done by keeping lists of length $\lg N$, where N is the maximum expected number of servers.⁵⁰ The use of a successor list requires slight modifications of the stabilizing scheme, because the list of successors needs to be kept updated. Basically, a server copies its successor's list, drops the last element, and adds the successor to the front. Over time the lists will be corrected in scenarios with a long time between changes, but if there are ongoing changes, the lists will rarely be correct. However, this does not deteriorate correctness or even performance in a significant way.

⁵⁰ Although it would also be possible to count the number of servers by sending a counter with the ring.

Chord does not map keys to values, but rather keys into servers responsible for an interval of keys (this is not different for the other solutions discussed previously). Chord does not guarantee that keys, which are inserted, can be retrieved again, only that the current server responsible for the key can be located. A system that uses Chord, must be able to provide redundancy of the keys stored in Chord, if redundancy is needed.

Chord can be used to map keys to locations. In YÅPS this can be used to map block IDs to contact information for storage servers. Given a recipe, which contains a set of block IDs, a user can locate the blocks on the storage servers, using Chord.

Chord can also be used to implement keywords, but it would then be possible to verify the content of a server, using the keywords. YÅPS will not support this.

6.5 Discussion

Searching on loose keys requires a mapping from keywords to documents at the server that has to answer queries. An operator of a server with this information cannot use denial as a defense, as she is able to check find out what her server stores information about. Denial is an important defense in YÅPS, and therefore YÅPS does not make searches possible.

I chose Chord [57] to provide a lookup table, which can be used to efficiently map block IDs to locations, and furthermore, allow storage servers to update information on block locations. Such updates would allow more flexibility in the storage of blocks.

The use of Chord for location requires a number of servers to run Chord. I shall refer to these as *index servers*. Furthermore, infrastructure is needed to keep the index servers updated to reflect the blocks at storage servers. There is also a requirement for redundancy of the location information at the index servers, as Chord does not provide this, I shall return to this issue in Section 7.4

7 Miscellaneous Design Issues

Dost thou think, because thou art virtuous, that there shall be no more cakes and ale.
—Shakespeare

In the preceding sections I have taken decisions that defines the components in YÄPS. YÄPS consists of:

- *Storage servers*: Documents are divided into blocks. The blocks are split into shares using secret sharing, and stored on the storage servers.
- *Rerouting servers*: Anonymity is provided by a rerouting network consisting of rerouting servers using a variant of the Mixminion mix network [15].
- *Index servers*: Blocks on the storage servers are located using index servers based on Chord [57]. These servers map the ID of a block into location information, which is returned to users.

I shall now discuss open problems remaining from the preceding sections on anonymous communication, secret sharing, and location, as well as remaining issues related to the goals. The following sections discuss:

- *Section 7.1*: How blocks and servers are identified.
- *Section 7.2*: How redundancy of blocks and location information is implemented.
- *Section 7.3*: Which servers should be provided with anonymity?
- *Section 7.4*: How index servers can authenticate updates and how the location information in the Chord infrastructure can be replicated.
- *Section 7.5*: How storage servers can be found by users and how restrictions can be supported. And also block size and lifetime of blocks.
- *Section 7.6*: How information about the rerouting network can be provided to rerouters, users, and other servers. Furthermore the availability problems with the rerouting network are discussed.
- *Section 7.7*: What happens if servers fail?
- *Section 7.8*: How updateable documents can be implemented.
- *Section 7.9*: How recipes can be shortened to make it convenient to hide or communicate them.

I shall give a summary in Section 7.10.

7.1 Identification

Servers and blocks must have a unique ID. The ID makes it possible to locate blocks and servers, and also provides integrity protection for blocks, as well as a way to authenticate servers.

7.1.1 Server IDs

Servers must have an ID, which can be used for authentication. The ID must not contain information, which can help adversaries to identify the servers location or the operator of the server. Such an ID can be made using asymmetric cryptography:

- Each server generates an asymmetric key pair.
- The ID of a server is a secure hash of the public key.

The key pair represents a pseudonym of a server. This ID-scheme is similar to the *node identities* [27] in GUNet. For example, servers can authenticate themselves using digital signatures.

I discuss the use of a secure hash function to produce IDs in the next section.

7.1.2 Block IDs

The block IDs must uniquely identify blocks. Another consideration is integrity protection of the blocks. The blocks must be protected against tampering by adversaries and as the IDs are fixed values, it seems obvious to base the ID on the content. Furthermore, a block must be able to have multiple IDs to support replication of the block as well as the information needed to locate the block.

I shall base the ID on a secure hash function to provide integrity protection, and use a salt [30, Chapter 8] to provide multiple IDs, $x|y$ stands for the concatenation of x and y :

$$\begin{aligned} \text{ID}_0 &= h(\text{block}) \\ \text{ID}_n &= h(\text{ID}_0|n) \text{ for all } n > 0 \end{aligned}$$

Where the salt n is interpreted as a number.

Given a block or the ID_0 of a block, any ID_n can be generated in constant time.⁵¹ The reason that ID_0 is a special case is because it must be possible to create other IDs from this. From ID_0 any ID can be information to locate a block, because other IDs can be generated from this. I shall use **ID number** to denote the salt n in ID_n .

⁵¹ An alternative method is to use iterated secure hashing ($\text{ID}_0 = h(\text{block}), \text{ID}_n = h(\text{ID}_{n-1})$ for all $n > 0$), this method allows the creation of any ID_s from any ID_n , where $s \geq n$. This method is more elegant, but slower.

The use of secure hashing produces IDs, which are expected to be spread uniformly in the ID space. The spreading of IDs helps to map the IDs into different index servers.

I recommend the use of the secure hash function SHA-1[39, Section 9.4.2, iii], which produces a 160 bit digest. The ID space of 160 bits makes sure IDs are not duplicated.⁵²

7.2 Redundancy

I have described how documents can be split into a number of blocks and how secret sharing is used to create a new set of blocks. The problem is to store these blocks in a redundant way on the storage servers to allow the retrieval of the document if some of the storage servers fail.

There are two approaches to introduce redundancy: (1) replication of blocks, and (2) using error correction codes to construct a new set of blocks with redundancy.

Replication is a simple strategy, where each block is stored at multiple storage servers. The disadvantage of this approach is that it requires a specific subset of storage servers to be available: if a document requires n blocks to be recreated, and each block is replicated p times, only p^n combinations out of the possible $\binom{p \cdot n}{n}$ can be used to recreate the document.

An alternative approach is to use an **information dispersal algorithm** (IDA), such as Rabin's⁵³ [46], in the following I refer to this as **IDA**. If a document requires n blocks to be recreated, these can be used with IDA to create $n \cdot p$ blocks, where all $\binom{p \cdot n}{n}$ combinations can be used to recreate the document. IDA is an optimal solution regarding redundancy, as at least n blocks are needed to recreate the document, and all possible combinations are usable.

Example: If I want to publish 6 blocks, and replicate each block 10 times (using 60 blocks of space), there are 10^6 subsets of servers that allows me to recreate the document. If I used IDA instead, I can create 60 distinct blocks using the same space as before but will now be able to recreate the document from any subset with 6 distinct elements. That is, all $\binom{60}{6} > 4.8 \cdot 10^{77}$ possible subsets with 6 elements.

The example shows that replication requires more space than IDA to provide the same redundancy and suggests that IDA shall always be used to provide

⁵² Even if an amount of 1 PB (10^6 GB) is stored in the system, requiring twice the space (caused by the use of secret sharing), this is not a problem. If the block size is assumed to be only 2000 bytes, and each block is saved 100 times to provide redundancy (requiring a total amount of 200 PB space), each with an ID, the total number of IDs would be less than 2^{47} ($2 \cdot 10^{15} B / 2000 B \cdot 100 = 10^{14} < 2^{47}$). This set of IDs is a tiny fraction of all the possible IDs, as there are 2^{160} possible IDs. A random created ID would already exist with probability of 1 to 2^{113} . In comparison, the probability of me being struck by lightning *and* winning the big lottery in the same day [53, Section 1.7] is estimated to be something like 1 to 2^{55} , that is 2^{58} (or around 10^{40}) times greater than the probability of picking a string that would hash to an existing ID.

⁵³ Rabin's IDA uses calculations in a finite field \mathbb{Z}_p . A string S of length L elements in \mathbb{Z}_p is divided into n sequences of size m . n vectors of size m are created, where each subset with m elements are linearly independent. The vectors are combined with the sequences to produce n sequences of size m . From an arbitrary subset with m of the resulting sequences, S can be recreated. The field $\mathbb{Z}(2^8)$ can be used to work efficiently with bytes, as it has 256 elements. I shall use IDA in $\mathbb{Z}(2^8)$ with a fixed size of m equal to the block size.

redundancy. However, there are issues with IDA, concerning secret sharing, as I shall discuss in the next section.

7.2.1 IDA and Secret Sharing

The problem with IDA [46] is that it results in a new set of blocks that can be shown to intuitively belong together: secret sharing results in two sets of blocks: s_a —the random blocks used for secret sharing—and s_b —the resulting shares. If IDA is just applied to the superset $s_{ab} = s_a \cup s_b$ to create a new set of blocks S_{ab} , then an adversary, who possessed the recipe, would be able argue that these blocks all originated from the same document. As it would be a strange coincidence if someone could make such a recipe based on random blocks in the system, this argument would probably win over other explanations (this is the same problem as I discussed in Section 5.4).

The problem can be solved if IDA is applied independently on the two sets: this results in two new sets S_a and S_b . The blocks in each of these sets can also be shown to belong together, but each sets can only be shown to belong together to produce a random string. An adversary can produce the strings corresponding to s_a and s_b , but will not be able to show that the strings are related. Furthermore, the structure in IDA makes it possible to (re)create a random string from any random set of blocks, this implies that a block can belong to a random string of any size (and it would be easy to create a recipe for any document that included a specific block).

If IDA is applied independently on the two sets of blocks, there is no reason to retrieve existing blocks for secret sharing. The primary reason to use existing blocks is to save space, but no space is saved if IDA is applied independently on the two sets of blocks (as this results in two new sets of blocks). Instead, a publisher should just generate a set of pseudo-random blocks for secret sharing.

The reason IDA cannot be used as secret sharing by itself is that it is not easy to show how a block created by IDA can be used to recreate any possible document.

7.2.2 Discussion

IDA [46] can provide space-efficient, redundant storage of blocks, but is impossible to combine directly with entanglement. Replication is a simpler way to implement redundancy and replication makes entanglement possible. In later sections I shall show how to shorten recipes (Section 7.9) and how updateable documents can be supported (Section 7.8). Both short recipes and updateable documents requires the use of replication. If random blocks are retrieved for secret sharing, then these might also be necessary to replicate, depending on how they are already replicated (a user can check this).

If IDA is used, each of the resulting blocks have distinct IDs, which must be stored in the recipe. A replication strategy needs to save only ID_0 for each block, and the highest ID number used, corresponding to the number of replicas. Each storage server is then responsible for a replica identified by an ID number.

Publishers can make contracts with storage servers to generate new replicas if the number of replicas decrease below a defined threshold. Dynamic creation of replicas is easy for blocks that were published with replication. For blocks saved with IDA it is not possible to recreate lost blocks or add new blocks without the recipe. Instead, storage servers can watch a small subset of blocks and simply replicate the blocks if the set shrinks below a threshold defined by the publisher.⁵⁴

If a user wants to retrieve a block, the block must be available from a storage server and the index server must be able to map the ID of the block into storage server.

A publisher can make one or more storage servers responsible for a block, this depends on the scheme used for redundancy. If exactly one storage server is responsible for a block, the storage server can create multiple IDs for a block, and update the location information at the different index servers. If multiple storage servers are responsible for a single block, the publisher can give each storage server responsibility for an interval of IDs (defined by the ID numbers).

The schemes presented to support redundancy of blocks do not require any support from the servers, only of the client programs. The clients software in YÅPS must support both IDA and replication.

7.3 Anonymity

Users and storage servers must be protected to resist censorship and it is not possible to protect the rerouters, as these must be available to all the users and servers. The remaining question is whether the index servers should be provided with anonymity?

The index servers are primarily used by the users to map block IDs to locations, but they also communicate with rerouters and storage servers. The users and the storage servers communicate anonymously with the index servers.

The index servers communicate with each other to update the Chord [57] infrastructure. If the index servers are anonymous, there must be a way to map the index server IDs to location information: the index servers must have another (un-anonymous) index service available to support their anonymity. The need for another index service indicates that it is a problem to provide the index servers with anonymity (as it creates a problem corresponding to the problem the index servers solve).

If it was possible to provide index servers with anonymity, this would provide the participating index servers with more protection, but the system as a whole would not be more resistant, as the rerouters would still be un-anonymous.

Index servers in YÅPS shall not be provided with anonymity.

⁵⁴ **Example:** n blocks are necessary to recreate a document, and $2n$ blocks are generated by a publisher, using IDA. If the $2n$ blocks is divided into n pairs, then each storage server used to store a block is responsible for the pair, which the block belongs to. If the other part of the pair disappears, then the block stored can be replicated and the replicas observed, to guarantee a minimum of replicas.

7.4 The Index Servers

The index servers are responsible for the mapping of IDs to locations. Their main function is to map block IDs into block locations. I have chosen to implement the mapping using Chord [57], with a server's ID defining the server's key space.

There are two problems related to the operation of the index servers.

- How are the information on the index servers inserted and updated?
- How do the index servers handle internal replication to avoid key loss if index servers fail?

In the following I shall show how these problems can be solved.

The storage servers are responsible for updating the index servers. Updates requires authentication to prevent adversaries from manipulating the information.

There are two problems related to authentication. First, when the information for an ID is inserted for the first time, how can this information be verified as correct? Second, when information is updated, how can the storage server be authenticated as the one responsible for the update?

The first problem can be solved because it is possible for an index server to verify the provided information. For blocks on storage servers, the index server can request the block anonymously from the storage server to verify it has the block.

The second problem can be solved by authentication of the storage server, using its ID and asymmetric keys. The storage server can transfer its public key (or ID) with the first insertion of new information. When the information is updated, the update is signed by the storage server's private key. The index server can then verify the update using the storage server's key. This scheme can fail, for example if a block is inserted twice or if the storage server loses its private key. The index server can then fall back to the first scheme, but with the modification that it will not accept updates on a block that already are inserted by a storage server, as long as that server still serves the block.

Because Chord maps keys into servers, not values (Section 6.3), a Chord server, which fails, results in a loss of the data in the index server's key space. The storage servers are expected to inform the index servers about all their blocks at regular intervals, this fixes corruption and inserts missing keys, but this is an expensive operation and it should not be done often. It is therefore necessary to protect the keys in Chord against servers that fail. If a server leaves, it can notice its successor and transfer its keys. If a server fails, it will probably first be noticed by its predecessor. The missing keys would now be in the failing server's successor's key space, this suggests that a backup of the keys on a index server should be kept at the successor. If an index server's successor keeps a backup of the index server's keys, it implies that the index server must push updates onto its successor. Index servers joining the Chord ring are also given the responsibility for the replication of the predecessor server. Depending on the stability, it might be necessary to store a backup of more than one predecessor's key space.

Chord should be used recursively, which means that if an index server cannot answer a request it should not return the address of the closest server, but instead forward the request. As the index servers are not anonymous, this strategy reduce the traffic the rerouters must handle, and also reduce the latency of requests (note that Figure 7) shows an iterated lookup).

7.5 Storage Servers

The storage servers can have policies for the usage of the servers (this is different from index servers and rerouters that cannot restrict access), which can include restrictions on the users who can publish or retrieve blocks.

Storage servers are anonymous, but can be contacted using an m-mix (Section 4.5). Users and other storage servers must be able to contact a storage server from its ID. The index servers can be used to map storage server IDs into contact information. As the anonymous storage servers use an m-mix as contact point, the index servers can map a storage server's ID into a storage server ID and a list of m-mixes. The storage servers must keep the list of m-mixes updated (as well as the reply blocks at the m-mixes). The list can either contain direct address information—like an IP address—or the ID of the m-mixes. Indirect addresses provide m-mixes with the ability to update their address, but require users or servers to make an extra lookup on the index servers. The index servers can also be used to map the IDs of blocks into storage servers: the storage server places a list of m-mixes at the index servers, which is inserted under a block ID. YÅPS must support all the listed schemes.

Users who wants to publish a document also needs a way to find a usable set of storage servers. I shall explain how this can be done in the following sections.

Storage servers can provide unlimited or restricted access. I shall use **public servers** to denote the first category, and **private servers** to denote the other category.

Public servers work like the servers in Freenet [11] or GNUnet [4], whereas private servers provide storage like the servers in Free Haven [20] or Eternity [1].

I shall discuss public servers in Section 7.5.1 and return to private servers in Section 7.5.2. The schemes presented in the following sections do not require direct support from the system, and are not bound to any specific protocols. However, YÅPS must support the attachment of arbitrary content to be sent along with requests and answers. This support would allow challenges, responses, certificates, and signatures to be transferred.

Furthermore, I shall discuss the size of blocks in Section 7.5.3, and the lifetime of blocks in Section 7.5.4.

7.5.1 Public Servers

Public servers are open to publishing for everyone. The set of public servers must be announced. A list can be posted to a website, a Usenet newsgroup, in a printed publication, or by using another out-of-band channel. Another solution is to announce these servers at the index servers, for example, this can be

done by hashing the string “PUBLIC STORAGE SERVER” concatenated with a short bit string (for example 16 bits), and insert the location information there. Updates can be authenticated as described in Section 7.4. A storage server can try different strings in order, until it finds an unused string. In a startup phase the strings can be tried in order, but in a working system the starting point for an insertion or search for a storage server should be chosen at random to spread the load of the servers. Such a scheme is supported by YÅPS, as it is just a special case of location.

Users or adversaries can fill public servers up with garbage. This problem is present in current open systems like Freenet [11] and GUNet [4]. A partial solution to this problem is to issue a challenge, users must solve to publish a block. Challenges work by letting users *pay* with computation resources for each block they wish to publish. Challenges can be implemented as hash-based challenges.⁵⁵ This scheme requires users and adversaries to provide computation resources to be able to use the servers. The scheme increases the cost of attacks, as an attacker must provide the necessary computation resources.

Schemes for challenges can be implemented on the top of YÅPS.

7.5.2 Private Servers

Private servers can have policies based on contracts or trust relations. In the following I shall discuss how trust can be used to restrict storage server access.

The problem is to restrict the access to the storage servers when both users and servers are anonymous. The problem can be solved by providing the users with a capability based on trust relations.

I believe the best way to implement trust is to use a **web of trust** [26, Chapter 1] strategy. Web of trust is an alternative trust model introduced by Phil Zimmermann for use in the email encryption tool *Pretty Good Privacy* (PGP). The traditional trust model for Internet systems is the hierarchical **public key infrastructure (PKI)**, where central certificate authorities (CAs) must be trusted. For systems like APSs it is unsound to place trust on any centralized authority.⁵⁶ Web of trust is a decentralized alternative based on direct trust. The infrastructure is supported only by the participants, and no participant is given any special roles (like the CA in PKI). In this model a participant knows the public key of a number of other participants, who have been verified directly. Furthermore, a subset of these other participants are trusted (in varying degree) to certify the public keys of other participants. This network can be used to establish paths of trust to any participant who can be reached in the network. The hierarchical model can be emulated in a *web of trust* model, in effect participants choose their own CAs among other participants. That CA-like structures can be emulated makes the model flexible and also usable in larger organizations. Web of trust is analyzed and discussed in [59].

⁵⁵ An example of such a challenge is to give a string s to a user and demand another string S , which given as input to a secure hash function would produce a string that contains s as a substring. See *hashcash* for more details of such a scheme, <http://www.hashcash.org> (November 2003).

⁵⁶ To get CAs to work right in the first place is difficult in the first place [25].

But how does this work with anonymity? Traditional certificates with identification cannot be used, these must be replaced with certificates representing pseudonyms. I shall illustrate how these can be used with an example:

Example: If I run a storage server and want to provide space for dissidents of Ruritania, I can anonymously inform Amnesty International about this. Amnesty International can use their contact network to put trust on certain Ruritanian persons and inform them about my service. Because I trust Amnesty International, I am able to verify dissidents of Ruritania if they try to publish something using my server. In practice I get a certificate from Amnesty International with their public key, and Amnesty International uses the corresponding private key to sign certificates of the dissidents. When the dissidents contact my server they must sign their communication with their private key, and refer to their certificate. I can then verify that I communicate with them using the public key of Amnesty International to verify their certificates, and the public key from their certificates to verify the communication.

The problem with this approach is that I need at least to store the certificate from Amnesty International to be able to verify messages from the dissidents (assuming they send their certificate as part of the communication, otherwise I would also need that). The certificate can put me in trouble if the secret Ruritanian police visits me and get access to my server and finds the certificate of Amnesty International. The secret police must be able to recognize this certificate and if Amnesty International created the certificate only for use by me and the dissidents, this could be difficult. Furthermore, anonymity is the first line of defense and strong denial can be used as defense, but the existence of the Amnesty International certificate might in itself be enough to get me into trouble, and the adversary might put my server under surveillance (without my knowledge) and wait for a communication with the Ruritanian dissidents and watch me use the certificates.

These problems are minor, but there is an easy solution. I can use a trusted proxy in a less restrictive society. Instead of sending anything directly to me, requiring me to verify it, it can be sent to a proxy for approval. The proxy needs to have the certificates, and can verify the published material to belong to the dissidents (but the proxy is placed in a society outside the secret polices hands). The proxy can then forward the material to my server. The proxy prevents the secret police from finding anything on my server to link me to the dissidents, but the adversary would be able to tell that I store something from the proxy.

YÅPS must support authentication needed to implement the presented strategies based on web of trust. The use of proxies is not something YÅPS need to support, as it is easy to implement on the top of YÅPS.

7.5.3 Block Size

The storage servers are used to store blocks. The size of these blocks must be decided. The choice of block size has effects on resource usage and security. Different block sizes requires handling of the block size, and complicates secret sharing. I have decided to provide only one block size to simplify secret sharing and block handling.

Small blocks requires more requests when a document are to be recreated (as it consists of more blocks), but minimizes internal fragmentation on the storage servers. Large blocks requires fewer requests, but introduces wasted space at the storage servers. Half the last block of a document can be expected to be wasted at the storage servers and it also wastes network traffic as the full block needs to be communicated to the user retrieving the document. The waste of bandwidth caused by an increase in requests is however expected to dominate the network overhead for the last half block.

As storage is currently cheaper than bandwidth, I believe that blocks must be rather large. I suggest a block size of 100kB for YÅPS.

7.5.4 Lifetime of Blocks

Storage servers can have their own storage policies on the lifetime of blocks. For example, they can store them as long as the server is active, remove the least requested blocks, the oldest blocks, or blocks stored for entities without a contract. The different lifetimes is a problem because existing blocks might be used for secret sharing. One solution is to ignore this problem and require the users to replicate the blocks used for secret sharing. Another solution is to let YÅPS store lifetime information with blocks, and use this when blocks are served to secret sharing. When a block is stored, and the publisher has an idea of the lifetime of the block, this must be communicated with the block. When a publisher requests a block to use for secret sharing, she should also send an estimate along with the request, and the storage server can then try to match this lifetime with a block with the same lifetime.

I propose to store the lifetime as an expire date, for example saved as a year and a day offset (the date would require less than 32 bits).

7.6 Rerouting Servers

The rerouting servers are implemented as a variant of Mixminion [15] mixes. The mixes provide anonymous communication. The mixes decrypt incoming packets, keep the packets in a pool, and encrypt and resend packets. Users and servers must be able to retrieve information about the mixes, before they can use the mixes. I shall discuss this problem of spreading information about the mixes in the following. In the next section I shall discuss availability.

If all the mixes keeps a current list of a set of mixes, users and servers can learn about the set of rerouters by retrieving such a list from any rerouter. The list must contain the public keys of the mixes, as well as contact information such as an IP address or domain name. Each mix can create a certificate by signing a string containing their ID, public key, address, and a date interval where the certificate is valid. If the information is changed a new certificate can be issued. The next problem is to distribute the certificates to all the mixes. For small sets of mixes, the certificates and updates can be spread by a flooding strategy: each mix sends updates to all the mixes it knows and they resend the updates to the rerouters they know (unless the mix already have seen the update, then it should be ignored), and so on. I speculate that flooding will work well— even for thousands of mixes, however, another solution is required for huge

rerouting networks. A possible solution is to arrange the mixes in a ring like Chord (Section 6.3), based on their IDs. Mixes can then send updates to their successor, and ignore when the update gets back. This update scheme ensures a small amount of traffic.⁵⁷

The mixes use key rotation, and each time they make a new key, the mixes creates and distributes a new certificate. Users can get a current list of mixes from an arbitrary mix, but this requires that users knows at least one mix. Out-of-band channels is be used to let users learn about a small set of mixes. The index servers can also be used as described in Section 7.5.1 for public storage servers.

To provide encryption between mixes, forwarded packets must be encrypted. This can either be done using the next mix' public key or a session key can be used to save computation time (the session key can be transferred using the public key and cached and reused). This scheme also authenticates the next mix, as the next mix' private key is needed for decryption. YÅPS must support both schemes.

7.6.1 Availability

The rerouting network creates an availability problem. The path of rerouters used in rerouting requires all the rerouters to function properly, otherwise a packet cannot reach its destination. The problem is not critical for paths created by users or servers for immediate use, as a new path can be used if the first fails. A more serious problem is reply blocks with a long lifetime, as a recipient will be unreachable if a reply block includes a rerouter that does not work. A simple solution is to use a set of reply blocks, which use different paths, instead of a single reply block.

In the design I propose in Section 4.5, a set of reply blocks is placed at the m-mixes chosen by a recipient. The m-mix then chooses a reply block at random from this set when it forwards messages to the recipient. When a sender sends a message to an anonymous server, the reply block chosen at the m-mix can contain a mix that does not forward the message. The sender will not be informed of this and must wait for an answer and eventually resend the message. The user can resend the message by another m-mix, but even if the same m-mix was chosen, the randomization at the m-mix would probably result in another reply block. This solution is not ideal, because it can result in a huge latency, and it is impossible to contact an anonymous server if one mix from each reply block at the m-mixes, are not working. The rerouters must be regarded as one of the most vulnerable parts of the system. Tarzan provides a secure solution, which is less prone to these problems, but it is expensive in resources for all participants (Section 4.3.1, p. 31), and expose users.

⁵⁷ If the updates rate is too slow, it can be made faster: a mix sends updates both to its successor and to $\lg n$ random mixes (for n mixes). An update seen before by a mix should be ignored and updates should contain a time to live (for example, initialized to $\lg n$). The time to live should be decreased by a mix before the update is forwarded. When a rerouter decreases the time to live to 0, then the request should be forwarded only to the rerouter's successor. This optimization based on randomizing, reduces the time from $O(n)$ to an expected $O(\lg n)$, but with an increase in the number of messages from $O(n)$ to $O(n \lg n)$.

I have proposed a length of 5 to 10 rerouters (in Section 4.3.2) and expect rerouters to have a high availability (Section 3.2), but I also require the design to work if the availability is reduced to 50% on average. At 50%, even a path of 5 rerouters will work only on average $0.5^5 = 3\%$ of the time. There is no easy solution to this problem, but it suggests that rerouters must be operated on stable machines, and also that it is necessary to keep information about availability. Each rerouter can keep a *hit list* with the rerouters it has communicated successfully with, this information can be kept with the certificates. The list can be updated regularly by sending special pings to the rerouters. The pings can be simple packets that are sent back to the sender, using a small path. If the packet returns, then it can be assumed that all the rerouters on the path work. The hit list can be spread out to other rerouters, and provided to users when they request a list of current rerouters. This system should weed out rerouters with an unacceptable availability. I believe the practical deployment of anonymous remailers proves that it is possible to provide a mix network that is stable enough to use in a system like YÅPS.

The availability of YÅPS depends on the availability of the rerouters, which suggests that only servers with an expected high availability should be used as rerouters and that a system is needed to help users and servers not to use rerouters with low availability.

7.7 Failing Servers

Servers are expected to fail. In the following I shall briefly mention what happens when the different servers fail. I shall use requester to denote both servers and users.

- *Index servers*: If an index server fail, a requester can try another ID number. A requester receives an error message (unless it is the index server she sends the request to). The requester have to wait as long for an error message as she would for an answer.
- *Storage servers*: If a storage server fails, a requester must try another one. A requester will not be noticed about an error, but should try another server after an amount of time. A requester have to wait at least as long for the message as she would for an answer.
- *Rerouters*: If a rerouter fails, a requester must try another path. A requester will not be noticed about an error, but should try another path after an amount of time. A requester have to wait at least as long for the message as she would for an answer.

As can be seen, a requester cannot know the difference between failing storage servers and rerouters, which suggests that a requester should always try another path and another storage server. Rerouters are assumed to be relatively stable, and requesters should always use an updated list of rerouters.

7.8 Updateable Document

In the following I shall describe how updateable documents can be integrated into YÄPS. There are several problems with updateable documents. First, it requires a way to authenticate the publisher of a document. Second, how is the document integrity protected? Third, how can it be implemented to provide redundancy?

I have proposed a system where documents are stored in blocks. The blocks are located from an ID derived from the content of the block. This scheme is impossible to extend to support documents that can be modified (because updates would change the ID and make it impossible to locate the block).

I shall start by looking at the problem of providing updateable blocks and then show how updateable blocks can be used to implement updateable documents.

7.8.1 Updateable Blocks

Updateable blocks can be implemented using asymmetric cryptography (this is done in Freenet [11]): a publisher creates an asymmetric key pair for each updateable block and a secure hash of the public key is used as the ID of the block (instead of the ID scheme that use a secure hash of the content). If a digital signature made with the corresponding private key is included in the block, then users retrieving the block is able to verify the integrity using the public key (the public key must be distributed with recipes for the document). The storage server can authenticate updates, as the new blocks signature can be verified by the storage server using the public key of the publisher (this must be sent along with the first version of the block). In a protocol for updates all the data must be integrity protected. The integrity protection can be done by signing a secure hash of all the traffic sent from a user to the storage server (a similar approach is used in TLS [19]). I shall use **static block** to refer to normal blocks that are not updateable.

Multiple IDs for a block can be created with the same scheme as constant blocks, as explained in Section 7.1.2. The use of a new asymmetric key pair for each updateable block results in unique IDs for each block and furthermore makes it impossible for adversaries to use the digital signatures to show that a group of updateable blocks are related.

Blocks are fixed sized, this includes updateable blocks, it is therefore not possible to increase the size of updateable blocks.

Updateable blocks must be the same size as other blocks to make it easy to manage the blocks. A digital signature uses 256B to support a 2048-bit key. I would also recommend that a counter is saved with an updateable block to ensure that old updates never overwrite newer blocks. Updated blocks are sent with a new counter, and the update is only accepted if the new counter is greater than the old counter. The counter must be able to support all the needed updates and support initialization with a random value to make it hard to distinguish updateable blocks from static blocks. If 5 bytes are reserved for the counter, 2^{39} updates are allowed on average. The signature and counter requires less than 300B in total.

Updateable blocks should be secret shared like static blocks. The secret sharing must be done before the signing to make it possible to verify the integrity of the resulting block. The exclusive-OR scheme makes this easy.

Redundancy for updateable blocks can be provided by replication. This replication results in blocks, which are stored on multiple servers. When users update a block, they need to update all the existing replicas. During the update process more than one version of the block exists at the same time.

Updateable blocks must be marked to ensure that they are not used for secret sharing with other blocks. This information is available from the ID and block: if a secure hash of the block does not result in the ID, then the block is an updateable block. The information is also indirectly available at a storage server, because it needs to store a public key corresponding to the block. A storage server should use this information, to avoid returning updateable blocks as a response to a request of a random share for secret sharing.

The blocks are integrity protected by the publishers signature and can therefore be transferred between storage servers without a risk of changes by malicious storage servers.

7.8.2 Updateable Documents

Updateable blocks can be used to implement updateable documents: updateable blocks can simply contain a recipe. New blocks can be created to reflect changes in a document and the recipe in the updateable block changed accordingly.

If a document is too large to be described by a recipe that can be stored in a single block, an updateable block can be a recipe for a document containing another recipe, this is discussed further in Section 7.9.

This proposal for documents that can be updated meets the goals of YÅPS: it provides the same security on integrity as the storage of normal blocks (but is more complicated).

7.8.3 Security Issues

Even though secret sharing makes it possible for anyone to relate an updateable block to any document, the scheme makes a connection between the publisher and the storage server. The connection is defined by the private key of the publisher, as only the publisher possess this key. A recipe from a publisher can be used to show what document an updateable block belongs to. A storage-server operator can still deny knowledge of the content, which the updateable block can be used to create, but the defense is weaker than for static blocks, because updateable blocks can be shown to belong to a certain block with high probability.

I conclude that updateable blocks weakens the strong denial defense, but I speculate that the small difference will not change the outcome of a legal case. The reason is that the defense is based on the fact that an operator cannot know what she stores on her server (and this would also be the case for updateable blocks).

7.9 Recipes

Recipes represents the information a user needs to recreate a document. Recipes are required to store the IDs (or public keys for updateable blocks) of a set of blocks as well as the size of the original document. Furthermore recipes must contain other information needed to recreate a document, for example information about the replication scheme.

I shall discuss alternative recipes in Section 7.9.1 and the security problems related to the possession of a recipe in Section 7.9.2.

7.9.1 Alternative Recipes

Users are not limited to use a specific format for recipes, they can define their own format. A standard is however needed for documents that are published for a large set of users who cannot be expected to have a special variant of the client program.

I have presented recipes as a list of indirect references to storage servers, using the index servers as an indirection. The level of indirection makes it possible for storage servers to update the location information and keep recipes valid.

If blocks are stored on stable servers and expected to be used within short time (the location information will be unusable after a key rotation), it can be usable also to save location information for the storage server rather than just the ID of a block. The location information makes it possible a users to retrieve a block directly, which saves the user from having to use the index servers. If the location information does not work, a user can still find the updated locations using the index servers.

7.9.2 Short Recipes

Recipes contain information to recreate a document. The possession of a recipe will probably be considered illegal if the recipe can be used to create an illegal document. In the United States, linking to sites, which distributes copyright protected music illegally, is itself illegal,⁵⁸ but the *Ogle* dvd-player software for Debian Linux,⁵⁹ is distributed without the DeCSS⁶⁰ library, because this library is illegal (for example in the United States). Instead the installation package includes a program that downloads the DeCSS library and compiles it. This scheme has not caused legal trouble. I conclude that the legal status of recipes is unclear. However, certainly there *are* nations where some recipes, would be illegal.

In areas where it is illegal to possess a recipe it is important that recipes are easy to hide. Fortunately, all the data in the recipe with the exception of the

⁵⁸ *MP3 Site Sues RIAA Over Linking* in Wired Magazine, <http://www.wired.com/news/print/0,1294,36778,00.html> (November 2003).

⁵⁹ *ogle 0.8.2-11*, DVD player with support for DVD menus, <http://packages.debian.org/stable/graphics/ogle.html>, (October 2003.)

⁶⁰ *The Openlaw DVD/DeCSS Forum Frequently Asked Questions (FAQ) List*, Harvard Lawschool, <http://cyber.law.harvard.edu/openlaw/DVD/dvd-discuss-faq.html> (November 2003).

document size ⁶¹ is random looking (it can be considered pseudo random, as it is created by secure hashing), making recipes hard to recognize. If an adversary suspects that something is a recipe it is however trivial to verify by trying to recreate the document.

To reduce the problem of hiding recipes, recipes can be shortened: recipes can be published as blocks. This requires that a recipe is smaller than the size of a block. If a block is 100kB, it can contain $100\text{kB}/20\text{B} = 5000$ IDs, corresponding to documents up to the size of $5000/2 \cdot 100\text{kB} = 250\text{MB}$ (not including space for the documents size or public keys for updateable blocks). For documents, which use more space, the block can instead contain a recipe for another recipe (of up to 250MB and this scheme can be repeated to support any document size). This scheme can reduce the information needed to recreate a document to a single ID of 20B (or two if the blocks is secret shared). This amount of data can be handled as a handwritten note (for example in hex, resulting in 40 or 80 digits and letters) and is small enough to hide with steganography and communicate by covert channels [40].

This shortening scheme can be implemented as part of the client program it does not affect the servers.

7.10 Summary

In these sections, I have discussed the open problems related to anonymous communication, secret sharing, and location, as well as the remaining issues related to the goals.

The following is a summary of the discussion:

- *ID*: Blocks and servers are issued an ID either based on content or an asymmetric key pair. The server IDs support authentication.
- *Redundancy*: Publishers are responsible for redundancy of publications. Both replication and IDA is supported for document content on storage servers whereas location information is replicated multiple places on the index servers. The location information is also internally replicated by the index servers.
- *Index servers*: The index servers are also used to announce public storage servers. The Index servers are not provided with anonymity.
- *Rerouters*: There are availability problems with the rerouting network, which requires the rerouters to provide high availability.
- *Updateable documents*: Updateable documents are supported by the use of updateable blocks containing recipes.
- *Recipes*: Recipes can be shortened by storing them in the system as blocks.

In the next section I shall give an overview of YÅPS based on the decisions in previous sections.

⁶¹ The document size can be dropped if padding can be made in the document to match a multiple of blocks.

8 Yet Another Anonymous Publication System

It's wonderful to see those who turned technology against free expression for so long now scrambling to catch up with those setting information free.
—Alan Brown, Red Rover [6]

In the following I shall give an overview of YÅPS and discuss the security and the requirements of users and servers.

I shall begin with an overview of YÅPS in Section 8.1. The overview is followed by a discussion of security in Section 8.2 and an overview of requirements in Section 8.3. I shall then discuss performance in Section 8.3 and the prototype I have implemented in Section 8.5. A summary is provided in 8.6.

8.1 Overview

In the following I shall give an overview of how documents are published and retrieved, and what operations the different servers are providing.

8.1.1 Publication

When a user publish a document the publication results in a recipe, which can be used to recreate the document. Part of the publication process is to find a suitable set of storage servers. There are two types of storage servers in YÅPS: public and private servers (Section 7.5). Public servers are open to anyone and the IDs of public servers can be announced by out-of-band channels, for example web pages or Usenet, but the index servers can also be used to announce such servers (Section 7.5.1). Private servers can have arbitrary restrictions on their use and are announced using out-of-band channels. I expect that lists of storage servers are circulated for both kinds of servers.⁶² The users have to use the index servers to map the ID of a storage server into the address of an m-mix. Messages to a storage server can be send by means of the m-mix.

Users also needs to know about the rerouters to be able to communicate with the storage servers. Users can learn about the rerouting network by retrieving a list of certificates from an arbitrary rerouter.

When a publisher wants to publish a document D and have found a suitable set of storage servers, she must:

- Divide D into the set s_a of n blocks. Store the document size in the recipe.
- If necessary: pad the last block with 0 s.
- Generate or retrieve a set s_b , of n random blocks.
- Create a set s_c of n blocks by secret sharing of the sets s_a and s_b .

⁶² Such lists consist of the storage servers' IDs and look random. The lists are therefore difficult to identify by an adversary.

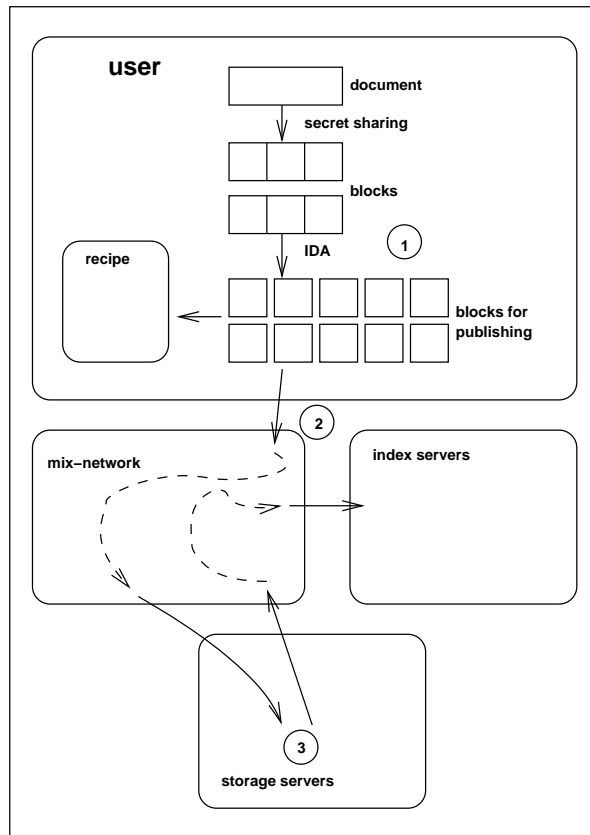


Figure 8 Publishing in YAPS: (1) the user creates a set of blocks for publishing and a recipe, (2) the user publish the blocks on a number of storage servers, and (3) the storage servers publish location information on the blocks on the index servers.

- *If IDA is used for redundancy:*
Create two new sets of blocks S_b from s_b and S_c from s_c . Store the IDs of S_b and S_c in the recipe and publish the blocks on a set of storage servers.
- *Otherwise:*
Store the IDs of s_b and s_c in the recipe and publish the blocks on the set of storage servers. The blocks should be replicated to provide redundancy.

When a block is stored at a storage server, the storage server returns a status of the storage request. The storage servers are responsible for updating the index servers if a block is added to the storage server's storage. The publication process is shown in Figure 8. A user can generate a reply block (SURB) and send this with the first message to a server to be anonymous.

8.1.2 Retrieval

To be able to retrieve a document, a user must first obtain a recipe and know an index server. A recipe contains a list of block IDs, a user can use the index servers to map the block IDs into a list of m-mixes, which can be used to contact the storage servers.

The user must:

- *If IDA was used for redundancy:*
Lookup the locations of a set of blocks necessary to recreate s_b and s_c .
Retrieve the blocks (repeat if some of the blocks cannot be retrieved).
Recreate the sets s_b and s_c , use secret sharing to create s_a .
- *Otherwise:*
Lookup the locations of the blocks and retrieve the blocks (repeat with other ID numbers if it fails) to get s_b and s_c . Use s_b and s_c to create s_a with secret sharing.
- Concatenate the blocks and remove padding (if necessary).

The retrieval process is shown in Figure 9. A user can generate a reply block (SURB) and send this with the first message to a server to be anonymous.

8.1.3 The Index Servers

An index server provide two operations:

- *Lookup:* A request contains an ID of a block. If the index server is responsible for the ID it is either returned or a message indicating it does not exist is returned. If the index server is not responsible for the ID then the server forwards the request to another index server.
- *Insert:* Works like a lookup to find the right server. A request contains an ID, a string and a digital signature (the request also contains a public key if the insert is not an update of already inserted information). The $(ID, string)$ -pair is inserted, if the insert is accepted.

Both operations relies on the Chord design [57] and the insert functionality requires procedures for authentication. (Section 7.4). The index servers replicates data internally (Section 7.4) to provide failure resistance.

When a storage server inserts the information for a block the string contains the server's ID and the ID of an m-mix.

8.1.4 The Storage Servers

A storage server stores blocks for users and is anonymous. The storage servers must keep the index servers updated with information about their m-mixes and the blocks they store. A storage server provides the following operations:

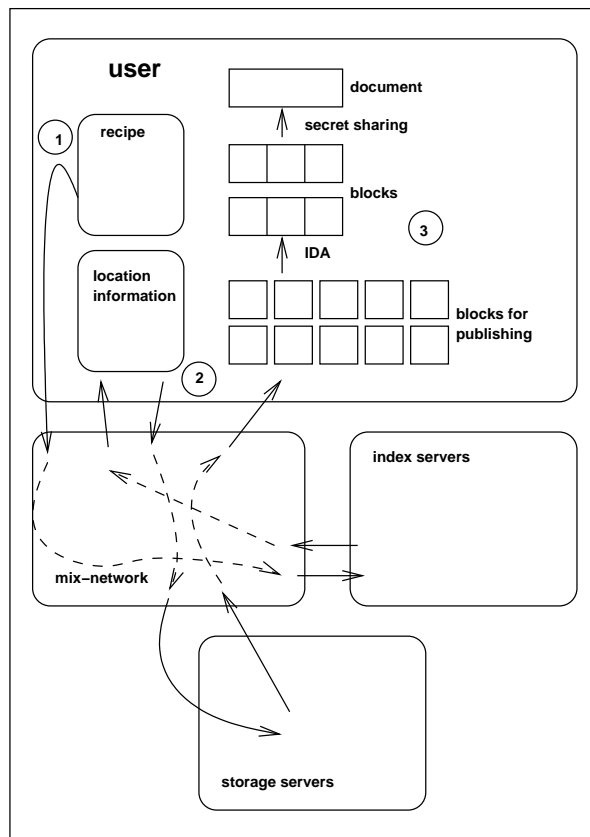


Figure 9 Retrieval in Y&A;PS: (1) the user retrieves location information for the block IDs in the recipe from the index servers, (2) the user retrieves the blocks, and (3) the user recreates the document from the blocks.

- *Retrieve block*: A request for a block includes an ID. If possible the storage server returns the block corresponding to the ID. Otherwise an error is returned.
- *Retrieve random block*: A random block is returned by the storage server. The request can contain a lifetime, if this is the case, the lifetime is used to find a block with a matching lifetime.
- *Store block*: A request for the storage of a block. If the request is accepted, the storage server stores the block and updates the index servers—otherwise an error is returned.
- *Update block*: A request for the update of a block. A storage server must verify the updated block before changing the block (see 7.8). An update can also be used to remove a block: if the replacement is an empty block, the block is removed.

A storage server must save meta data belonging to a block. For example, a public key for updateable blocks, the ID number of a block ID, or a lifetime for a block.

8.1.5 The Rerouting Servers

A rerouting server receives and resends packets using the information in the packets. A rerouting server supports two operations:

- *Resend packet*: Receive a packet and resend it.
- *Retrieve list*: A request for a list of known rerouters. The rerouter returns its list over known rerouters.

A rerouter must also be able to support the m-mix role, which requires the rerouter to keep a list of reply blocks for a set of servers and use these to forward packets to anonymous recipients. Furthermore the rerouters must keep a list of reliable rerouters (Section 8.1.5).

As I shall describe later (in Section 8.2.1), the rerouters can also function as proxies for some users.

The rerouters are implemented as a variant (Section 4.5) of Mixminion [15].

8.2 Security

In the following I shall discuss the security provided by YÅPS. I shall describe the realistic attacks against YÅPS, and how this affects users and server operators. In Section 8.2.1 I shall discuss the use under an oppressive regime.

Users and storage servers are protected with anonymity. Anonymity prevents direct attacks because an adversary has to break the anonymity in order to mount direct attacks.

YÅPS does not provide protection against edge attacks as it requires a large amount of resources and introduces other attacks. I consider the edges to be the most vulnerable part of YÅPS. Edge attacks can be mounted by eavesdropping on the network connections of suspects or by adding controlled rerouters to the system. If an adversary needs to identify a specific user or storage server this is hard for a large rerouting network, because it requires the breaking of the rerouting network. Attacks to identify storage servers are considered easier than attacks to identify users, because storage servers have a fixed ID. The fixed ID makes it possible to send large amount of traffic to the storage server and this can make edge attacks easier.

The solution I have proposed is expected to be resistant to attacks by strong adversaries, as described in the threat model (Section 3.1), with the exceptions I have just mentioned.

Index servers are used to map IDs into locations. An index-server operator cannot know the relation between IDs and documents, as the IDs on a specific index server are mapped into random blocks. The missing knowledge

provides the index-server operator with strong denial. Rerouters process traffic like other routers on the Internet, but they cannot know about the content they process. Both index servers and rerouters are advertised in public, which expose these servers to direct attacks. This exposure implies that the rerouters and index servers not should be operated from places where it is illegal, and that the these servers must be protected from attacks by adversaries. Furthermore, a denial-of-service attack can be directed against index and rerouting servers. The easiest way for an adversary to censor YÅPS is to block access by mounting a denial-of-service attack against the rerouters, or block potential users from the access to the rerouters.⁶³ I expect the legal status of both rerouters and index servers in YÅPS to correspond to the status of existing remailers. Mixmaster remailers are currently operated in Austria, Canada, France, Germany, Italy, Poland, Spain, and the United States.⁶⁴

Legal attacks can be mounted against users (if they can be identified), because users will most likely possess whole documents, whereas storage-server operators can deny that they store parts of any specific document if they are identified and face prosecution. The denial defense is slightly weaker for storage-server operators that store updateable blocks (discussed in Section 7.8.3).

The use of *denial* as a defense against legal attacks is unclear as of this writing. In relation to ISPs and defamation, there have been some cases in the United States,⁶⁵ and it seems that in these cases missing knowledge removes responsibility.

8.2.1 Oppressive Regimes

In an oppressive regime with tight control over the communication infrastructure, it is hard to operate any kind of underground communication network. The regime is assumed to control the network connections and the use of a system such as YÅPS for any purpose would probably be illegal.

Rerouters and index servers are exposed, consequently these servers should not be operated under an oppressive regime. Storage servers and users are better protected with anonymity (the denial aspect for storage servers cannot be expected to make a difference). The anonymity is not provided before communication enters the rerouting network, so the communication with the first router must be protected from the adversary. This protection can be provided implicitly if the adversary has limited control of the network connections. Covert channels [40] and steganography can be used in that case. Hiding of the communication requires support from the rerouters. I believe it could be interesting to research the use of CAPTCHAs⁶⁶ [60], and the use of emails that looks like spam, to hide communication.

⁶³ The last option can be made harder by providing proxies as described in Section 8.2.1.

⁶⁴ Estimate based on a research of the domain names of the mixmaster remailer list, November 26 2003, <http://anon.efga.org/Remailers/TypeIIList/pubring.mix> (November 2003).

⁶⁵ The Florida Bar Computer Law Committee Online Journal, *Online Service Provider Liability for Defamation*, http://www.computer.flabar.org/2_spring96/online.html (November 2003).

⁶⁶ CAPTCHAs are tests that computers cannot pass, but humans can pass, for example words encoded in pictures. The CAPTCHA Project, <http://www.captcha.net/> (November 2003).

If adversaries know a set of rerouters, then they can block access to these. A large set of proxies, which just forward and receive traffic to and from users, can be placed outside the oppressive regime to make such attacks, which use blocking, harder.

8.2.2 Conclusion

If an adversary can obtain access to the computers of users then the adversary might be able find illegal documents or recipes. If an adversary can get access to a storage server, the adversary can show that the storage server can store blocks from any specific document, but the operator would be able to deny this. Both users and storage servers are however protected by strong anonymity.

Index servers and rerouters are not anonymous, but they do not store documents. These servers can however easily be attacked if they are operated in an area where it is illegal to operate such servers or by denial-of-service attacks. Furthermore, rerouters present a target for adversaries that are attacking anonymous users or storage servers indirectly.

I conclude that under oppressive regimes with tight control over the communication infrastructure it is impossible to operate YÅPS and similar APSs in a safe way. Some servers must be exposed, and anonymity cannot be guaranteed on the network connections in the oppressive regime (because the rerouters placed there can easily be controlled or blocked by the regime). The core problem for an APS under an oppressive regime is to hide the activities of the system.

8.3 Requirements

In the following I shall discuss the practical requirements related to the use of YÅPS, as well as the requirements of the servers.

8.3.1 Users

Users who want to publish a document needs a computer with Internet access, a client program, and a list of usable rerouters and storage servers.

If a user wants to publish a document of size n blocks, then the user must first use the document to create a set of blocks. The creation includes retrieval or construction of another n blocks for secret sharing. To support redundancy, a user can either replicate the n or $2n$ blocks a number of times or use IDA to create a new set of blocks. If IDA is used and the document must be available even if half the storage servers used fail, then $4n$ blocks must be published.

Users who want to retrieve a document needs a computer with Internet access, a client program, a recipe, and a list of usable rerouters and index servers. A user must retrieve $2n$ blocks to recreate a document of size n blocks.

The requirements for users regarding space, bandwidth, and computation are modest.

Users can be anonymous when they use YÅPS if the network connection to the first rerouter is not observed by an adversary.

8.3.2 Storage Servers

A storage server primarily needs space to store blocks and bandwidth to serve blocks to users. Bandwidth is also needed to update index servers.

The availability of storage servers should be high to support efficient retrieval, but redundancy ensures availability of a document even if some servers fail. The redundancy factor used in the storage process can be adjusted pragmatically to work with the availability of the storage servers.

Storage servers can be anonymous if the network connection to the first rerouter is not observed by an adversary.

8.3.3 Index Servers

Index servers need space to store location information for a number of keys as well as a backup of a set of neighbor's location information. About 256B are needed to store location information for an ID⁶⁷ and this is expected to be replicated at least one time at another index server. I believe that these storage requirements implies that more than 1000 IDs with location information can be stored pr. MB provided (including one replica). Space is also needed to support the Chord infrastructure, this is $O(\lg s)$ for s index servers.⁶⁸ The overall space requirements are small.

The function of the index servers is to map IDs into locations. The requests are sent from anonymous servers and users. The network traffic generated by the requests is small. An index server will receive a request, and probably forward it to another index server, an answer will be returned, that must be sent back to the requester. Furthermore, traffic is needed to get updates from the other servers. The bandwidth requirements of index servers are small compared to storage servers and rerouters.

The availability of index servers should be high to support efficient responses but the redundancy factor used to replicate IDs and location information can be adjusted pragmatically to work with the availability of the index servers.

Index servers are not protected against direct attacks. For example, an adversary can attack index servers by legal means or using denial of service attacks.

8.3.4 Rerouting Servers

The primary requirements for rerouters are bandwidth and high availability and the rerouters also need an amount of space to keep a pool of messages, and a cache of already seen packets (I expect the total space requirements to be less than 200MB).

The requirement for bandwidth depends on the number of packets routed through the rerouting network but the bandwidth proposed rerouting solution

⁶⁷ For blocks, the index server must keep an ID (20B) and a value, which contains: (1) a storage server pseudonym (20B), (2) a public key (192B, this is enough to store a 1500 bit public key.), and (3) rerouter ID (20B).

⁶⁸ I speculate this is less than 10KB for each index server, resulting in a requirement of $\lg s \cdot 10KB$ at each server.

scales well with the number of rerouters. As YÅPS is a storage system and all blocks published or retrieved by users are communicated using the rerouting network, there will be a large amount of traffic. Furthermore, rerouters needs to communicate with other rerouters about changes in the rerouting network, this communication also requires bandwidth.

The availability of rerouters is essential (as I have discussed in Section 8.1.5). In a practical system the rerouters should be provided by operators, who can provide high bandwidth and high availability of servers

Rerouters can also provide a proxy service for users (as I have described in Section 8.2.1) this requires extra software, but not any special resources from the rerouters.

Rerouters are not protected against direct attacks, for example attacks by adversaries based on legal means or denial-of-service attacks. Rerouters are central for the security in YÅPS and an obvious target for adversaries who wants to identify users or storage servers or block anonymous access to the system.

8.4 Performance

I shall now given an estimate of the performance of YÅPS from a users perspective.

The efficiency of the index servers is based on the number of index servers. I assume the number of index servers is N , lookups will then take $O(\lg N)$ time. I assume a payload of size 0.2 times the size of a block (see Appendix B.2).

In the analysis I have ignored servers that fail, as failing servers does not change the time complexities. However, in practice failing servers have a large impact on performance—specially failing rerouters on a path used by a publisher. I shall discuss failing servers in Section 8.4.3.

I shall begin with an analysis of the publication of documents in Section 8.4.1 and then continue with retrieval of documents in Section 8.4.2. I discuss failing servers in Section 8.4.3 and concludes in Section 8.4.4

8.4.1 Publication

A user who wishes to publish a document of n blocks must first create a set of blocks for publication, using secret sharing and a redundancy scheme (for replication this could include the retrieval of blocks for secret sharing). The creation of the set of blocks for publication requires $O(n)$ time and space and results in a set of $O(n)$ blocks (at least $2n$ blocks because of secret sharing—and $4n$ blocks is expected to provide redundancy).

The publication requires the user to find a storage server for each block in the set. I expect users will do this using out-of-band channels, which should result in a list of IDs. The user can then use the IDs to lookup location information on the storage servers. The lookup is done using the rerouting network, which introduces a latency on communication, dependent on the number of rerouters on a path and the number of failing rerouters. The latency will be experienced as a constant, c_l , and it is expected to be dependent of the available bandwidth

in the rerouting network. Each lookup requires $O(\lg N)$ time to be fulfilled by the index servers. As N is constant for the publication of a document, a lookup takes a constant amount of time c_i . The result is a constant amount of time $O(c_i \cdot c_l)$, for the lookup of the storage a server.

The publisher then have to publish the blocks on the storage servers. The publishing is also done using the rerouting network, but the transfer requires more traffic than requests. As the blocks fits into the payload of 5 packets, I expect this to take around $O(4n \cdot 5c_l)$ time. However, as more than one block can be published at a time, this reduces the time in practice.

In short it requires $O(n) + O(4n \cdot c_l \cdot c_i) + O(4n \cdot 5c_l) = O(n)$ time to publish a document. As both lookups and publication operation can be done at the same time, the time is expected to be dominated by the latency in the rerouting network for small documents: $2c_l$. The document is expected to use $4n$ space on the storage servers.

Before the document can be retrieved using the recipe, the storage servers must update the location information on the blocks on the index servers.

8.4.2 Retrieval

In the following I shall ignore details explained under publication.

A user who wants to retrieve a document of n blocks must retrieve at least $2n$ blocks. To retrieve the blocks, the user must first locate the storage servers, which requires $O(2n \cdot c_l \cdot c_i)$ time. The user can then retrieve the blocks, using $O(4n \cdot 5c_l)$. The recreation of the document requires $O(n)$ time and space.

In short it requires $O(2nc_l \cdot c_i) + O(4n \cdot 5c_l) + O(n) = O(n)$ time to retrieve a document. Again the latency $2c_l$ caused by the rerouting network is expected to dominate the retrieval of small documents.

Example: I shall illustrate the retrieval of the document *The New Testament of the Bible*, which requires about 1032kB compressed.⁶⁹ I assume a block size of 100 kB, which implies that the document requires 22 blocks to be recreated. First, I have to locate the 22 storage servers, this requires $O(2n \cdot c_l \cdot c_i)$ time. I expect c_l to be the dominant part, I speculate c_l to be around 5 minutes.⁷⁰ Second, I have to request the blocks and have them transferred to me, this is $O(2n \cdot 5c_l)$, again I expect c_l to be the dominant part, around 5 minutes. I expect the total time used to be around 10 minutes because the requests can be send in parallel. Note that the transfer of the blocks is assumed to take less than 2.5 minutes (the latency for the answer from the storage servers), which equals a bandwidth of 110 kbps. If the user and the storage servers are assumed to have this bandwidth, it seems reasonable to conclude that the latency caused by the rerouter is the dominating part of the retrieval.

⁶⁹ I found the text in Project Gutenberg, <http://textual.net/access.gutenberg> (October 2003).

⁷⁰ I have assumed the average time to travel one mix is 20 seconds, and each packet have to travel 7.5 mixes on average. Furthermore a a request and an answer must be sent, resulting in 300 seconds.

8.4.3 Failing Servers

If servers fail it results in a large impact on the performance. As I have written in Section 7.7, a user will not be noticed about a failed server faster than she would get an correct answer. If a user has to send a request p times to get an answer, and retries after c_t minutes, the operation requires at least $p \cdot c_t$ minutes.

Example: If some of the servers fails when I retrieve the document from the previous example, then the effect will be dominated by the maximum number of servers that fail in the location and retrieval of a single block. A single error will add 5 minutes or 50% to the retrieval, two errors doubles the retrieval time to 20 minutes, and so on.

8.4.4 Conclusion

A theoretical analysis of YÅPS shows that it requires $O(n)$ time to publish or retrieve a document of n blocks. Furthermore, the analysis suggests that the latency introduced by the rerouting network is the dominant limiting factor of the performance experienced by users. Failing servers increases the time needed to publish or retrieve a document significantly. But errors on different blocks are not expected to increase the time more than the time needed to fulfill the operations for the block where most servers failed. I believe it will be possible to retrieve documents in less than 60 minutes, which was the goal (Section 3.6).

8.5 Prototype

Though this be madness, yet there is method in it. . .
—Hamlet

I have implemented a prototype to obtain practical experience with parts of YÅPS and to prove that the design works. I have implemented a variant without anonymous communication and using a centralized index service, as time did not allow for a full implementation. Furthermore, the prototype did not support updateable blocks or IDA [46]. Anonymous communication, Chord [57], and IDA have been implemented and tested elsewhere. Anonymous communication have been proved in practice by different implementations including Onion Routingcite[32] and remailers (including a prototype implementation of Mixminion⁷¹).

The prototype consists of the following elements:

- *Index server:* A centralized hash table that maps IDs to IP addresses.
- *Storage server:* A server that can store and return blocks given an ID. The storage server can also return random blocks. The storage server updates the index server.

⁷¹ As of October 30 2003, Mixminion reached version 0.0.5.3, <http://www.mixminion.net/> (November 2003).

- *Client*: Programs that are used to publish or retrieve a document. When a document is published, random blocks are retrieved, new blocks generated from the document and the random blocks, and the new blocks published at different storage servers. A recipe is created, this can be used to retrieve the document by first finding the blocks using the index server, and then retrieving the necessary blocks.

The implementation supports multiple IDs to provide redundancy, as well as replication. The implementation was done in C++ and Perl.⁷²

The prototype was tested in a network of 17 machines, with a central index server and 16 storage servers. The servers are implemented with persistent storage using simple file-based journals. The programs implementing the index and storage servers can be stopped at any point and recover without inconsistencies.

The implementation worked as expected: it is possible to publish and retrieve documents—also if some of the storage servers fail. None of this is surprising, but the prototype confirms the expectations: the design is possible to implement and deploy.

The prototype was fast, but this was also expected, as the prototype did not implement index servers or anonymous communication.

From the experiences with the implementation of the basic infrastructure that has not been tested in other designs I conclude that YÅPS is possible to implement and deploy.

8.6 Summary

In these sections I have presented YÅPS and discussed security, requirements, performance, and described my prototype implementation. The following is a summary of the sections:

Security: Users and storage servers cannot be directly attacked in YÅPS and storage servers are protected against legal attacks by strong deniability. Index servers and rerouters can be directly attacked, which suggests that they should be placed in an area where the operation of such servers are legal. Censorship of the YÅPS is hard to exercise but the system can be made unusable by a denial-of-service attack against the rerouters (or index servers) or by blocking access to the rerouters (or index servers). Under oppressive regimes with tight control over the communication infrastructure it is impossible to operate YÅPS and similar APSs in a safe way.

Requirements: Users who want to publish a document needs a computer with Internet access, a client program, and a list of usable rerouters and storage servers. The requirements for users regarding space, bandwidth, and computation are modest. A storage server primarily needs space to store blocks and

⁷² The implementation consists of C++ programs (3000 lines) and Perl-scripts (500 lines). Most of the C++ code deals with the handling of packets. The C++ library ACE [52] was used to provide network communication in the implementation. The client programs are Perl-scripts that use components implemented in C++, as well as Unix utilities.

bandwidth to serve blocks to users. Bandwidth is also needed to update index servers. Index servers are required to provide bandwidth and a modest amount of space. Rerouting servers are expected to provide a large amount of bandwidth and high availability.

Performance: A theoretical analysis of YÅPS shows that it requires $O(n)$ time to publish or retrieve a document of n blocks. Furthermore, the analysis suggests that the latency introduced by the rerouting network is the dominant limiting factor of the performance experienced by users. Failing servers will increase the time needed to publish or retrieve a document significantly. But errors on different blocks are not expected to increase the time more than the time needed to fulfill the operations for the block where most servers failed.

Prototype: I have implemented a prototype to test parts of the design, which have not been tested elsewhere. The prototype worked, and I conclude that YÅPS is possible to implement and deploy.

9 Conclusions

We are not now that strength which in old days
Moved earth and heaven, that which we are, we are;
One equal temper of heroic hearts,
Made weak by time and fate, but strong in will
To strive, to seek, to find, and not to yield.
—Homer, Ulysses.

In this thesis I have presented *yet another anonymous publication system*—YÅPS. Part of the work has been to specify the goals of YÅPS and discuss related work. The goal of the thesis was a system, which is hard to break by powerful adversaries, but also possible to deploy in an environment with limited trust and resources. I believe this is true of YÅPS.

The problems I have addressed and solved in this thesis include:

- *Providing anonymous communication.*
I have presented a variant of Mixminion [15] for anonymous communication (Section 4.5), which provides more flexibility than other mix-network solutions. The variant is expected to be more secure than OR networks.
- *Finding a way to divide documents into blocks, such that:*
 - *participants can deny knowledge of the content of the blocks they store.*
YÅPS provides strong deniability, based on secret sharing, for storage servers (Section 5), I believe this is done better than in any existing system.
 - *documents can be recreated even if some of the servers fail.*
YÅPS use IDA [46] and replication to create redundancy among blocks (Section 7.2), and I have suggested a design based on Chord [57], which provides redundancy of the information needed to locate blocks (Section 7.4).
- *Making it possible to locate and retrieve the blocks needed to recreate a document within reasonable time.*
I have showed that users will be able to retrieve documents from YÅPS within around 10 minutes (Section 8.4), which meets the goal (Section 3.6).

I have implemented a prototype to test parts of the design, which have not been tested by others. The prototype worked successfully and I conclude that the design is possible to implement and deploy.

Furthermore, I have discussed anonymous publication in a political and ethical context to put my work into a greater perspective. I have also presented a number of issues concerning practical deployment of an APS, including: operation under oppressive regimes, trust, and legal issues.

9.1 Future Work

The design I present must be analyzed by others and a full prototype implemented and tested in practice. In the work with YÅPS I have also identified some areas, which deserve further research:

- *Robust anonymous communication*: Efficient rerouting schemes require rerouters with high availability. Is it possible to design an efficient rerouting system, which provides strong protection and provide better failure resistance than current designs? For example, is it feasible to build a hybrid system where a DC-network-based scheme is used to communicate between rerouters?
- *APSs under oppressive regimes*: APSs are often presented as a means for people living in oppressive regimes, but the use in such environments introduces many problems. A classification of different environments and their requirements of an APS is needed.
- *Hiding communication*: Can data be communicated as emails that look like spam? Can CAPTCHAs [60] be used to create efficient hidden communication channels?

Another interesting area outside computer science that needs further research is the legal status of APSs: in which nations are people who provide a storage service responsible for the content? And where does strong denial work as a legal defense?

A Reply Block Example

But since the aliens only delivered super technology and not magic when they crashed we'll just have to make do with faking moon landings and tinfoil hats.

—Unknown

In the following I shall explain how reply blocks works by giving an example of how Bob can create a reply block and show how Alice can use Bob's reply block to send a message to Bob. The example shows how asymmetric and symmetric keys are combined. I use an example with two mixes on the path chosen by Bob.

In the example I shall use $a|b$ to denote a concatenated with b and $e(k,p)$ to denote the ciphertext made by encrypting the string p with the key k .

Bob wants to receive a message anonymously, and creates a reply block

Bob starts by choosing two mixes, m_0 and m_1 , and creates two symmetric keys k_0 and k_1 . Bob then encrypts his own address with k_0 , concatenates the result with k_0 and encrypts the result with m_0 's public key. The new result is concatenated with m_0 's address to get:

$$a_{m_0}|e(pub_{m_0}, k_0|e(k_0, a_B))$$

This string is padded and encrypted with k_1 . The result is then concatenated with k_1 and encrypted with m_1 ' public key. Concatenated with the address of m_1 , it results in the reply block r_B :

$$r_B : m_1|e(pub_{m_1}, k_1|e(k_1, a_{m_0}|e(pub_{m_0}, k_0|e(k_0, a_B))|pad_B))$$

The padding is added if the reply block is shorter than a predefined reply block size (for example if a header supports a path of n mixes, but fewer mixes are used to create the path).

Alice wants to send a message to Bob

Alice obtains r_B and can use this to send a message to Bob. Alice uses r_B as the header of her packet, and inserts the message M as the payload:

Header: $m_1|e(pub_{m_1}, k_1|e(k_1, a_{m_0}|e(pub_{m_0}, k_0|e(k_0, a_B))|pad_B))$
Payload: M

Alice sends the packet to m_1 .

A packet arrives at $m1$

At $m1$ the address is removed from the header and the first part is decrypted with $m1$'s private key to get k_1 . The rest of the header is padded with 0s to get:

$$e(k_1, a_{m0}|e(pub_{m0}, k_0|e(k_0, a_B))|pad_B)|pad_0$$

This string is decrypted with k_1 , including pad_0 to get:

$$a_{m0}|e(pub_{m0}, k_0|e(k_0, a_B))|pad_B|pad'_0$$

I use ' to note that pad_0 has been transformed to a pseudo-random string pad'_0 . The next destination is now known to be $m0$. And the header has the same size as the original header. The payload is encrypted with k_1 to get the packet to be resent:

$$\begin{array}{ll} \text{Header:} & a_{m0}|e(pub_{m0}, k_0|e(k_0, a_B))|pad_B|pad'_0 \\ \text{Payload:} & e(k_1, M) \end{array}$$

The packet is sent to $m0$.

A packet arrives at $m0$

At $m0$ the address is removed from the header and the first part is decrypted with $m0$'s private key to get k_0 . The rest of the header is padded with 0s to get:

$$e(k_0, a_B)|pad_B|pad'_0|pad_0$$

This string is decrypted with k_0 :

$$a_B|pad'_B|pad''_0|pad'_0$$

Decrypting the padding is necessary to prevent tagging attacks. The pseudo-random transformation ensures that it is impossible to tag a padded header, as the next mix transforms the header inclusive padding.

The next destination is now known to be B . The payload is encrypted with k_0 :

$$\begin{array}{ll} \text{Header:} & a_B|pad'_B|pad''_0|pad'_0 \\ \text{Payload:} & e(k_0, e(k_1, M)) \end{array}$$

The packet is sent to B .

Bob receives a packet

Bob receives the packet. He can discard the header and use k_0 and k_1 to decrypt the payload to get M .

A.1 Integrity Protection of Header

I did not include integrity protection of the header in the example, as this would make the explanations more complicated. In a real system, Bob would include integrity protection in the header. Bob can include integrity protection because padding a the mixes is done by a pattern known by Bob. Take the header seen by m_0 :

$$a_B | pad'_B | pad''_0 | pad'_0$$

In a real system this would include a secure hash, h_0 , of the rest of the header:

$$a_B | h_0 | pad'_B | pad''_0 | pad'_0$$

And m_0 would check that:

$$h(pad'_B | pad''_0 | pad'_0) = h_0$$

Bob can generate and include h_0 because he knows the content of the rest of the header and the transformations on this. In this explanation there is only padding left in the header, because it the header seen by the last mix on the path, but it works the same way for the headers seen by any mix on a path.

B Communication Issues

What we have got here is failure to communicate. Some men you just can't reach.

—Captain, Cool Hand Luke (1967)

In the following I shall discuss a few issues related to communication in a practical implementation.

I shall discuss the communication layer in Section B.1 and the payload size for anonymous communication in Section B.2.

B.1 Communication Layer

In YÅPS, the users communicate with the storage and index servers using the rerouting network. The communication consists of messages sent as packets through the rerouting network. Rerouters also communicate with each other, this is also done using rerouting packets. Rerouting packets cannot be expected to come in order, this requires the communicating parties to reorder the packets if this is necessary. The rerouting network does not provide reliable communication of messages as the rerouters can fail, therefore the programs used by communicating parties must handle this.

The index servers communicate with each other to maintain the Chord infrastructure, using Chord messages.

A reliable transport protocol is required to communicate the packets and messages. TCP can be used, but it is probably more efficient to use UDP with a simple protocol to support reliability. Furthermore, other transport protocols, such as SMTP and HTTP, should preferably be supported, for users and storage servers, which have limited network access. Users can have limited network access if they are on a network without a direct Internet connection or if the Internet connection is controlled by an adversary. SMTP is inefficient, but it is trivial to let the rerouters send and receive email. HTTP can be used if the rerouters hosts a special web server. Requests from a user can be encoded in an URL and the server can respond with an URL where the result of the request can be found. The rerouter can then fulfill the request and place the result on the announced location. If possible, all servers should provide HTTP and SMTP communication to allow a maximum of flexibility.

B.2 Payload Size for Anonymous Communication

The payload size in the packets used for anonymous communication corresponds to the maximum message size. The packets have fixed size, this implicates that a large payload size results in a large overhead for small messages.

The packet size is determined by the payload size as well as the space needed for routing information. In Mixminion [15], the headers requires 4kB.

In most communications there are at least one small packet communicated for each block transferred. For example, a request for a specific or random block to a storage server results in the transferring of one block (or none).

I have decided to use a block size of 100kB (Section 7.5.4). Based on calculations of the traffic needed to publish documents and retrieve them 10 times, I propose a payload size of 20kB.

List of Internet References

- Anonymizer, US sponsors Anonymiser, <http://www.theregister.co.uk/content/6/32922.html> (November 2003).
- CAPTCHA Project, the, <http://www.captcha.net/> (November 2003).
- Chaos Communication Camp 2003, Conference, <http://www.ccc.de/camp/2003/conference/index.en.html> (November 2003).
- Cypherpunk remailer, a collection of emails describing the development of the Cypherpunk remailer, <http://cryptome.org/zks-v-tcm.htm> (November 2003).
- Cypherpunks, homepage, <http://www.csua.berkeley.edu/cypherpunks/Home.html> (November 2003).
- DeCSS, The Openlaw DVD/DeCSS Forum Frequently Asked Questions (FAQ) List, Harvard Lawschool, <http://cyber.law.harvard.edu/openlaw/DVD/dvd-discuss-faq.html> (November 2003).
- Distributed Systems Lab., University of Copenhagen, <http://www.distlab.dk> (November 2003).
- Electronic Frontier Foundation, <http://www.eff.org/issues/usapa/> (October 2003).
- European Conference for Object-Oriented Programming, Workshop on Communication Abstractions for Distributed Systems, <http://www.ecoop.tu-darmstadt.de/workshops/03.phtml> (November 2003).
- Free Haven
 - development mailing list, on entanglement, April 15 2001, <http://archives.seul.org/freehaven/dev/Apr-2001/msg00007.html> (November 2003).
 - project homepage <http://freehaven.net/> (November 2003).
- Freenet
 - China News on Freenet (in Chinese), <http://freenet-china.org/> (November 2003).
 - Fracturing P2P Networks article on Slashdot, <http://slashdot.org/articles/03/10/06/1058250.shtml> (November 2003).
 - misuse of, discussion on Slashdot, <http://yro.slashdot.org/article.pl?sid=03/09/03/226243> (October 2003).
 - on use in China, GrepLaw: *Ian Clarke on Freenet and his Decision to Leave the USA*, <http://grep.law.harvard.edu/article.pl?sid=03/09/02/0125236> (November 2003).

- practical performance, discussion on *freenet-support* mailing list, <http://www.mail-archive.com/support@freenetproject.org/msg01955.html> (November 2003).
- Hacktivism, homepage, <http://hacktivism.com> (November 2003).
- Hash Cash, homepage, <http://www.hashcash.org> (November 2003).
- International System of Units (SI) prefixes, <http://physics.nist.gov/cuu/Units/prefixes.html> (November 2003).
- Java Anonymous Proxy
 - case of, The Register, <http://theregister.co.uk/content/55/32450.html>, (October 2003).
 - homepage http://anon.inf.tu-dresden.de/index_en.html (October 2003).
- Lawsuits
 - American Health Scan v. Technical Chem. and Prods., Inc., http://www.cybersecuritieslaw.com/lawsuits/cases_corporate_cybersmears.htm, (November 2003).
 - Florida Bar Computer Law Committee Online Journal, the, *Online Service Provider Liability for Defamation*, http://www.computer.flabar.org/2_spring96/online.html (November 2003).
 - MP3 Site Sues RIAA Over Linking, in Wired Magazine, <http://www.wired.com/news/print/0,1294,36778,00.html> (November 2003).
 - Napster, close down, Find Law, Legal News and Commentary, <http://news.findlaw.com/legalnews/lit/napster/> (October 2003).
 - United States District Court for the District of Columbia, Civil Action No. 98-116, http://www.loundy.com/CASES/McVeigh_v_Cohen.html (November 2003).
- Kazaa Homepage, <http://www.kazaa.com> (November 2003).
- Mixmaster
 - Remailer Attacks, <http://www.obscura.com/~loki/remailer/remailer-essay.html> (November 2003).
 - Remailer List, <http://anon.efga.org/Remailers/TypeIIList/pubring.mix> (November 2003).
- Mixminion, project homepage, <http://www.mixminion.net/> (November 2003).
- Ogle, DVD player package for Debian Linux, <http://packages.debian.org/stable/graphics/ogle.html>, October 2003.

- Penet remailer, wikipedia article, http://www.wikipedia.org/wiki/Penet_remailer (October 2003).
- Project Guthenberg, <http://textual.net/access.gutenberg> (October 2003).
- Soros foundations network, the, <http://www.soros.org/> (October 2003).
- Timothy McVeigh, case of, <http://www.infoplease.com/ipa/A0882060.html> (October 2003).

References

- [1] Ross Anderson. The Eternity Service. In *Proceedings of Pragocrypt '96*, 1996. URL citeseer.nj.nec.com/anderson96eternity.html.
- [2] David Banisar, Gus Hosein, Simon Davies, Heather Ford, Karen Banks, and Wendy Grossman. Silenced, an international report on censorship and control of the internet. Technical report, Privacy International and the GreenNet Educational Trust, September 2003. Available from www.privacyinternational.org/survey/censorship.
- [3] Krista Bennett and Christian Grotho. gap - practical anonymous networking, 2003. URL citeseer.nj.nec.com/bennett02gap.html. In 3rd Workshop on Privacy Enhancing Technologies (PET 2003), Dresden, March 2003. Post-conference proceedings to be published by Springer. <http://www.ovmj.org/GNUnet/papers.php3?xlang=English>.
- [4] Krista Bennett, Christian Grothoff, Tzvetan Horozov, and Ioana Patrascu. Efficient sharing of encrypted data. In Lynn Batten and Jennifer Seberry, editors, *Proceedings of the 7th Australasian Conference on Information Security and Privacy, ACISP 2002*, volume 2384 of *Lecture Notes in Computer Science*, pages 107–120. Springer-Verlag, Heidelberg, 2002. URL citeseer.nj.nec.com/article/bennett02efficient.html.
- [5] Oliver Berthold, Andreas Pfitzmann, and Ronny Standtke. The disadvantages of free MIX routes and how to overcome them. In *International workshop on Designing privacy enhancing technologies*, pages 30–45. Springer-Verlag, Heidelberg, 2001. ISBN 3-540-41724-9.
- [6] Alan Brown. *Peer-to-Peer: Harnessing the Benefit of a Disruptive Technology*, chapter 10, pages 133–144. O'Reilly, Sebastopol, CA, 2001.
- [7] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2), February 1981. <http://www.eskimo.com/~weidai/mix-net.txt>.
- [8] David L. Chaum. The Dining Cryptographers Problem: Unconditional Sender and Receptient Untraceability. *Journal of Cryptography*, 1:65–75, 1988. <http://www.cam.cornell.edu/~sharad/herbivore/dcnets.html>.
- [9] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM Press, 2003. ISBN 1-58113-735-4.
- [10] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002. URL citeseer.nj.nec.com/article/clarke02protecting.html.

- [11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 2000*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer-Verlag, Heidelberg, 2001. URL citeseer.nj.nec.com/clarke00freenet.html.
- [12] Richard Clayton, George Danezis, and Markus G. Kuhn. Real world patterns of failure in anonymity systems. In *Information Hiding: 4th International Workshop, IHW 2001*, volume 2137 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, Heidelberg, January 2001. <http://citeseer.nj.nec.com/clayton01real.html>.
- [13] Lance Cottrell. Mixmaster & remailer attacks. Cottrel’s work is unpublished, but is mentioned in many publication, including [54] and [16], <http://www.obscura.com/~loki/remailer/remailer-essay.html> September 2003, 1998.
- [14] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
- [15] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol, 2003. <http://www.mixminion.net/>.
- [16] Claudia Diaz and Andrei Serjantov. Generalising mixes. To appear in *Proceedings of Privacy Enhancing Technologies*, March 2003, Dresden (Germany). <http://www.esat.kuleuven.ac.be/~cdiaz/activities.html>, 2003.
- [17] Claudia Diaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Information theory and anonymity. In B. Macq and J.-J. Quisquater, editors, *Proceedings of the 23rd Symposium on Information Theory*, 2002. Belgium. <http://www.esat.kuleuven.ac.be/~cdiaz/activities.html>.
- [18] Claudia Diaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In Dingledine and Syverson, editors, *Proceedings of Privacy Enhancing Technologies*, volume 2482 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, Heidelberg, 2002. <http://www.esat.kuleuven.ac.be/~cdiaz/activities.html>.
- [19] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999. URL <ftp://ftp.internic.net/rfc/rfc2246.txt>, <ftp://ftp.math.utah.edu/pub/rfc/rfc2246.txt>. Status: PROPOSED STANDARD. <ftp://ftp.internic.net/rfc/rfc2246.txt>.
- [20] Roger Dingledine, Michael J. Freedman, and David Molnar. *Peer-to-Peer: Harnessing the Benefit of a Disruptive Technology*, chapter 12, pages 159–190. O’Reilly, Sebastopol, CA, 2001.

- [21] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 2000*, volume 2009 of *Lecture Notes in Computer Science*, pages 67–95. Springer-Verlag, Heidelberg, 2001. URL citeseer.nj.nec.com/dingledine00free.html.
- [22] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router, November 2003. Unpublished, submitted November 2003, <http://freehaven.net/tor/>.
- [23] Shlomi Dolev and Rafail Ostrovsky. Xor-trees for efficient anonymous multicast and reception. *ACM Transactions on Information and System Security*, 3(2):63–84, May 2000. <http://citeseer.nj.nec.com/dolev98xortree.html>.
- [24] Lyn Dupré. *Bugs in Writing: A Guide to Debugging Your Prose*. Addison-Wesley, Boston, 1995.
- [25] Carl Ellison and Bruce Schneier. Ten risks of pki: What you’re not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000. <http://www.schneier.com/paper-pki.html>.
- [26] Phil Zimmermann et al. An introduction to cryptography, 2000. Part of PGP 7.0 User’s Guide, <http://www.pgpi.org/doc/guide/7.0/en/intro/>.
- [27] Ronaldo A. Ferreira, Christian Grothoff, and Paul Ruth. A transport layer abstraction for peer-to-peer networks. In *Proceedings of the Third International Workshop on Global and Peer-to-Peer Computing (G2P2PC)*, 2003. <http://www.ovmj.org/GNUnet/papers.php3?xlang=English>.
- [28] Michael J. Freedman and Robert Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, D.C., November 2002. URL citeseer.nj.nec.com/freedman02tarzan.html.
- [29] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In Joe Kilian, editor, *Proceedings of CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2001.
- [30] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O’Reilly & Associates, Inc., 2nd edition, 1996.
- [31] Sharad Goel, Mark Robson, Milo Polte, and Emin Gün Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003. Computing and Information Science Technical Report, TR2003-1890.
- [32] David M. Goldschlag, Michael G. Reed, , and Paul F. Syverson. Onion Routing for Anonymous and Private Internet Connections. *Communications of the ACM*, 42(2), February 1999. <http://www.onion-router.net/Publications.html>.

- [33] Yong Guan, Xinwen Fu, Riccardo Bettati, and Wei Zhao. An Optimal Strategy for Anonymous Communication Protocols. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 257. IEEE, 2002. URL citeseer.nj.nec.com/guan02optimal.html.
- [34] Markus Jakobsson. Flash Mixing. In *Proceedings of Principles of Distributed Computing - PODC '99*. ACM Press, 1999. URL <http://www.rsasecurity.com/rsalabs/staff/bios/mjakobsson/flashmix/flash%mix.pdf>.
- [35] Gene Kan. *Peer-to-Peer: Harnessing the Benefit of a Disruptive Technology*, chapter 8, pages 94–122. O'Reilly, Sebastopol, CA, 2001.
- [36] Dennis Kügler. An analysis of gnunet and the implications for anonymous, censorship-resistant networks, 2003. URL citeseer.nj.nec.com/577663.html. In 3rd Workshop on Privacy Enhancing Technologies (PET 2003), Dresden, March 2003. Post-conference proceedings to be published by Springer. URL <http://www.ovmj.org/GNUnet/papers.php?xlang=English>.
- [37] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002. ISBN 1-58113-483-5.
- [38] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, pages 59–72, August 2000. URL citeseer.nj.nec.com/waldman00publius.html.
- [39] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 1997. ISBN 0-8493-8523-7.
- [40] Ira S. Moskowitz and Myong H. Kang. Covert channels - here to stay? In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 235–244, Gaithersburg, MD, 1994. National Institute of Standards and Technology.
- [41] Miyaku Ohkubo and Masayuki Abe. A Length-Invariant Hybrid MIX. In *Proceedings of ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2000.
- [42] Andreas Pfitzmann and Maria Köhntopp. Anonymity, Unobservability, and Pseudonymity – A Proposal for Terminology. In Hannes Federath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 1–9. Springer-Verlag, Heidelberg, 2001.
- [43] Andreas Pfitzmann and Maria Köhntopp. Anonymity, Unobservability, and Pseudonymity – A Proposal for Terminology, 2003. Draft v0.14, based on [42].
- [44] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997. URL citeseer.nj.nec.com/plaxton97accessing.html.

- [45] Karl R. Popper. *Open Society and Its Enemies (Volume 2)*. Princeton University Press, 1971.
- [46] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989. ISSN 0004-5411.
- [47] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001. URL citeseer.nj.nec.com/ratnasamy01scalable.html.
- [48] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications*, 16(4), 1998. URL citeseer.nj.nec.com/20193.html.
<http://www.onion-router.net/Publications.html>.
- [49] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, November 1998. ISSN 1094-9224.
- [50] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001. URL citeseer.nj.nec.com/article/rowstron01pastry.html.
- [51] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002. URL citeseer.nj.nec.com/saroiu02measurement.html.
- [52] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Vol. 1: Mastering Complexity with ACE and Patterns*. C++ In-Depth Series. Addison-Wesley, Boston, 2002.
- [53] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., New York, NY, USA, 1994. ISBN 0-471-59756-2 (paper).
- [54] A. Serjantov, R. Dingledine, and P. Syverson. From a trickle to a flood: active attacks on several mix types. In *Information Hiding: 5th International Workshop, IH 2002*, volume 2578 of *Lecture Notes in Computer Science*, pages 36–72. Springer-Verlag, Heidelberg, 2003. citeseer.nj.nec.com/serjantov02from.html.
- [55] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979. ISSN 0001-0782.
- [56] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A Protocol for Scalable Anonymous Communication. In *Proceedings of Symposium on Security and Privacy, May 12 - 15, 2002, Berkeley, California*, page 58. IEEE, 2002. URL citeseer.nj.nec.com/508163.html.

- [57] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001. ISBN 1-58113-411-8. URL citeseer.nj.nec.com/article/stoica01chord.html.
- [58] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical report, MIT, January 2002. Technical Report. Available from <http://www.pdos.lcs.mit.edu/chord/>.
- [59] Srdjan Čapkun, Levente Buttyán, and Jean-Pierre Hubaux. Small worlds in security systems: an analysis of the PGP certificate graph. In *Proceedings of the ACM New Security Paradigms Workshop, 2002.*, 2002. URL citeseer.nj.nec.com/capkun02small.html.
- [60] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2003. http://www.cs.cmu.edu/~biglou/captcha_crypt.pdf.
- [61] Marc Waldman and David Mazi. Tangler: a censorship-resistant publishing system based on document entanglements. In *ACM Conference on Computer and Communications Security*, pages 126–135, 2001. URL citeseer.nj.nec.com/waldman01tangler.html.
- [62] Ben Y. Zhao, John Kubiawicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, april 2001. URL citeseer.nj.nec.com/zhao01tapestry.html.