Design and Analysis of an Anonymous Communications Channel for the Free Haven Project

by

Michael J. Freedman

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

The Free Haven Project aims to deploy a system for anonymous, distributed data storage which is robust against attempts by powerful adversaries to find and destroy stored data. To support anonymous publishing and retrieval operations, we use a mixnet for communications. We provide an outline of a formal definition of communications anonymity, to help characterize and compare real-world projects and their success at achieving protection from disclosing identity. In addition, we describe a variety of attacks on the system and ways of protecting against these attacks. We discuss our design and implementation of a communications module to interface between the Free Haven publishing system and the communications channel. Finally, based on our analysis of anonymity and other projects, we present some future directions for designing robust anonymous communications channels.

Thesis Supervisor: Ron Rivest
Title: Webster Professor of Computer Science and Engineering

Contents

Chapter 1

Introduction and Requirements

1.1 Motivation

Every man should know that his conversations, his correspondence, and his personal life are private. — Lyndon B. Johnson, president of the United States, 1963–69

Over the past decade, the Internet has seen explosive growth in both the number of users and the sheer quantity of available materials. This expansion has changed its role from the realm of academic research towards mainstream use. As more conventional activities are performed on the Internet everyday – reading the New York Times, shopping for clothes, cars, and electronics, finding information about weekend entertainment events, posting baby pictures for relatives, and ordering books and movies for home delivery, among many others – the potential becomes ever greater that personal information will be monitored and logged.

The New Yorker magazine explained in a famous cartoon, "On the Internet, nobody knows you're a dog" [30]. Unfortunately, the opposite has become increasingly true. While people do not communicate face-to-face in this medium, identities can be assigned to Internet users nonetheless. Virtually every email a user sends, every post to a newsgroup, every purchase made online, and every World Wide Web page accessed can be monitored and logged by some third party. Furthermore, the growth of database use, the technological trend of cheaper and larger hard disk storage, and the increased speed of communications channel all provide the means for easier storage of personal information.

In these ways, the Internet provides an improved means of forming personal dossiers on individuals. A collection of information from diverse sources, these dossiers may be created and used by a multitude of entities: governments, corporations, industries, organizations, and other individuals. The potential employer can purchase an applicant's social security number online, then run background checks with a click of the mouse. An advertisement agency or telemarketer can pinpoint exactly the consumer base to contact. Databases are being placed onto the Internet; with today's advanced search and data-mining technologies, electronic cross-linking becomes ever greater.

The lack of privacy for online activities is fairly obvious. Email headers include the routing paths of email messages, including DNS names and IP addresses. Web browsers normally display user IP addresses; web servers can be configured to log this information, the servers that referred clients to it, and the frequency and times of user accesses. Cookies on a client's browser may be used to store persistent user-information. Commonly-used online chat applications such as ICQ and Instant Messenger also divulge IP addresses. With additional effort – such as automatically `finger`ing the client machine shortly after accesses – a remote party can often deduce the exact user with high probability.

But the Internet also offers the potential for greater personal privacy. Enabling technologies can bring privacy to areas and activities where that was previously impossible. Communications anonymity provides a means for reaching this goal. This type of anonymity "blinds" any information that may be divulged on a communications channel between any two or more parties.

This paper addresses anonymity considerations for the selection of a proper communications channel. Our motivation is the design and implementation of the Free Haven system [10], an anonymous publication system. We present some intuitive notions of types or degrees of anonymity, and use models of adversaries and types of attacks to assess real-world projects in the face of these definitions of anonymity. These models will be used elsewhere to further analyze the functional dependence between publishing and communication agent anonymity.

1.2 Background: The Free Haven Project

The Free Haven Project aims to deploy a system for distributed data storage which is robust against attempts by powerful adversaries to find and destroy stored data.[1] Free Haven uses a secure mixnet for communication, and it emphasizes distributed, reliable, and anonymous storage over efficient retrieval. We provide an outline of a formal definition of anonymity, to help characterize the protection that Free Haven provides, and also to help compare related services. Some of the problems Free Haven addresses include providing sufficient accountability without sacrificing anonymity, building trust between servers based entirely on their observed behavior, and providing user interfaces that will make the system easy for end-users.

[1]The motivation and project summary of the Free Haven project, presented here, are written by Roger Dingledine.

1.2.1 Free Haven Motivations

Increases in speed and efficiency on the Internet have not brought comparable increases in privacy, not only in terms of anonymous communications, but also in regards to anonymous publishing. Court cases, such as the Church of Scientology's lawsuit against Johan Helsingius [16] or the more recent OpenDVD debate [23] (and subsequent arrest of DeCSS author Jon Lech Johansen), demonstrate that the Internet currently lacks an adequate infrastructure for truly anonymous publication or distribution of documents or other data.

Indeed, there are a number of other deeper motivations for the deployment of an anonymous publishing service such as Free Haven. Not only do we hope to assist those like Helsingius and Johansen, but we have the wider goals of pushing the world a few more steps in the direction of free and open information and communication. In Germany, Internet Service Providers such as AOL are liable for the content that passes across their systems [34]. Recent British legislation threatens to make citizens responsible for the content of encrypted documents that they're holding, even if they don't possess the ability to read these documents [31]. Such restrictions on the free flow of information, however, are already being attacked: for example, American hackers are attempting to break holes in China's "Great Firewall" to allow Chinese citizens access to Western media [4].

In addition to such revolutionary actions, there are a wide range of activist projects which employ the internet for publicity but focus on helping real people in the real world. Such projects include Pirate Radio [9], a loose confederation of radio operators joined in the belief that ordinary citizens can regulate the airwaves more efficiently and more responsibly than a government organization; as well as mutual aid societies such as Food Not Bombs! [18], an organization which "serves free food in public places to dramatise the plight of the homeless, the callousness of the system and our capacity to solve social problems through our own actions without government or capitalism."

By providing tools to enable safer and more reliable communication for organizations fighting for increased rights of individuals rather than nations or corporations, as well as strengthening the capabilities of political dissidents and other individuals to speak out anonymously about their situations, the members of the Free Haven Project hope to help pave the way to a modern society where freedom of speech and freedom of information are integral parts of everyday life.

1.2.2 Project Summary

The Free Haven Project intends to deploy a system which provides a good infrastructure for anonymous publication. Specifically, this means that the publisher of a given document should not be known; that clients requesting the document should not have to identify themselves to anyone; and that the current location of the document should not be known. Additionally, it would be preferable to limit the number of opportunities where an outsider can show that a given document passed through a given computer. A more thorough examination of our requirements and notions of anonymity can be found in Chapter 2.

The overall design is based on a community of servers (which as a whole is termed the 'servnet') where each server hosts data from the other servers in exchange for the opportunity to store data of its own in the servnet. When an author wishes to publish a document, she breaks the document into shares, where a subset (any k of n) is sufficient to reconstruct the document. Then for each share, she negotiates for some server to publish that share on the servnet. The servers then trade shares around behind the scenes. When a reader wishes to retrieve a document from the servnet, she requests it from any server, providing a location and key which can be used to deliver the document in a private manner. This server broadcasts the request to all other servers, and those which are holding shares for that document encrypt them and deliver them to the reader's location. Also behind the scenes, the shares employ what is essentially the 'buddy system' to maintain some accountability: servers which drop shares or are otherwise unreliable get noticed after a while, and are trusted less. A trust module on each server maintains a database on the behavior of each other server, based on past direct experience and also what other servers have said. For communication both between servers and between the servnet and readers, we rely on an existing mixnet infrastructure to provide an anonymous channel.

The system is designed to store data without concern for its popularity or controversial nature. Possible uses include storing source code or binaries for software which is currently under legal debate, such as the recent DeCSS controversy or other software with patent issues; publishing political speech in an anonymous fashion for people afraid that tying their speech to their public persona will damage their reputation; or even storing more normal-looking data like a set of public records from Kosovo.

Free Haven is designed more for anonymity and persistence of documents than for frequent querying — we expect that in many cases, interesting material will be retrieved from the system and published in a more available fashion (such as normal web pages) in a jurisdiction where such publishing is more reasonable. Then the document in the servnet would only need to be accessed if the other sources were shut down.

The potential adversaries are many and diverse: governments, corporations, and individuals all have reason to oppose the system. There will be social attacks from citizens and countries trying to undermine the trust in the security of the system, as well as attacking the motivation for servnet node operators to continue running nodes. There will be political attacks, using the influence of a country's leaders to discourage use of the servnet. There will be government and legal attacks, where authorities attempt to shut down servnet nodes or arrest operators. Indeed, in many cases ordinary citizens can recruit the power of the government through lawsuits or subpoenas. Multinational corporations will hold sway over several countries, influencing them to pass similar laws against anonymous networks. There will be technical attacks, both from individuals and from corporations and national intelligence agencies, targeted either at the system as a whole or at particular documents or node operators, to reduce the quality of service or gain control of part of the network. Clearly the system needs to be designed with stability, security, and longevity in mind.

1.3 Design Requirements

In order to fulfill the anonymity goals of the Free Haven system, we require a minimum level of anonymity to be provided by any communications channel which we utilize. Beyond that, requirements for such a channel fall into two categories:

- Required Operations:
  - The channel must provide a mechanism for anonymously **sending** a message to a node.
  - The channel must provide a mechanism for anonymously **broadcasting** a message to known nodes.
  - The system must provide a mechanism for pseudonymously **naming** a node within the network.
  - The system must provide a mechanism for smoothly **adding nodes** to the communications channel without impacting functionality.
  - The system must provide a mechanism for **removing nodes** from the channel.
- Desired Goals:
  - The channel should have sufficiently **low latency** to provide timely message transmission between parties according to the system requirements.
  - The channel should provide **delivery robustness** for messages, so that communicating parties may assume that messages are reliably transmitted.
  - The channel should provide **routing robustness** between any two parties: the loss of a non-trivial percentage of participating nodes should not imply significant or permanent loss of the ability to anonymously communicate between these parties.
  - The system should be **resistant to attack**, both traffic analysis, computational attacks, and similar timing, coding, and volume attacks.
  - The system must be **decentralized** to maintain efficiency, security, and reliability: no single node or small subset of the nodes should be a bottleneck anywhere in the channel.

This paper discusses the design and implementation of a communications module for the Free Haven system. In chapters 2 and 3, we build an intuitive notion of types of anonymity and possible system attacks. In chapter 4, we describe some real-work projects that may fulfill our communications channel requirements. We present a design and implementation of the Free Haven *comm* module in chapter 5, to interface between the underlying publishing system protocols and communications layer. We finish by describing some new ideas for communications channels in chapter 6, with the aim of providing strong anonymity and fulfilling more of our other desired goals.

1.4 About this document

As the Free Haven project is a collaborative effort of several individuals, sections of this paper were formulated and written by other students. The other project members contributing to this document are Roger Dingledine (Sections 1.2 and 2.3), David Molnar (Section 3.3 and Appendix B), and Brian Sniffen (Section 6.3).

Chapter 2

Defining Communications Anonymity

Many projects are focused on the design and deployment of an anonymous communications channels. We present an extensive related works section that enumerate these various implementations. While many present some degree of anonymity, most of these systems generally fail to specify what protections agents do and do not receive.

This section attempts to define the various types and degrees of anonymity on a communications channel. We first describe what a speaker might expect to achieve with regards to privacy. We list some agents in an anonymous communication channel, and address anonymity for each of these agents separately. We conclude the section by presenting some intuitive notions of anonymity, to be used in later for discussing attacks and actual communications channels.

2.1 Defining Privacy

In particular, the speaker may have control in several different dimensions over dissemination of information about his own speech:

1. **Linkability:** The speaker controls the ability of readers to link his utterances. The distinguishing characteristics which provide this linkability are called a pseudonym.

2. **Readers:** The speaker controls which other parties will be able to read his speech.

3. **Persistence:** The speaker controls how long his speech persists after the publication.

4. **Content Leaks:** The speaker controls how much partial information is leaked based on the content of his speech. For instance, the speaker may choose to reveal certain personal

credentials.

5. **Channel Leaks:** The speaker controls how much partial information is leaked based on the communications channel he chooses to use.

6. **Replies:** The speaker controls whether readers can reply to his utterances. Further parameters include whether the reply is private, and the persistence of the ability to reply.

2.2 Agents and Operations

The above list describes some freedoms which a given *speaker* may have available to him. A number of these freedoms have analogs when considered in the context of other parties in a publishing system.

In order to fulfill the insertion and retrieval of documents to remote servers, publication agents make use of a communications channel. The agents on this channel include the message sender, the message receiver, and nodes within the channel. They must support some generalized `send` primitive for point-to-point and/or multi-cast functionality. The design of this channel dictates the extent of protection provided against channel leaks. The choice of channel also dictates the method in which message replies are performed. We consider the various types of anonymous communication properties.

2.2.1 Sender-anonymity

Sender-anonymity means that the identity of the party who sends a message remains hidden, while making no claim as to the anonymity of the recipient or message itself. This characteristic of anonymity is an integral part of virtually any anonymous communications channel. As is the case with receiver-anonymity as well, sender-anonymity may take the form of a true anonymity or pseudonymity, depending upon the context and goals of a communications channel. We shall consider these forms of anonymity in greater depth in section 2.3.1.

2.2.2 Receiver-anonymity

Receiver-anonymity means that the identity of a message's recipient is hidden. For true receiver-anonymity, this identity should be unknown to both the sender of the message and any third party that might be monitoring communications. Secret drop-boxes for spy (or kidnapping) networks are a non-technical example of historical channels that provide receiver-anonymity.

2.2.3 Unlinkability of sender and receiver

The unlinkability of sender and receiver describes an end-to-end notion of anonymity, as both agents exist at the endpoints of an anonymous communications channel as we have defined them. Even if a sender and receiver are both known to have taken part in *some* communication, they should not be identifiable as communicating with each other.

2.2.4 Node-anonymity:

Node-anonymity refers to the anonymity of servers which comprise a communications channel. These servers should not be identifiable as nodes within this channel; to ensure this notion, the presence of communication to and from a node also should not be perceived. This protection is crucial for jurisdictions in which possession of a message imparts some risk, even for as temporary a time as necessary for transmission. Furthermore, this type of anonymity provides protection from localities and situations in which even association with an anonymous communications channel is dangerous.

2.2.5 Carrier-anonymity

Carrier-anonymity means that a node should not be identifiable as a carrier involved with communicating a message between some sender and recipient. This is a weaker form of privacy than node-anonymity. In the former case, a third party should not be able to determine whether a node might be involved in *any* communication; in this case, a node should not identifiable as a link in a *specific* communication.

2.3 Anonymity Characteristics

The speaker has control over whether to speak over a given channel, based on the characteristics of that particular channel. This means that he might tailor his speech, or choose not to speak at all, based on how much protection the channel provides and how much anonymity he desires.

The above classes of anonymity describe the issues regarding each of the agents or operations in the system. However, there are some other broader characteristics of anonymity to consider. [1]

2.3.1 Linkability: Anonymity vs. Pseudonymity

Anonymity, when compared to pseudonymity, means that the agent performing the operation has no observable persistent characteristics. For instance, turning on a radio is a nice way of receiving information anonymously (modulo Tempest attacks). Reading the advertisements in the New York

[1] This characteristics section appears almost verbatim in [10].

Times is another good example, though again it's not perfect, since it seems conceivable that somebody could track who purchases a given newspaper. In general, broadcast media like this are good ways to achieve anonymity.

Pseudonymity, on the other hand, means there is some characteristic associated with the agent for that transaction, and this characteristic provides some mechanism for recognizing that other transactions also involved this party.

Both anonymity and pseudonymity in this context retain the notion of privacy of location. Location describes the actual physical connection to the communications medium: the speaker is in some sense physically *at* his location. Anonymity is in some sense 'more private' than pseudonymity, because there's less to trace, but having a pseudonym does not necessarily imply that location is public – the pseudonym could well be a reply block on a mixnet, or even simply a keypair which an author uses to sign all of his documents.

2.3.2 Partial Anonymity vs. Full Anonymity

The notion of full anonymity is similar to the use of one-time passwords in encryption: fully anonymous speech means that hearing the speech gives a listener no more information about the identity of the speaker than he had beforehand. In reality, though, every set of candidates is limited in size, and indeed the adversary often has partial information about the suspect – for instance, "he or she has a high-bandwidth Internet connection", or "he or she probably lives in California based on activity patterns and routing analysis."

So in this clearer context, the question shifts from "is it anonymous?" to "is it anonymous enough?" If the original set of suspects has n members, then for sufficiently large n a system which leaks no information that might reduce the set of suspects seems to be "anonymous enough." However, we have to bear in mind that we may well be trying to protect the identities of all the users of a given service – that is, even evidence implying that a given user is one of n users of the service may be sufficient to make him suspicious, thus compromising his anonymity. This may discourage corporations and persons who are particularly concerned about their reputation from participating in a given anonymous service, and indeed it may put ordinary users at risk as well.

It is not even as simple as whether a user is inside or outside the "set of suspects." Often there is no clearly delineated set, and for each user no boolean value of "suspected" or not. A given member of this set might be more suspicious than another member; if the adversary has this knowledge beforehand, then the system can still be fully anonymous if it does not leak any new information to confirm or deny that adversary's initial guesses.

2.3.3 Computational vs. Information-Theoretic Anonymity

One issue to consider is the notion of how protected a given address is: does it rely on computational complexity to protect its anonymity (e.g., a reply block address on a conventional mixnet), or does it use some other technique to make the address unknowable even in the face of a computationally powerful adversary?

There are really two alternatives to computational anonymity. The first alternative is that the adversary has the transcript of the communication, but is still unable to break its anonymity – this is what we call information-theoretic anonymity. This might be modeled by a trusted third party (with trusted channels) acting as moderator, or it might perhaps be implemented in some mechanism similar to a one-time pad.

The second option is that the adversary is simply unable to obtain a transcript of the communication, or perhaps the usefulness of the transcript "expires" quickly enough to be equivalent to no transcript at all. This might happen in the case of some physical medium which is untappable (perhaps quantum channels give us some starting point for this), or in the case of an improved mixnet where a given address no longer maps to or is traceable to its destination after the transaction. This also implies that there is no mechanism for the adversary to take a snapshot of the entire network at that point – if he could, then he might be able to go offline with that snapshot and break the anonymity of the channels.

2.3.4 Perfect Forward Anonymity

Perfect forward secrecy means that after a given transaction is done, there is nothing new that the adversary can "get" to help him decrypt the transcript. Similarly, perfect forward anonymity is the notion that after a given transaction is done, there is nothing new that the adversary can get that can help him identify the location or identity of either of the communicating parties.

In effect, this might be achieved by negotiating a "session location" (the anonymity analog of a session key) between the parties for the purposes of that transaction. For instance, this session location might be a double-blinded virtual location in a high-entropy onion-routed network, where the transaction takes place effectively instantaneously, and then all records of paths to and from the virtual location are destroyed.

In this case, a snapshot could in fact be taken of the system at that point, but this falls under the realm of computational anonymity as described above.

Chapter 3

Attacks

While explaining the motivations behind an anonymous publishing system like Free Haven, we enumerated a number of possible adversaries, diverse in both goals and resources. Several of the types of attacks that may be employed cannot be handled merely through technology: these include social attacks on system security and servnet node operators, political attacks to discourage servnet use, and government and legal attacks to shut down nodes or arrest operators. The success of these attacks will often depend upon the political and jurisdictional clime of servnet nodes' physical location. On the other hand, the success of technical attacks – from individuals, organizations, corporations, or national security agencies – is contingent upon the system's security and robustness to attack.

There are three primary modes of attack: on the communications channel, on the Free Haven servnet, and on individual files. As the security and anonymity of a system is only as strong as its weakest link, we have considered all three of these, and take appropriate countermeasures for many of these attacks.

We have defined ideal anonymity in terms of a communications channel in chapter 2. In doing so, we specified a list of protections to provide for system agents and operations. This section describes the various types of attacks an adversary or group of colluding adversaries might use against the communications channel of Free Haven, relating the effect of these attacks on the level of anonymity maintained.

Adversaries operating on the communications channel may seek to weaken one of the five types of communication anonymity: sender-anonymity, receiver-anonymity, unlinkability between sender and receiver, node-anonymity, and carrier-anonymity. We consider both passive and active adversaries, attacking nodes within the communications channel, internal links between nodes within the channel, and endpoint links from the channel to users (i.e., the sender or receiver, which are both servnet nodes in the case of Free Haven).

3.1 Communications Nodes

The following attacks assume an active adversary that controls one or more nodes within the communications channel:

- **Denial of Service Attack:** An "evil" node within the communications channel can selectively drop messages/packets that it receives at will.

  Prevention: There is basically nothing a system can do to stop a node from behaving in such a manner; however, users can occasionally "ping" various nodes to determine response time. If a node drops a sufficient number of packets and cannot differentiate between ping and data packets, users will come to realize that the node is not reliable and will stop using it.

  Many sources maintain statistical information on the reliability of mixnet nodes, especially Cypherpunk (Type I) and Mixmaster (Type II) remailers [11]. The most common of these networks use only a small number – a dozen or two – public remailers that are known. Naive denial of service attacks would be noticed.

- **Traceroute Collusion Attack:** A corrupt coalition of nodes within the system collude in order to trace certain messages through the communications channel. An "evil" node receives a message, knowing the IP address of both the last-hop and next-hop. Given the ability to collude with a sufficient number of nodes that have received the same message, the path through the communications channel can be traced, as well as ultimately finding the message sender or receiver.

  Prevention: An ideal anonymous communication system will be distributed, as any corrupted central system risks the exposure of both sender and receiver. For a distributed system, a route traversing k nodes preserves anonymity given a maximum of $k-1$ adversaries along this path. This protection also requires that adversaries cannot track a message across one hop, requiring both that the message changes across every node and adversaries cannot perform effective traffic analysis.

- **Cut-the-Channel Collusion Attack:** Similar to the traceroute attack, adversaries need to control a majority of nodes or bottlenecks within the communications channel. If these evil nodes drop packets and perform denial of service on a large scale, an adversary can watch which connections remain, recognizing more easily the normal communications path used between two users.

  Prevention: Similar to other collusion attacks, a distributed system with many independent operators reduces the possibility of a large number of colluding node adversaries. System users can recognize the widespread failure of nodes and stop using the communication channel.

Obviously, this presents a system-wide denial of service attack, affecting the users' overall trust in the system.

- **Traffic Mangling Attack:** To perform a traffic mangling attack, an adversary requires control of nodes at the edges of the communications channel. When an entry node receives a message from a sender, it mangles the message such that transmission or routing will occur properly, but an exit node on the other edge of the communications channel can recognize the mangled message. The colluding nodes can communicate outside of the normal channel, and establish sender-receiver linkability and IP correlation. Therefore, this attack does not require a large control over the system, such as the traceroute collusion attack, to link communicating agents.

  Prevention: There are two main defenses against this type of attack. First, message packets should be encrypted or encoded in such a way that any change by a node – the packet mangling itself – will be detected by other nodes, and the packet discarded. This defense relies on the assumption that not all nodes along the message's path are compromised. Many existing systems (e.g., Onion Routing, Freedom) use symmetric key link-layer encryption to counter a traffic mangling attack. Second, strong partial anonymity is a defense mechanism against this attack. Namely, if an "evil" node cannot determine whether it exists at the true edge of the communications channel (i.e., it cannot tell if the agent from which the message arrives is the initial sender), the node can only reveal some k-anonymous set of possible senders or receivers.

Form-based proxy systems (Anonymizer, LPWA) cannot really be analyzed in terms of these various forms of attack. Indeed, these systems basically rely on a single trusted third party: the proxy itself. If an adversary manages to take control of the proxy, both sender-anonymity and receiver-anonymity are lost, and linkability between the two is also established. Carrier- and node-anonymity are only relevant to distributed systems.

3.2 Communications Channel Links

This section describes a number of attacks that an adversary may perform on links within the communications channel. The adversary's goal is to determine a message's sender or receiver, or to provide linkability between the two agents. Many of these attacks hold the greatest risk to systems attempting to achieve full n-anonymity, where n is the number of system users. In the case of systems that provide partial anonymity, an adversary may gain information from traffic analysis or a similar attack. Yet, the adversary only reveals a k-anonymous set ($k < n$), describing an probabilistic distribution of anonymity, where k identities are below entropy threshold and thus "exposed." Let us consider the following attacks:

- **Computational Attack:** An adversary can sniff a link within the communications channel, and thus be able to read anything that is sent over that link. Presumably, both the data and transmission path are encrypted. To ensure the anonymity of sender and receiver agents, an adversary should not be able to easily determine this transmission path. Various levels of anonymity result from the security of the path's encryption and encoding. Obviously, this attack can also be performed at nodes within the channel or at its edges.

  Prevention: For naming schemes which rely on computationally-secure reply blocks or pseudonyms, an adversary with sufficient computing power can eventually decrypt the name and determine agent identity. If the communications channel relies on partial anonymity, successfully decrypting the name will only reveal a k-anonymous set of possibilities.

- **Message Coding Attack:** An adversary can trace or link a message that does not change its coding during transmission. If links can be passively monitored, the listener can determine the path that a message takes through the communications channel. Obviously, this attack can also be performed by colluding nodes within the channel, but this attack does not require that much penetration into the system.

  Prevention: An end-to-end encoding or encryption scheme is used by the sender, such that the message changes across each link in the communications channel. In a distributed system, a link-to-link scheme is not sufficient to prevent collusion attacks, as messages need to be different at the edges of the channel to protect sender/receiver anonymity.

- **Message Volume Attack:** An adversary can analyze traffic across a link and examine packet size. If a message of the same or similar length is detected traveling through various communications links, a global observer can determine the transmission path and ultimate sender/receiver.

  Prevention: All messages in the system should be of the same size or of a random size. Messages should therefore be padded with random bits, or concatenated with other messages and padded to the specified size.

- **Traceroute Replay Attack:** Within an anonymous communications channel, messages are transmitted to a given reply block or pseudonym of the receiver. An attacker listening on the link can record messages that pass by, and then attempt to forward traceroute the channel by flooding the system with the replayed message. Similarly, if some reply path or nym is given for the sender, the adversary can try to reverse traceroute the message by flooding the sender. The attacker can then perform traffic analysis to detect which links within the channel, or edges of it, see a rise in traffic. This rise in traffic suggests the message's route.

Prevention: To stop forward traceroute replay attacks, a nonce should be included in each message, and nodes should not resend a message that has already been sent. Nodes would be required to store a "graveyard" of used nonces to lookup. Also, an attacker must not be able to change the included nonce. The system can protect against reverse traceroute replay attacks by only including a way to reach the sender if a reply is necessary, as well as making this information only available to the receiver.

- **Intersection Attack:** An adversary may trace user identities by examining usage patterns over a long period. Similar to distinguishing characteristics of a speaker, users may manifest distinguishable behavior. For example, they may exhibit typical on-line/off-line periods, utilize similar resources over time, contact the same destinations or Internet sites regularly, and so on. If any distinguishing characteristics are transmitted (i.e., pseudonyms, Cookies), an adversary can link past and future communications to the specific user with greater certainty.

  Prevention: We cannot determine any protection against this type of attack. Others have questioned if this problem is even solvable [5].

- **Sniping and Cut-the-Internet-Backbone Attacks:** This attack is similar to the cut-the-channel collusion attack between nodes. An adversary has the ability to snipe specific links with the system for selective denial of service, or bring down large segments of the Internet to destroy inter-node links within the communications channel. The attacker can then see which connections of interest remain and perform traffic analysis.

  Prevention: This attack is beyond the resources of many individuals and organizations. A sniping attack might help an adversary perform traffic analysis to some degree; if the attacker has the ability to cut any link at will, they can eventually expose senders and receivers. However, national intelligence agencies are probably the only organizations able and willing to "Cut-the-Internet-Backbone."

3.3 Communications Channel Edges

This section describes a number of attacks that both active or passive adversaries may perform on edges of the communications channel. The edges of the channel correspond to links between a communicating agent (i.e., sender, receiver) and the immediate node within the channel to which it communicates.

- **Timing Attack:** An adversary can attempt to link a specific message between two parties by watching endpoint send and receive actions. Given the would-be transmission time of this message, the attacker can consider the correctness of these two endpoints.

Prevention: A message can only be protected from a timing attack by hiding the message with others. The linkability between sender and receiver can obviously be established if only one message is transmitted within a certain time period. This protection is established by introducing latency, adding dummy messages, or reordering packets.

Unless system load is extremely low, a timing attack is likely to expose linkability only with real-time services. Many systems – such as the original Chaumian mix – are based on `sendmail`, already having quite high and variable transmission time. However, systems designed to allow `telnet`, Web browsing, IRC, and other such services risk linkability from timing attack.

Adding a variable delay decreases the ability to perform timing attacks given a reasonable system load. However, the system has a higher latency than necessary, and the system is still open to traffic analysis attacks, especially if the number of messages across the channel remains small.

Dummy messages can be transmitted instead to remove this unnecessary latency, adding load to the system. If adversaries are given only a possible transmission duration, this solution increases the difficulty of correlating times to specific messages. Dummy messages are an end-to-end solution, whereas link-level garbage can be recognized and thereafter ignored by an adversary that watches the endpoints.

Lastly, communication channel nodes can reorder packets when they are received. Nodes store n packets. Upon receiving more packets, the node chooses some k packets at random from this $(n + k)$ pool and sends them out.

- **Trickle Attack:** An adversary has complete active control of all the edges of the communications channel. The adversary stops all incoming messages from entering the channel except one. The next incoming message is not released until the first message is detected along some edge exiting the channel. As only one message is transmitted through the channel at once, the global observer can establish linkability between sender and receiver.

  Alternatively, an adversary can achieve active control over all the links of some internal node within the channel. This type of attack is more easily attained than system-wide control, and allows attacks as described in section 3.1.

  Prevention: We cannot determine any protection against this attack for communications channels which provide an explicit mapping of names to sender/receiver agents. For systems which provide partial anonymity even after exposure, the "edge" of the communications channel only reveals a k-anonymous set of possible receivers.

- **Identification Flooding Attack:** An adversary can flood with the system with identifiable packets. A message can only remain hidden within the context of other known messages.

During normal operation, each system user would send only one message during each time interval, thus producing an independent set of anonymous messages. However, if an attacker floods the system and fills a node's reorder buffer with n packets, it removes the node's defense against timing attacks and allows a certain message to be more identifiable.

Prevention: A flooding attack is very difficult to defend against for practical Internet systems. Ideally, the system would be able to establish a unique, anonymous identity for each of the n users that send messages during one transmission interval. Performing adequate authentication of message senders while maintaining anonymity is a difficult problem. Possible solutions include the use of pseudonyms for partial anonymous systems, requiring that adversaries cannot control a significant number of pseudonyms. Similarly, some form of blind signature scheme can be used for anonymous authentication.

- **Traffic Flooding Attack:** Similar to the identification flooding attack, an adversary sends a large number of messages into the system to greatly increase traffic along certain paths. The adversary then proceeds to measure a rise in traffic along the communications channel's edges or along internal links. This form of traffic analysis suggests the route taken to reach a specific reply block or pseudonym receiver.

  Prevention: The system should ensure that an equal number of packets are sent between each link during some time interval, by either introducing dummy packets or latency. This protection is similar to that used for timing attacks. However, maintaining steady traffic along the edges of a communications channel is more difficult, given the possibility of very bursty traffic. The system can add dummy packets to the endpoint, but would then require client-side filtering.

- **Pseudonymity Marking Attack:** An adversary masquerades as a normal user and distributes unique names to other distinct agents in the system. These unique names correspond to different reply blocks or pseudonyms, such that the last hop of the transmission path is different for each name. The adversary can then correlate the last hop of the received message to a specific sender. Linking a sender agent to an individual user or IP address remains a separate problem, unless this can be determined during name distribution.

  Prevention: This attack is only viable against a communications channel in which the receiver has an explicit transmission path, such as with remailer reply blocks, as opposed to multi-cast or random-walk functionality. Secondly, a sender can defend against this attack by getting the receiver's name from a third party – such as another trusted agent or some specified meeting-place for name distribution.

- **Persistent Identity Attack:** The persistent identity attack is not an attack per se, but rather an inherent anonymity weakness with any persistent naming infrastructure. If an adversary manages to disclose user information with any of the attacks we have enumerated, the adversary can correlate any future use of this name with the exposed individual.

Prevention: System agents – senders and receivers – should use dynamic naming to provide forward anonymity, or rely on a partial anonymity scheme such that disclosure of agents only reveals a k-anonymous set of possibilities.

- **The Need for "Recipient-Hiding" Public Key Encryption:**[1] We call a public key cryptosystem *recipient-hiding* if it is infeasible to determine, given a ciphertext, the public key used to create that ciphertext. The recipient-hiding property is *not* implied by the standard definition of semantic security (even with respect to adaptive chosen ciphertext attack). Moreover, it is not even achieved in common practical constructions. This has implications for mixnets which use reply blocks that are separate from the body of the message.

  To see that semantic security and recipient hiding are independent, consider any semantically secure cryptosystem C. Construct the cryptosystem C' which is just like C in every way, except that it appends the public key used to encrypt to every ciphertext. All messages produced by C' with the same public key are indistinguishable from each other if the messages produced by C are, and so C' is semantically secure – but it is the very opposite of recipient hiding.

  In practice, mail programs such as PGP tend to include the recipient's identity in their header information. Even if headers are stripped, David Hopwood has pointed out in the case of RSA that because different RSA public keys have different moduli, a stream of ciphertext taken modulo the "wrong" modulus will tend to have a distribution markedly different from the same stream taken modulo the "right" modulus. This allows an adversary to search through a set of possible public keys to find the one which is the best fit for any ciphertext, even if OAEP or similar padding is used.

  In mixnets which provide reply blocks, the reply block is often treated as opaque(for example Babel [15]) and prepended to the message to be sent. This means that the message is available for inspection by each intermediate hop with no processing at each hop; for this reason the message is often encrypted with the public key of the recipient. The point of a reply block is to provide a chain of mix nodes between the sender and the recipient, in which intermediate nodes are supposed to know neither the sender nor the receiver. If the recipient can be identified by simply inspecting the message, then every single intermediate node knows the destination, and knows approximately where it is in the reply block. This may leak an undesirable amount of information about the mixnet.

[1]This attack against certain cryptosystems was suggested by David Molnar, and the resulting "recipient-hiding" requirement and discussion was written by him.

Prevention: Rivest suggested that randomized cryptosystems (such as the Goldwasser-Micali cryptosystem) might possess this property. Independently, Lysyanskaya and Wagner proposed a version of ElGamal in which all parties share the same modulus as a concrete example. There is a formal definition of recipient-hiding proposed on sci.crypt by Hopwood [17], with application to showing that a variant of Bellare, Abdalla, and Rogaway's DHAES scheme [1] achieves the recipient-hiding property, along with a variant of RSA in which moduli are generated to be close together. It seems that recipient-hiding cryptosystems may not be that hard to construct, once the requirement is recognized. The problem is that because previous systems were not designed with anonymity in mind, commonly deployed cryptosystems may not be recipient-hiding.

Chapter 4

Related and Alternate Works

Several approaches have been proposed to achieve anonymity on an Internet communications channel. This section describes several real-world projects, which we analyze in terms of anonymity provided and resistance to various types of attack. We include a more in-depth review of anonymous communications channels in appendix B, truncated here for length and readability.

4.1 Proxy Servers: Anonymizer

Proxy services provide one of the most basic forms of anonymity, inserting a third party between the sender and recipient of a given message. Proxy services are characterized as having only one centralized layer of separation between message sender and recipient. The proxy serves as a "trusted third party," responsible for removing distinguishing information from sender requests.

The Anonymizer was one of the first examples of a *form-based web proxy* [3]. Users point their browsers at the Anonymizer page at `www.anonymizer.com`. Once there, they enter their destination URL into a form displayed on that page. The Anonymizer then acts as an `http` proxy for these users, stripping off all identifying information from `http` requests and forwarding them on to the destination URL.

The functionality is limited. Only `http` requests are proxied, and the Anonymizer does not handle cgi scripts. In addition, unless the user chains several proxies together, he or she may be vulnerable to an adversary which tries to correlate incoming and outgoing `http` requests. Only the data stream is anonymized, not the connection itself. Therefore, the proxy does not prevent traffic analysis attacks like tracking data as it moves through the network.

Proxies only provide unlinkability between sender and receiver, given that the proxy itself remains uncompromised. This unlinkability does not have the quality of perfect forward anonymity, as proxy users often connect from the same IP address. Therefore, any future information used to gain linkability between sender and receiver (i.e., intersection attacks, traffic analysis) can be used against previously recorded communications.

Sender and receiver anonymity is lost to an adversary that may monitor incoming traffic to the proxy. While the actual contents of the message might still be computationally secure via encryption, the adversary can correlate the message to a sender/receiver agent.

This loss of sender/receiver anonymity plagues all systems which include external clients which interact through a separate communications channel – that is, we can define some distinct edge of the channel. If an adversary can monitor this edge link or the first-hop node within the channel, this observer gains agent-message correlation. Obviously, the ability to monitor this link or node depends on the adversary's resources and the number of links and nodes which exist. In a proxy system, this number is small. In a globally-distributed mixnet, this number could be very large. The adversary's ability also depends on her focus: whether she is observing messages and agents at random, or if she is monitored specific senders/receivers on purpose.

4.2 Mixmaster Remailer

The pursuit of anonymity on the Internet was kicked off by David Chaum in 1981 with a paper in Communications of the ACM describing a system called a "Mix-net" [7]. This system uses a very simple technique to provide anonymity: a sender and receiver are linked by a chain of servers called Mixes. Each Mix in the chain strips off the identifying marks on incoming messages and then sends the message to the next Mix, based on routing instructions which are encrypted with its public key. Comparatively simple to understand and implement, this Mix-net (or "mix-net" or "mixnet") design is used in almost all of today's practical anonymous channels.

Until the rise of proxy-based and TCP/IP-based systems, the most popular form of anonymous communication was the *anonymous remailer*: a form of mix which works for e-mail sent over SMTP. We describe the development of remailer systems in greater depth in appendix B; in short, the evolution of remailers has led to the Mixmaster Type II remailer, designed by Lance Cottrell [11].

Each Mixmaster remailer has an RSA public key and uses a hybrid symmetric-key encryption system. Every message is encrypted with a separate 3DES key for each mix node in a chain between the sender and receiver; these 3DES keys are in turn encrypted with the public keys of each mix node. All Mixmaster messages are padded to the same length.

When a message reaches a mix node, it decrypts the header, decrypts the body of the message, and then places the message in a "message pool." Once enough messages have been placed in the pool, the node picks a random message to forward. As the underlying next-hop header in the message has been decrypted, the node knows to which destination to send this message. In this way a chain of remailers can be built, such that the first remailer in the chain knows the sender, the last remailer knows the recipient, and the middle remailers know neither.

Remailers also allow for *reply blocks.* These consist of a series of routing instructions for a chain of remailers which define a route through the remailer net to an address. Reply blocks allow users to create and maintain pseudonyms which receive e-mail. By prepending the reply block to a message and sending the two together to the first remailer in the chain, a message can be sent to a party without knowing his or her real e-mail address.

4.3 Rewebber

Goldberg and Wagner applied Mixes to the task of designing an anonymous publishing network called Rewebber[14]. Rewebber uses URLs which contain the name of a Rewebber server and a packet of encrypted information. When typed into a web browser, the URL sends the browser to the Rewebber server, which decrypts the associated packet to find the address of either another Rewebber server or a legitimate web site. In this way, web sites can publish content without revealing their location.

Mapping between intelligible names and Rewebber URLs is performed by a name server called the Temporary Autonomous Zone (TAZ), named after a novel by Hakim Bey. The point of the "Temporary" in the name of the nameserver (and the novel) is that static structures are vulnerable to attack. Continually refreshing the Rewebber URL makes it harder for an adversary to gain information about the server to which it refers.

4.4 Onion Routing

The Onion Routing system designed by Syverson, et. al. creates a mix-net for TCP/IP connections [40, 32]. In the Onion Routing system, a mixnet packet, or "onion", is created by successively encrypting a packet with the public keys of several mix servers, or "onion routers."

When a user places a message into the system, an "onion proxy" determines a route through the anonymous network and onion encrypts the message accordingly. Each onion router which receives the message peels the topmost layer, as normal, then adds some key seed material to be used to generate keys for the anonymous communication. As usual, the changing nature of the onion – the "peeling" process – stops message coding attacks. Onions are numbered and have expire times, to stop replay attacks. Onion routers maintain network topology by communicating with neighbors, using this information to initially build routes when messages are funneled into the system. By this process, routers also establish shared DES keys for link encryption.

The routing is performed on the application layer of onion proxies, the path between proxies dependent upon the underlying IP network. Therefore, this type of system is comparable to loose source routing.

Onion Routing is mainly used for sender-anonymous communications with non-anonymous receivers. Users may wish to Web browse, send email, or use applications such as `rlogin`. In most of these real-time applications, the user supplies the destination hostname/port or IP address/port. Therefore, this system only provides receiver-anonymity from a third-party, not from the sender.

Furthermore, Onion Routing makes no attempt to stop timing attacks using traffic analysis at the network endpoints. They assume that the routing infrastructure is uniformly busy, thus making passive intra-network timing difficult. However, the network might not be statistically uniformly busy, and attackers can tell if two parties are communicating via increased traffic at their respective endpoints. This endpoint-linkable timing attack remains a difficulty for all low-latency networks.

4.5 ZKS Freedom

Recently, the Canadian company Zero Knowledge Systems has begun the process of building the first mix-net operated for profit, known as Freedom [42]. They have deployed two major systems, one for e-mail and another for TCP/IP. The e-mail system is broadly similar to Mixmaster, and the TCP/IP system similar to Onion Routing.

ZKS's "Freedom 1.0" application is designed to allow users to use a nym to anonymously access web pages, use IRC, etc. The anonymity comes from two aspects: first of all, ZKS maintains what it calls the Freedom Network, which is a series of nodes which route traffic amongst themselves in order to hide the origin and destination of packets, using the normal layered encryption mixnet mechanism. All packets are of the same size. The second aspect of anonymity comes from the fact that clients purchase "tokens" from ZKS, and exchange these token for nyms – supposedly even ZKS isn't able to correlate identities with their use of their nyms.

The Freedom Network looks like it does a good job of actually demonstrating an anonymous mixnet that functions in real-time. The system differs from Onion Routing in several ways.

First of all, the system maintains Network Information Query and Status Servers, which are databases which provide network topology, status, and ratings information. Nodes also query the key servers every hour to maintain fresh public keys for other nodes, then undergo authenticated Diffie-Hellman key exchange to allow link encryption. This system differs from online inter-node querying that occurs with Onion Routing. Combined with centralized nym servers, time synchronization, and key update/query servers, the Freedom Network is not fully decentralized [12].

Second, the system does not assume uniform traffic distribution, but instead uses a basic "heartbeat" function that limits the amount of inter-node communication. Link padding, cover traffic, and a more robust traffic-shaping algorithm have been planned and discussed, but are currently disabled due to engineering difficulty and load on the servers. ZKS recognizes that statistical traffic analysis is possible [39].

Third, Freedom loses anonymity for the primary reason that it is a commercial network operated for profit. Users must purchase the nyms used in pseudonymous communications. Purchasing is performed out-of-band via an online Web store, through credit-card or cash payments. ZKS uses a protocol of issuing serial numbers, which are reclaimed for nym tokens, which in turn are used to anonymously purchase nyms. However, this system relies on “trusted third party” security: the user must trust that ZKS is not logging IP information or recording serial–token exchanges that would allow them to correlate nyms to users [38].

4.6 Web Mixes

Another more recent effort to apply a Mix network to web browsing is due to Federrath et. al.[6] who call their system, appropriately enough, “Web Mixes.” From Chaum’s mix model, similar to other real-time systems, they use: layered public-key encryption, prevention of replay, constant message length within a certain time period, and reordering outgoing messages.

The Web Mixes system incorporates several new concepts. First, they use an adaptive “chop-and-slice” algorithm that adjusts the length used for all messages between time periods according to the amount of network traffic. Second, dummy messages are sent from user clients as long as the clients are connected to the Mix network. This cover traffic makes it harder for an adversary to perform traffic analysis and determine when a user sends an anonymous message, although the adversary can still tell when a client is connected to the mixnet. Third, Web Mixes attempt to restrict insider and outsider flooding attacks by limited either available bandwidth or the number of used time slices for each user. To do this, users are issued a set number of blind signature tickets for each time slice, which are spent to send anonymous messages. Lastly, this effort includes an attempt to build a statistical model which characterizes the knowledge of an adversary attempting to perform traffic analysis.

4.7 Crowds

The Crowds system was proposed and implemented by Michael Reiter and Avi Rubin at T&T Research, and named for collections of users that are used to achieve partial anonymity for Web browsing [36]. A user initially joins some crowd and her system begins acting as a node, or anonymous *jondo*, within that crowd. In order to instantiate communications, the user creates some path through the crowd by a random-walk of *jondos*, in which each *jondo* has some small probability of sending the actual `http` request to the end server. A symmetric key is shared amongst these path *jondos* to ensure link-encryption within a crowd. Once established, this path remains static as long as the user remains a member of that crowd. The Crowds system does not use dynamic path creation so that colluding crowd eavesdroppers are not able to probabilistically determine the initiator (i.e., the actual sender) of requests, given repeated requests through a crowd. The *jondos* in a given path also share a secret *path key*, such that local listeners, not part of the path, only see an encrypted end server address until the request is finally sent off. The Crowds system also includes some optimizations to handle timing attacks against repeated requests, as certain HTML tags cause browsers to automatically issue re-requests.

Similar to other real-time anonymous communication channels (Onion Routing, the Freedom Network, Web Mixes), Crowds is used for senders to communicate with a known destination. The system attempts to achieve sender-anonymity from the receiver and a third-party adversary. Receiver-anonymity is only meant to be protected from adversaries, not from the sender herself.

The Crowds system serves primarily to achieve sender and receiver anonymity from an attacker, not provide unlinkability between the two agents. Due to high availability of data – real-time access is faster that mix-nets as Crowds does not use public key encryption – an adversary can more easily use traffic analysis or timing attacks. However, Crowds differs from all other systems we have discussed, as users are *members* of the communications channel, rather than merely communicating *through* it. Sender-anonymity is still lost to a local eavesdropper that can observe all communications to and from a node. However, other colluding *jondos* along the sender's path – even the first-hop – cannot expose the sender as originated the message. Reiter and Rubin show that as the number of crowd members goes to infinity, the probable innocence of the last-hop being the sender approaches one.

Chapter 5

Communications Module: Design and Implementation

This section details the design and implementation of a communications module for Free Haven, serving as an interface between the anonymous publication system and its corresponding anonymous communications channel(s).

5.1 Communictions in Free Haven

The Free Haven design is based on a community of servers – termed the "servnet" – where each server hosts data from the other servers in exchange for the opportunity to store data of its own in the servnet. Inherent to the anonymity and reliability of the system are several design requirements.

- The system must provide a mechanism for anonymously **inserting** a document into the servnet.
- The system must provide a mechanism for anonymously **retrieving** a document from the servnet.
- The system must provide a mechanism for smoothly **adding servers**.
- The system must provide a mechanism for **expiring** documents.
- The system must provide a mechanism for **recognizing inactive or dead servers** and eventually no longer using or querying them.

The first three requirements of Free Haven depend heavily on an anonymous communications channel. Such a channel is necessary to allow the anonymous insertion and retrieval of files in the servnet, as well as introducing new servers to existing servnet nodes. Any broadcasts and point-to-point communications between servnet nodes need to be performed anonymously to maintain privacy of a node's identity.

5.2 Module Design

An anonymous communications channel is used to carry messages between servnet nodes. In order to provide an abstraction for the specifics of our communication channel, we provide a communication module that interfaces with both the outer anonymous channel and "inner" *haven* module, which provides the actual document publishing, storage, and retrieval functionality.

In our design summary in section 1.3, we enumerated a number of our required operations – sending messages, broadcasting messages, naming nodes for message delivery, adding nodes, and removing nodes – and desired goals – low latency, delivery and routing robustness, resistant to attack, and decentralized. While many of these operations and goals are inherently part of the communications channel, the module should provide a flexible and robust interface layer between the two subsystems.

5.2.1 Supported Module Operations

Specifically, the *comm* module to support the following operations or characteristics:

- The *comm* module should send data to *haven* when it becomes available from the communications channel.
- The *comm* module should send data to the communications channel when it becomes available from the *haven* module.
- Messages can arrive simultaneously from the communications channel.
- Messages can arrive from the *haven* module simultaneously as messages arrive from the communications channel.
- Messages on a socket can arrive in a delayed manner.
- The *haven* module can *fail* unexpectedly, but *comm* should still continue normal operation, and reconnect to *haven* when it becomes available again.

5.2.2 Module Data Structures

The communications module has several associated data structures:

- **Incoming Feeder Queue:** List of $\{socket, filedesc, filename\}$ tuples that correspond to feeder programs currently being processed by the communications module.
- **Haven Message Queue:** List of messages that have arrived from the communications channel, awaiting transmission to the haven module.
- **Node Database:** Database of information stored for all-known servnet nodes.
  Key: Hash of public key
  Value:
  - **Public key**
  - **Waiting message queue:** List of filenames queued for each node in the database, corresponding to outgoing messages from the haven module.
  - **Length of queue**
  - **Cost:** Total "weight" of waiting messages in queue
  - **Communications channel type:** Mixmaster, ZKS, etc.
  - **Address / routing info:** Reply blocks, pseudonyms, etc.
  - **Statistics:** Timeout and availability considerations

5.2.3 System Modularity

The Free Haven design stresses modularity between various pieces of the system – The *haven* module, the *comm* module, the *trust* module, the *crypto* module, the *ui*, and the actual communications channel – providing a strong separation of function. While the initial proof-of-concept implementation of Free Haven will assume a 1:1 *haven* to *comm* module relation, with both processes running on the same machine, this relationship is not necessarily fixed. Indeed, with only slight modification to the actual socket code, our design allows for several *haven* process to share a single network *comm* process interfacing with different communications channels. In order to support this scenario, we should not assume a secure connection across the *comm–haven* socket. Instead, all data across this socket uses both public-key encryption for security and base-64 encoding within data blocks to ensure proper user-formatting of information.

5.3 Module Implementation

In Chapter 2 we discussed the anonymity offered by various anonymous channels; in Chapter 6 we will suggest some other possible channel designs. The current implementation, however, utilizes the Mixmaster remailer [11]. Mixmaster was chosen because of its simple command-line interface, strong anonymity offered through Chaumian mixing, protection from traffic analysis and other attacks due to its high latency, message padding, and packet reordering and buffering within nodes. Furthermore, Mixmaster is freely available, making it quite suitable for an open project such as Free Haven.

The code base allows for the easy incorporation of different mixnets and other anonymous communications channels, by adding simple switching logic to the `do_transmit` function and specifying the proper mixnet type and address in the node database. Indeed, the only primitive that we require is a generalized `send` function. Therefore, a servnet node can actually specify the communications channel on which it wishes to communicate.

The *comm* module implementation also provides database flexibility. We provide an abstraction layer on top of the Node DB, separating any database-specific operations from the user. The GPL-released `gdbm` is currently used in the initial implementation; a relational database of greater functionality is being considered for further development.

5.3.1 Implementation Pseudocode

The communications module provides an "always-on" module to handle incoming and outgoing messages from a servnet node. The basic operation of the module is as follows:

- Create a non-blocking listening tcp socket for incoming communications for feeder programs. These *feeders* are system process that will pass messages from the communications channel to the Free Haven communications module.

- Loop on following control structure:

  1. Connect to haven socket. If this socket is not available, continue processing available information and try again upon next iteration.
  2. If data available on *haven* socket, process the outgoing message.
     (a) If message is of type `broadcast`, enumerate the list of servnet nodes and `transmit` the message to each node.
     (b) If message is of type `transmit`, enqueue the message within the node's waiting message queue for later processing.
     (c) If message is of type `introduce`, add the corresponding information about a new node to the node database.
  3. If new *feeder* attempts to connect to non-blocking incoming socket, enqueue the new *feeder*.
  4. For all *feeder* sockets on which data is available:
     (a) Read all the information from the socket into the *feeder's* corresponding file.

/* Communications Handling Functions */
void handle_ports(int incoming_port, int listen_socket, int haven_port);
void create_feeder_entry(int new_feeder_socket);
void process_feeder_entry(struct feeder_t *feeder, struct feeder_t *prev_feeder);
void enqueue_haven_message(char *filename);
void process_freehaven_message(char *filename);
void process_internal_message(char *filename);
void send_file_to_haven(char *filename);
void exit_comm();
/* Transaction Functions */
int transmit(struct tag_t *tag_list);
int do_transmit(char *PK);
int broadcast(struct tag_t *tag_list);
/* Node Database Functions */
void initialize_nodedb(void);
datum construct_gdbm_entry(struct nodedb_entry_t *entry);
struct nodedb_entry_t reconstruct_nodedb_entry(void *noded_value);
int node_change_PK(char *hPK, char *newPK);
int node_add_new_node(char *hPK, char *PK, char *address, char *mixnet, int statistics);
struct message_queue_t* node_get_message_queue(char *hPK);
char* node_get_PK(char *hPK);
char* node_get_mixnet(char *hPK);
char* node_get_address(char *hPK);
char* node_get_busiest_PK();
int node_add_waiting_msg(char *hPK, char *filename);
int node_get_waiting_msgs(char *hPK);
int node_set_statistics(char *hPK, int statistics);
int node_get_statistics(char *hPK);
void free_node_msgs(struct nodedb_entry_t nodedb_entry);
void close_nodedb();

Table 5.1: Communications Module API

   (b) If the *feeder* has reached the file EOF, place the file into the haven message queue and close the *feeder*.

5. Send incoming messages to the haven module, extracting from haven message queue.

6. Transmit waiting messages for a node, concatenating and padding to reach a total packet size up to a random or statically-assigned length. Any messages that cannot fit within this buffer remain in the waiting messages queue for later transmission.

5.3.2 Design Discussion

The communications module itself requires access to the crypto module. Communications between the *haven* and *comm* modules should be ASCII-armored with base-64 encoding, as the messages use standard XML format, with data delimited by begin $< tag >$ and end $< \backslash tag >$. Using base-64 encoding stops an user from placing $<$ or $>$ symbols into the internal data block, which would confuse message parsing. Furthermore, communications between the *haven* and *comm* are encrypted, to provide for the option of distributed module processes and multiple *haven* processes per *comm* process.

The communications module also needs to perform any necessary message encryption or encoding necessary, based on the type of communications channel used. For the current Mixmaster implementation, the *comm* process performs layered encryption of the message data. This layered encryption method – or "onion routing" – is the standard Chaumian mix-net technique to ensure anonymity. First, the sender chooses a route through the mixnet: Source S $\rightarrow$ Node A $\rightarrow$ Node B $\rightarrow$ Node C $\rightarrow$ Destination D pseudonym. Second, the sender signs the message with its private key pk_S and possibly encrypts via the destination's public key PK_D. Then, the sender encrypts the message and next-hop information along the mixnet in reverse: encrypt with PK_C, then encrypt with PK_B, and finally encrypt with PK_A. Therefore, only the proper node in the mixnet can decrypt the top layer of the message, exposing next-hop information and the proper onion message to relay. In other words, the encryption layers are unwound (or peeled like an onion) during mixnet transmission, before the message is relayed to its destination. The destination reply-block or pseudonym specifies a path to an explicit servnet node. Once the message arrives at the destination servnet node, the node decrypts the message using pk_D and verifies the sender's signature.

5.3.3 Optimizations

The communications module performs several efficiency and anonymity optimizations. First, haven message queues are built up within the *comm* process, and iteratively dequeued and sent to *haven*, protecting against bursty transfers from the communications channel.

Second, broadcasts can be requested by the haven module by merely calling a `broadcast` primitive, as opposed to querying for a list of available nodes, extracting each node's route and encrypting the message according to that route, then performing the `transmit` operation on each message. A single broadcast call decreases the quantity of socket-level requests from $O(n)$ to $O(1)$, where n is the number of available nodes.

Third, messages are queued during `transmit` for each node. One node is chosen at random (while attempting to ensure semi-fairness among nodes) to transmit its messages along to anonymous communications channel. Messages are concatenated and padded to a specific or random length to protect against message volume attacks. The act of enqueueing and concatenating messages reduces the number of messages to a specific node, possibly adding some protection against traffic analysis that seeks sender/receiver linkability. Furthermore, the random choice of a node during that time iteration adds some latency to haven outgoing messages, especially under high load. This technique may also protect against some form of traffic analysis, especially in light of Free Haven multi-step protocols, such as trading or buddy “squawking.”

Chapter 6

Future Work

A "standard" design for an anonymous communications channel is very much an open question. In previous chapters, we defined our notions of anonymity and considered how current works fulfill these requirements. In general, there are various considerations when designing an anonymous communications channel:

- Low latency
- Delivery robustness
- Resistance to traffic analysis and similar attacks
- Types of anonymity provided:
  - Anonymity vs. pseudonymity
  - Full vs. partial anonymity
  - Computational vs. information-theoretic anonymity
  - Perfect forward anonymity

Some of these goals are conflicting in nature. Systems which provide low latency and strong delivery robustness are generally more open to traffic analysis and other such attacks, as messages are routed quickly and sometimes repetitively through the channel. Still, while Free Haven stresses anonymity over availability, we would prefer a design which provides latency in the realm of seconds or minutes, as opposed to hours or days. If a high latency was to endure, Free Haven usage would be constrained to publishers and readers which specifically require our strong notions of anonymity. Similarly, lossyness within the channel degrades system performance. Free Haven can be designed to handle communications lossyness for file reconstruction by a robust information dispersal algorithm. Still, this condition has possible effects on the trust network and trading/buddy protocols, when the loss is not due to server failure, but rather to an unreliable communications channel. In this section, we present some alternative designs for an anonymous communications channel.

6.1 Garlic Routing

The concept behind garlic routing is based on the underlying mix-net design. A sender encodes routing information in a series of layered encryptions, forming an "onion" of encrypted information. Each node along the route decrypts the outer layer of the onion, exposing the next layer and determining the location of the next hop. Eventually, the entire onion is peeled the the message reaches its destination. A garlic packet looks similar to an onion packet, until it is unwrapped. A node then finds several garlic bulbs to transmit, instead of the normal single onion. Each bulb is a viable path-to-destination from that intermediate node, therefore providing several routes. Earlier intermediate nodes would have no knowledge of the path or existence of these newly exposed routes.

Garlic routing provides a few benefits. Delivery reliability and robustness is increased through path redundancy. Reply-block encoding can be implemented efficiently in terms of size, as reply blocks will only grow linearly with the total number of nodes in onion and garlic routes. The encryption of header information can be performed using a hybrid scheme: all garlic bulbs within a layer are encrypted with the same symmetric key, which is then encrypted with each of garlic node's public key. Therefore, the size of a garlic packet containing n bulbs is only $L + l * n$, where L is the size of the normal onion layer and l is the symmetric key length. A similar hybrid encryption scheme is used by PGP.

Based on the concept of a Chaumian mix-net, a garlic-routing mix-net provides computational anonymity based on the strength of encryption used on the garlic. Reply-blocks for the channel provide a route to an explicit destination, thus disclosure of the information will specify a receiver. This differs from our definition of partial anonymity schemes, where exposure of information will only yield some k-anonymous set of possibilities.

6.2 Iterative Exposure

The concept of iterative exposure is similar to that of a mix-net: each path node only knows the last hop from where it received the packet and can only expose the next hop to where it should send the packet. However, we do not use an layered encryption scheme. Header information is a list of entries, each one encrypted to a specific node, containing a simple message such as "Node A: send to node B." As with garlic routing, we can help provide delivery robustness through redundancy with a simple change: "Node A: send to nodes B, C, D." Entries in the routing list are randomly ordered. A node iterates through the list, and attempts to decrypt each entry and check if the entry corresponds to itself. Once determining the proper next hop information, the node can reorder the routing list to protect against message coding attacks. To adequately protect against this type of attack, some further change to the message itself would be required at each hop.

A destination can either provide a reply-block for a specific source, in which the source node has an encrypted element in the list, or a generic public reply-block. For the latter, one element needs to be public-ally readable so that a source knows where to "pick up" the path: a source can either send a message to this node as the first hop, or the source can provide its only anonymous path to this node by adding reply list entries. Admittedly, this node will lose carrier-anonymity to adversaries. However, such a generic reply block can protect against pseudonymity marking attacks, and it allows both the sender and receiver to specify paths accordingly to their own anonymity requirements.

6.3 Adaptive Control Net

The AC-Net seeks to make computational and traffic analysis attacks more difficult by specifying a naming scheme that does not yield an explicit destination. The advantage of this scheme is predicated on the benefits of partial anonymity: an adversary can only determine a k-anonymous set of destination possibilities.[1]

6.3.1 Joining an Existing Net

A node generates a series of n identical tokens, each of b randomly-chosen bits. This sequence of tokens is the node's address, and is kept private. To join an existing network, a node connects to a few existing public nodes and offers to trade one token with each of its neighbors. The node removes the token traded away from its address and replaces it with the neighbor's token. The node also adds the token traded away and the one received as its approximation of the neighbor's address.

6.3.2 Node Behavior

Nodes will occasionally receive messages, most likely in the form of encrypted data. Whether or not the node can read the data, it should always forward it properly to neighboring nodes to protect against traffic analysis of channel edges.

At the top of the message will be a series of tokens. A node compares these tokens to the approximation of the address of each neighbor, calculating the number of tokens in common with each. Of those which have the greatest commonality, the node picks some subset of these nodes at random, and sends them the message.

[1]The design of the AC-Net was initially proposed and described by Brian Sniffen.

6.3.3 Partial-Anonymous Naming of Nodes

In order for the protocol to work, tokens need to continue to spread throughout the network. Thus, each node should offer a trade to each of its neighbors within some user-set delay. The token chosen to trade is selected at random from the node's address, so it may not be one of the node's own tokens.

Eventually, the above protocol would lead to a completely homogeneous set of tokens. This would make message delivery problematic. Thus each node has a second user-set delay between mutations. When that delay has passed, the node replaces portions of its address with its own tokens.

A node publicizes its address by exposing some $\log n$ of its address tokens. A sender address a message to this $\log n$ address. Therefore, a node's public address yields partial anonymity: many such combinations result from the node's actual address, and many addresses can yield this set of $\log n$ address tokens. As destinations forward their own messages as well, an adversary cannot determine the actual destination only given the message's destination address. Obviously, messages are given a TTL (time-to-live).

6.3.4 Network Topology

This protocol relies on token gradients across the network. The message travels from regions of low potential to regions of high potential. For this to happen, the network probably requires a properly-connected graph. If nodes connect to large numbers of other public nodes, or edges between nodes are randomly distributed, the resulting network may not have clear token gradients. We have not yet determined what type of network topology is actually required for adequate message delivery, nor considered protocols to yield a desirable topology via distributed communications and control.

6.4 Zones

The Crowds system organizes users into collections of nodes called crowds, achieving partial-anonymity for non-local adversaries and sender-anonymity within each crowd. Crowds, as proposed by AT&T Research, focuses on anonymous Web browsing, in which users communicate with specific end servers. Built onto the Crowds model, the Zones system seeks to achieve anonymous communications between two users instead.

Communications from a sender are routed through the sender's zone as specified by the Crowds protocol. When the message is dispatched from a crowd node, however, it is not sent directly to an end server. Instead, the message is dispatched to another zone. Once within the destination zone, the message is either multi-casted to all nodes or forwarded around via some random walk. Each node attempts to decrypt each message received to determine if it is the proper receiver. If randomwalking is used, the proper receiver still wants to forward the message to protect against zone traffic analysis. This being so, a message's TTL has to be sufficiently high such that a random-walk will probabilistically find the proper receiver. This type of distribution strategy provides k-anonymous receiver-anonyminity, where k is the size of the receiver's zone.

6.5 Small Worlds Model

Social networks display two characteristics which initially may appear to be contradictory. First, social connections display clustering, whereby friends are likely to share the same group of friends. Second, they exhibit what has been termed by Stanley Milgram as the "small worlds effect" [27]. Namely, any two people can establish contact by going through only a short chain of intermediate acquaintances. Milgram proposed that all people in the world are separated by six intermediataries on average; this effect is better known as "Six Degrees of Separation" or the "Kevin Bacon Game."

6.5.1 Network Construction and Transmission

Mathematicians have begun studying sparse networks to prove the small worlds effect [41]. Using these models for network topology, we can construct an anonymous communications channel built on this network routing principle. Crowds or zones mimic small worlds of friends, without the necessary long-range connections that provide the small worlds effect. Therefore, we can achieve this effect by constructing zones whereby people also specify a few connections to users in other zones.

A node sends a message to all of its friends, who in turn propagate the message to the rest of their friends. While friends share many connections due the clustering effect – an explicitly formed by the creation of zones – the long-range connections allow the propagation of messages through the network. This model places a large load on the system, especially given the high degree of connectivity in most networks. We have also considered an alternative "directed" propagation of messages accordingly to some greedy heuristic, such as a Hamming distance or the protocol we describe in the Adaptive Control Net.

6.5.2 Network Anonymity

The anonymity of this communications channel is based on the presumed innocence of the Crowds system. An adversary within the network cannot determine whether a message received originated from its last hop, or was merely forwarded on by that node. As the number of nodes that may be involved in the message's path increases, the innocence of the last hop becomes probabilistically greater [36]. The system does not protect the anonymity of senders from a global observer.

K-anonymous receiver-anonymity can be achieved if receivers forward the message along. For simple broadcast networks, k is equal to the number of nodes in the entire network; for directed multi-cast networks, k is the number of nodes that received the message. With large values of k and a high propagation of message transmissions, adversaries have a difficult time in performing traffic analysis.

Messages should be encrypted to the receiver such that only the receiver can tell if the message is meant for her, such that an adversary cannot simply compare some naming scheme to link messages to receivers. A small worlds model likely requires a low-latency communications channel, such as that built directly on TCP/IP, due to the high bandwidth requires. Delivery robustness is assured also by the redundancy of paths to any destination. To protect against message coding and message volume attacks, messages can be point-to-point encrypted and padded to the same or some random size. Reordering messages within a given node helps protect against timing attacks similar to mix-nets. Like other schemes, we still witness the trade-off between a low latency channel and a resistance to traffic analysis.

Gnutella uses a similar six-degrees of separation model for network communication. Nodes broadcast a received message to all their friends, and messages include an explicit TTL to stop infinite looping. Messages are not encrypted or padded.

Chapter 7

Conclusions

The Free Haven project seeks to deploy a system of servers that allows for the anonymous publishing, storage, and retrieval of documents. Stronger anonymity is due to both communications and file storage protocols. This paper discusses the design and implementation of a communications module to interface between the Free Haven publishing system and the communications channel. This work is being performed collaboratively with developers of the trust system, the haven module for share storage, distribution, and retrieval, and the user interface. The ultimate completion of these modules will provide adequate functionality for an initial Free Haven release.

This paper addresses anonymity considerations for the selection of a proper communications channel. While our motivation is the design and implementation of the Free Haven system, the analysis of anonymity and protections from attack in existing communications systems has even wider appeal. Modeling the problem as a multi-tiered system – where anonymity depends on higher publishing layers but rests on the underlying communications channel – allows us to describe attacks the system more clearly, as well as to define what it means to "break" the anonymity of the system.

Note that the difficulty of designing a good anonymous communications channel is not limited to the difficulty of providing strong anonymity for the system. Indeed, there are a number of other qualities which the designer might desire, such as high availability or robustness of data, support for real-time protocols such as Web browsing or `telnet`-like access,or speed or efficiency of implementation or operation. Often the level of anonymity a system provides is the direct result of a tradeoff with the level of availability or flexibility of the system. By defining characteristics of anonymity which the ideal anonymous communications channel might achieve and enumerating the types of attacks that may be used against the system, we hope to provide a basis for better system design decisions.

Bibliography

[1] M. Abdalla, M. Bellare, and P. Rogaway. DHAES. Submission to IEEE P1363.

[2] Masayuki Abe. Universally verifiable mix-net with verification work independent of the number of servers. In *Advances in Cryptology – EUROCRYPT '98*, pages 437–447.

[3] The Anonymizer. http://www.anonymizer.com.

[4] Geremie R. Barme and Sang Ye. The Great Firewall of China. http://www.wired.com/wired/5.06/china.html.

[5] Oliver Berthold, Hannes Federrath, and Marit Kohntopp. Anonymity and unobservability on the Internet. In *Workshop on Freedom and Privacy by Design : CFP 2000*, 2000.

[6] Oliver Berthold, Hannes Federrath, and Marit Kohntopp. Anonymity and unobservability on the Internet. In *Workshop on Freedom and Privacy by Design : CFP 2000*, 2000.

[7] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 4(2), February 1982.

[8] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[9] Matt Dietrich. Fcc ruling won't affect low-power radio pioneer. http://www.infoshop.org/news5/kantako.html, January 2000.

[10] Roger Dingledine. The Free Haven Project. Master's thesis, MIT, 2000.

[11] Electronic Frontiers Georgia (EFGA). Anonymous remailer information. http://anon.efga.org/Remailers/.

[12] Ian Goldberg and Adam Shostack. Freedom network 1.0 architecture, November 1999.

[13] Ian Goldberg and David Wagner. Rewebber. *First Monday*.

[14] Ian Goldberg and David Wagner. Taz servers and the rewebber network enabling anonymous publishing on the world wide web. http://www.cs.berkeley.edu/~daw/classes/cs268/taz-www/rewebber.html.

[15] C. Gulcu and G. Tsudik. Mixing e-mail with Babel. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, pages 2–16, 1996.

[16] Johan Helsingius. press release announcing closure of anon.penet.fi. http://www.penet.fi/press-english.html.

[17] David Hopwood. Definition of recipient-hiding cryptosystem. sci.crypt usenet post.

[18] Infoshop.org. Social struggle. http://www.infoshop.org/faq/secJ4.html#secj41.

[19] M. Jakobsson. Flash mixing. In *Principles of Distributed Computing PODC '99.*

[20] M. Jakobsson. A practical mix. In *Advances in Cryptology – EUROCRYPT '98.*

[21] D. Kesdogan, A. Trofimov, and D. Trossen. Minimizing the average cost of paging on the air interface. In *KuVS Springer-Verlag*, 1999.

[22] Dogan Kesdogan and ... Stop and go mixes : Providing probabilistic anonymity in an open system. In *1998 Information Hiding Workshop.*

[23] Jason Kroll. Crackers and crackdowns. http://www2.linuxjournal.com/articles/culture/007.html.

[24] Lucent personalised web assistant. http://www.lpwa.com.

[25] David Michael Martin. PhD thesis, Boston University, 2000. http://www.cs.du.edu/ dm/anon.html.

[26] David Mazieres and M. Frans Kaashoek. The design and operation of an e-mail pseudonym server. In *5th ACM Conference on Computer and Communications Security*, 1998.

[27] Stanley Milgram. The Small World Problem. *Psychology Today*, 2:60–67, 1967.

[28] Mix-l mixmaster discussion list. mix-l-subscribe@egroups.com.

[29] Ulf Möller and Company. Mixmaster 2.9b source code. http://mixmaster.anonymizer.com.

[30] July 1993.

[31] British House of Commons. Regulation of investigatory powers bill. http://www.parliament.the-stationery-office.co.uk/pa/cm199900/cmbills/064/2000064.htm.

[32] Onion router. http://www.onion-router.net/.

[33] A. Pfitzmann, B. Pfitzmann, and M. Waidner. ISDN-Mixes : Untraceable communication with small bandwidth overhead. In *GI/ITG Conference: Communication in Distributed Systems*, pages 451–463. Springer-Verlag, 1991.

[34] Associated Press. German court: AOL liable for music piracy. http://www.usatoday.com/life/cyber/tech/review/crh053.htm.

[35] The proxomitron. http://www.proxomitron.cjb.net/.

[36] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *DIMACS Technical Report*, 97(15), April 1997.

[37] K. Sako and J. Killian. Receipt-free mix-type voting scheme. In *Advances in Cryptology – EUROCRYPT '95*, pages 393–403.

[38] Russell Samuels. Untraceable nym creation on the freedom network, November 1999.

[39] Adam Shostack and Ian Goldberg. Freedom 1.0 security issues and analysis, November 1999.

[40] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[41] Duncan J. Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, page 393, 1998.

[42] Zero Knowledge Systems. http://www.freedom.net/.

Appendix A

Acknowledgements

Quite a few people contributed to the ideas and designs contained herein.

- Roger Dingledine for the formulation of the Free Haven Project and allowing me the opportunity to research and design certain aspects of the project. None of this work would be possible without his direction and conceptions.
- David Molnar for significant discussions of anonymous communications channels and module design. Much of the "related works appendix" on mix-nets has been formulated by David.
- Brian Sniffen for helping design and implement the Free Haven trust module and for providing a specification of the AC-Net, as well as Joseph Sokol-Margolis for helping with the overall system design.
- Professor Ron Rivest for his role as my project advisor, his suggestions for the overall Free Haven Project design, and for teaching me about computer and network security in various classes at MIT.

Appendix B

Anonymous Communications Channels

We earlier described several major implementations of anonymous communications channels. This appendix serves to give a more detailed survey of research and development in the area of anonymous communications. Some of these projects are not implemented; some exist more as a proof-of-concept by their respective designers; and still others repeat design and functionality provided by like systems.

We review three main types of design: proxy-servers, mix-nets, and other anonymous communications channels.[1]

B.1 Proxy Services

Proxy services provide one of the most basic forms of anonymity, inserting a third party between the sender and recipient of a given message. Proxy services are characterized as having only one centralized layer of separation between message sender and recipient. The proxy serves as a "trusted third party," responsible for sufficiently stripping headers and other distinguishing information from sender requests.

Proxies only provide unlinkability between sender and receiver, given that the proxy itself remains uncompromised. This unlinkability does not have the quality of perfect forward anonymity, as proxy users often connect from the same IP address. Therefore, any future information used to gain linkability between sender and receiver (i.e., intersection attacks, traffic analysis) can be used against previously recorded communications.

[1]David Molnar contributed heavily to the writing of this section. Virtually the entirety of the text that is not repeated in the Related Works section 4 was written by him.

Sender and receiver anonymity is lost to an adversary that may monitor incoming traffic to the proxy. While the actual contents of the message might still be computationally secure via encryption, the adversary can correlate the message to a sender/receiver agent.

This loss of sender/receiver anonymity plagues all systems which include external clients which interact through a separate communications channel – that is, we can define some distinct edge of the channel. If an adversary can monitor this edge link or the first-hop node within the channel, this observer gains agent-message correlation. Obviously, the ability to monitor this link or node depends on the adversary's resources and the number of links and nodes which exist. In a proxy system, this number is small. In a globally-distributed mixnet, this number could be very large. The adversary's ability also depends on her focus: whether she is observing messages and agents at random, or if she is monitored specific senders/receivers on purpose.

B.1.1 Anonymizer.com

The Anonymizer was one of the first examples of a *form-based web proxy* [3]. Users point their browsers at the Anonymizer page at `www.anonymizer.com`. Once there, they enter their destination URL into a form displayed on that page. The Anonymizer then acts as an `http` proxy for these users, stripping off all identifying information from `http` requests and forwarding them on to the destination URL.

The functionality is limited. Only `http` requests are proxied, and the Anonymizer does not handle cgi scripts. In addition, unless the user chains several proxies together, he or she may be vulnerable to an adversary which tries to correlate incoming and outgoing `http` requests. Only the data stream is anonymized, not the connection itself. Therefore, the proxy does not prevent traffic analysis attacks like tracking data as it moves through the network.

B.1.2 Lucent's Proxymate

Chaining multiple proxies together by hand is a tedious business, requiring many preliminaries before the first web page is reached. Lucent's Proxymate software automates the process[24]. The software looks like a proxy sitting on the user's computer. By setting software to use the Proxymate proxy, the user causes the software's requests and traffic to go to the software, which then automatically negotiates a chain of proxies for each connection.

B.1.3 Proxomitron

Another piece of software which helps manage many distinct proxies in a transparent manner is Proxomitron[35]. In addition to basic listing and chaining of proxies, Proxomitron allows users to write filter scripts. These filters can then be applied to incoming and outgoing traffic to do everything from detecting a request for the user's e-mail address by a web site to automatically changing colors on incoming web pages.

B.2 Chaumian Mix-nets

The project of anonymity on the Internet was kicked off by David Chaum in 1981 with a paper in Communications of the ACM describing a system called a "Mix-net." This system uses a very simple technique to provide anonymity: a sender and receiver are linked by a chain of servers called Mixes. Each Mix in the chain strips off the identifying marks on incoming messages and then sends the message to the next Mix, based on routing instructions which encrypted with its public key. Comparatively simple to understand and implement, this Mix-net (or "mix-net" or "mixnet") design is used in almost all of today's practical anonymous channels.

B.2.1 Chaum's Digital Mix

Chaum's original paper introduced the basic concept of a Mix as a sort of "permutation box." On the incoming side is a list of messages representing the messages which have arrived at the Mix server, each of which is identified with a particular sender. On the outgoing side is a randomly permuted list of messages, which have lost their identification with the sender. The assumption is that if the Mix works correctly, no adversary can do better than guessing to link an incoming message with an outgoing message.

B.2.2 ISDN Mixes

Chaum's original Digital Mix was described in terms of a series of Mix nodes which passed idealized messages over a network. The first proposal for the practical application of mixes came from Pfitzmann et. al. [33], who showed how a mix-net could be used with ISDN lines to anonymize a telephone user's real location. Their motivation was to protect the privacy of the user in the face of a telephone network owned by a state telephone monopoly.

Their paper introduced a distinction between *explicit* and *implicit* addresses. An explicit address is something about a message which clearly and unambiguously links it to a recipient and can be read by everyone, such as a To: header. An implicit address is an attribute of a message which links it to a recipient and can only be determined by that recipient. For example, being encrypted with the recipient's public key in a recipient-hiding public key is an implicit address.

B.3 Remailers: SMTP Mix-nets

Until the rise of proxy-based and TCP/IP-based systems, the most popular form of anonymous communication was the *anonymous remailer*: a form of mix which works for e-mail sent over SMTP. Remailers are informally divided into three categories, called Type 0, Type 1, and Type 2.

B.3.1 Type 0: anon.penet.fi

One of the first and most popular remailers was `anon.penet.fi`, run by Johan Helsingius. This remailer was very simple to use. A user simply added an extra header to e-mail indicating the final destination, which could be either an e-mail address or a Usenet newsgroup. This e-mail was sent to the `anon.penet.fi` server, which stripped off the return address and forwarded it along. In addition, the server provided for return addresses of the form "anXXXX@anon.penet.fi"; mail sent to such an address would automatically be forwarded to another e-mail address. These pseudonyms could be set up with a single e-mail to the remailer; the machine simply sent back a reply with the user's new pseudonym.

The `anon.penet.fi` remailer is referred to as a Type 0 remailer for two reasons. First, it was the original "anonymous remailer." More people used `anon.penet.fi` than are known to have used any following type of remailer. Exact statistics are hard to come by, but X number of accounts were registered at `penet.fi`, and only Y are currently registered at `nym.alias.net`.

Second, `anon.penet.fi` did not provide some of the features which motivated the development of "Type I" and "Type II" remailers. In particular, it provided a single point of failure and the remailer administrator had access to each user's "real" e-mail address. In general, any remailer system which consists of a single hop is considered Type 0.

This last feature proved to be the service's undoing. The Church of Scientology, a group founded by the science fiction writer L. Ron Hubbard, sued a `penet.fi` pseudonym for distributing materials reserved for high initiates to a Usenet newsgroup. Scientology claimed that the material was copyrighted "technology." The poster claimed it was a fraud used to extort money from gullible and desperate fools. Scientology won a court judgment requiring the `anon.penet.fi` remailer to give up the true name of the pseudonymous poster, which the operator eventually did. This incident, plus several allegations of traffic in child pornography, eventually convinced Johan Helsingius to close the service in 1995[16].

Services similar to Type 0 remailers now exist in the form of "free e-mail" services such as Hotmail, Hushmail, and ZipLip, which allow anyone to set up an account via a web form. Hushmail and ZipLip even keep e-mail in encrypted form on their server. Unfortunately, these services are not sufficient by themselves, as an eavesdropping adversary can determine which account corresponds to a user simply by watching him or her login.

B.3.2 Type 1: Cypherpunks Remailers

The drawbacks of anon.penet.fi spurred the development of "cypherpunks" or "Type 1" remailers, so named because their design took place on the cypherpunks mailing list. This generation of remailers addressed the the two major problems with `anon.penet.fi`: first, the single point of failure, and second, the vast amount of information about users of the service collected at that point of failure. Several remailers exist; a current list can be found at the Electronic Frontiers Georgia site [11] or on the newsgroup alt.privacy.anon-server.

Each cypherpunk remailer has a public key and uses PGP for encryption. Mail can be sent to each remailer encrypted with its key, preventing an eavesdropper from seeing it in transit. A message sent to a remailer can consist of a request to remail to another remailer and a message encrypted with the second remailer's public key. In this way a chain of remailers can be built, such that the first remailer in the chain knows the sender, the last remailer knows the recipient, and the middle remailers know neither.

Cypherpunk remailers also allow for *reply blocks.* These consist of a series of routing instructions for a chain of remailers which define a route through the remailer net to an address. Reply blocks allow users to create and maintain pseudonyms which receive e-mail. By prepending the reply block to a message and sending the two together to the first remailer in the chain, a message can be sent to a party without knowing his or her real e-mail address.

B.3.3 Type 2: Cottrell's Mixmaster

While Cypherpunk remailers represented a major advance over anon.penet.fi, they fell short of the anonymity provided by the ideal mix. In 1995, Lance Cottrell outlined some of the problems with "Type I" remailers [11]:

- **Traffic Analysis:** Cypherpunk remailers tend to send messages as soon as they arrive, or after some specified amount of delay. The first option makes it easy for an adversary to correlate messages across the mix-net. It's not clear how much delay helps protect against this attack.
- **Does Not Hide Length:** The length of messages is not hidden by the encryption used by cypherpunk remailers [2]. This allows an adversary to track a message as it passes through the mixnet by looking for messages of approximately the same length.

Cottrell wrote the Mixmaster, or "Type II", remailer to address these problems. Instead of using PGP, Mixmaster uses its own client software (which is also the server software), which understands a special Mixmaster packet format. All Mixmaster packets are the same length. Every message is

[2]note that the definitions of semantic security and non-malleability do not seem to imply "length-hiding" either

encrypted with a separate 3DES key for each mix node in a chain between the sender and receiver; these 3DES keys are in turn encrypted with the RSA public keys of each mix node.

When a message reaches a mix node, it decrypts the header, decrypts the body of the message, and then places the message in a "message pool." Once enough messages have been placed in the pool, the node picks a random message to forward.

As of this writing, Mixmaster is in version 2.9b22[29]. Discussion of the project can be found on the mix-l mailing list[28]. A Mixmaster version 3 is planned in which nodes will communicate with each other via TCP/IP connections. All traffic will be encrypted with a key derived by a Diffie-Hellman key exchange and then destroyed immediately after the transaction is ended, thereby providing perfect forward secrecy. Unfortunately, the prototype specification for this protocol is only available in German and is not finished.

B.3.4 Nymservers and nym.alias.net

The reply blocks used by cypherpunks remailers are important for providing for return traffic, but they must be sent to every correspondent individually. In addition, using a reply block requires that a correspondent be familiar with the use of specialized software. This problem is addressed by *nymservers*, which act as holding and processing centers for reply blocks.

To use a nymserver, a user simply registers an e-mail address of the form "nym@nymserver.net" and associates a reply block with it. This association can be carried out via anonymous e-mail. Then whenever a message is sent to "nym@nymserver.net," the nymserver automatically prepends the associated reply block, encrypts the aggregate, and sends it off to the appropriate anonymous remailer.

The most popular nymserver may be the one run at nym.alias.net, which is hosted at MIT's Lab for Computer Science. A recent report by Mazieres and Kaashoek details the technical and social details of running the nymserver, including problems of abuse[26].

B.3.5 Remailer User Interfaces

The major reason for the massive popularity of `anon.penet.fi` was that it was extremely easy to use. Anyone who could type "Request-Remailing-To:" at the top of an e-mail message could send anonymous e-mail. With the advent of remailers which required the use of PGP or the Mixmaster software, the difficulty of using remailers increased. This difficulty was aggravated by the fact that for years, both PGP and Mixmaster were only available as command-line applications with a bewildering array of options.

B.4 Recent Mix-Net Designs

B.4.1 TAZ / Rewebber

Goldberg and Wagner applied Mixes to the task of designing an anonymous publishing network called Rewebber[13]. Rewebber uses URLs which contain the name of a Rewebber server and a packet of encrypted information. When typed into a web browser, the URL sends the browser to the Rewebber server, whch decrypts the associated packet to find the address of either another Rewebber server or a legitimate web site. In this way, web sites can publish content without revealing their location.

Mapping between intelligible names and Rewebber URLs is performed by a name server called the Temporary Autonomous Zone(TAZ), named after a novel by Hakim Bey. The point of the "Temporary" in the name of the nameserver (and the novel) is that static structures are vulnerable to attack. Continually refreshing the Rewebber URL makes it harder for an adversary to gain information about the server to which it refers.

B.4.2 Babel

Contemporary with Cotrell's Mixmaster is an effort by Gulcu and Tsudik called "Babel"[15]. Babel uses a modified version of PGP as its underlying encryption engine. This modified version does not include normal headers, which would include the identity of the receiver, the PGP version number, and other identifying information.

The Babel paper defines quantities called the "guess factor" and the "mix factor" which model the ability of an adversary to match messages passing through the mix with their original senders. Then several attacks are presented, including the trickle and flooding attack, along with some countermeasures. The paper is noteworthy in that it attempts to give an analysis of just how much the practice of batching messages helps the untraceability of a mix-net node.

B.4.3 Stop and Go Mixes

The next step in probabilistic analysis for mixnets comes in the work of Kesdogan, Egner, and Buschkes [22], who proposed the "Stop and Go Mix." They divide networks into two kinds: "closed" networks, in which the number of users is small, known in advance, and all users can be made distinct, and "open" networks like the Internet with extremely large numbers of users. They claim that perfect anonymity cannot be achieved in these open networks, because there is no guarantee that every single client of the mix node is not the same person coming under different names.

Instead, they define and consider a notion of *probabilistic anonymity*: given that the adversary controls some percentage of the clients, some other set of mix servers, and is watching a Mix, can the probability of correlating messages be quantified in terms of some security parameter? They consider queueing theory as an inspiration for a statistical model and manage to prove theorems about the adversary's knowledge in this model.

B.4.4 Variable Implicit Addresses

Later, Kesdogan et. al. applied Mixes to the GSM mobile telephone setting[21]. Here, the point is to allow for GSM roaming from cell to cell while still protecting the user's real location from discovery by the phone company or an outside intruder. This is done by the use of *variable implicit addresses*, which work as follows : each roaming area has a publically known and static explicit address. When the client GSM phone comes online or crosses the boundaries of a cell, it queries the surrounding cells and downloads these addresses. Then it creates a new address for itself which combines the addresses of its surrounding cells.

Then, instead of sending the entirety of the new address, the phone sends only some characters, say $logn$, of the address to the network to identify itself. The network then directs traffic intended for the phone to any cell which has those $logn$ characters in its address. A refinement process then takes place in which the phone gives out slightly more information to the system to improve performance by sending information to fewer cells, but not so much as to allow its location to be restricted to only one cell.

B.4.5 Jakobsson's Practical Mix

At EUROCRYPT '98, Jakobsson proposed a mixnet which was both practical and could be proved to mix correctly as long as less than 1/2 of the servers were corrupted[20]. The crucial idea is to treat the mixing as a secure multiparty computation in which each party is collaborating to make the collective mix look like a "random enough" permutation on a batch of messages. Then techniques of zero-knowledge proof are used by which each server can prove to all other servers that they are in fact conforming to the mix protocol. Deviating servers cannot produce valid proofs, and so can be caught and excluded from future mixing. Jakobsson's original protocol requires in the neighborhood of 160 modular exponentiations per message per server.

At PODC '99, Jakobsson showed how the use of precomputation could reduce the cost even further[19]. This new "flash mix" required only around 160 modular *multiplications* per message per server. This level of efficiency makes flash mixing competitive with the encryption used in anonymous remailers, and a serious candidate for low-latency mixing.

B.4.6 Universally Verifiable Mix-nets

With Jakobsson's design, the correctness of a mix-net can only be verified by the mix servers themselves. When more than a threshold of servers is corrupt, the verification fails. Because a user of the mix-net may not be aware of the corruption, this failure may be silent and therefore dangerous. One solution to this problem is a *universally verifiable* mix-net – a mix-net whose correctness can be verified by anyone, regardless of their status as server or user.

The concept was introduced by Killian [37], and recently a design of this type was proposed at EUROCRYPT '98 by Abe [2]. This design works along the similar broad lines as the Jakobsson design; each mix server uses zero-knowledge proofs to prove that it is acting in accordance with some protocol to randomly mix messages. The difference here is that these proofs are posted publically by the mix nodes instead of being multicast only to other mix nodes. The novel feature of Abe's design is that the work necessary to verify these proofs grows in a fashion independent of the number of servers. Unfortunately, verifying these proofs requires on the order of 1600 modular exponentiations per message.

B.4.7 Onion Routing

The Onion Routing system designed by Syverson, et. al. creates a mix-net for TCP/IP connections [40, 32]. In the Onion Routing system, a mixnet packet, or "onion", is created by successively encrypting a packet with the public keys of several mix servers, or "onion routers."

When a user places a message into the system, an "onion proxy" determines a route through the anonymous network and onion encrypts the message accordingly. Each onion router which receives the message peels the topmost layer, as normal, then adds some key seed material to be used to generate keys for the anonymous communication. As usual, the changing nature of the onion – the "peeling" process – stops message coding attacks. Onions are numbered and have expire times, to stop replay attacks. Onion routers maintain network topology by communicating with neighbors, using this information to initially build routes when messages are funneled into the system. By this process, routers also establish shared DES keys for link encryption.

The routing is performed on the application layer of onion proxies, the path between proxies dependent upon the underlying IP network. Therefore, this type of system is comparable to loose source routing.

Onion Routing is mainly used for sender-anonymous communications with non-anonymous receivers. Users may wish to Web browse, send email, or use applications such as `rlogin`. In most of these real-time applications, the user supplies the destination hostname/port or IP address/port. Therefore, this system only provides receiver-anonymity from a third-party, not from the sender.

Furthermore, Onion Routing makes no attempt to stop timing attacks using traffic analysis at the network endpoints. They assume that the routing infrastructure is uniformly busy, thus making passive intra-network timing difficult. However, the network might not be statistically uniformly busy, and attackers can tell if two parties are communicating via increased traffic at their respective endpoints. This endpoint-linkable timing attack remains a difficulty for all low-latency networks.

B.4.8 Zero Knowledge Systems

Recently, the Canadian company Zero Knowledge Systems has begun the process of building the first mix-net operated for profit, known as Freedom [42]. They have deployed two major systems, one for e-mail and another for TCP/IP. The e-mail system is broadly similar to Mixmaster, and the TCP/IP system similar to Onion Routing.

ZKS's "Freedom 1.0" application is designed to allow users to use a nym to anonymously access web pages, use IRC, etc. The anonymity comes from two aspects: first of all, ZKS maintains what it calls the Freedom Network, which is a series of nodes which route traffic amongst themselves in order to hide the origin and destination of packets, using the normal layered encryption mixnet mechanism. All packets are of the same size. The second aspect of anonymity comes from the fact that clients purchase "tokens" from ZKS, and exchange these token for nyms – supposedly even ZKS isn't able to correlate identities with their use of their nyms.

The Freedom Network looks like it does a good job of actually demonstrating an anonymous mixnet that functions in real-time. The system differs from Onion Routing in several ways.

First of all, the system maintains Network Information Query and Status Servers, which are databases which provide network topology, status, and ratings information. Nodes also query the key servers every hour to maintain fresh public keys for other nodes, then undergo authenticated Diffie-Hellman key exchange to allow link encryption. This system differs from online inter-node querying that occurs with Onion Routing. Combined with centralized nym servers, time synchronization, and key update/query servers, the Freedom Network is not fully decentralized [12].

Second, the system does not assume uniform traffic distribution, but instead uses a basic "heart-beat" function that limits the amount of inter-node communication. Link padding, cover traffic, and a more robust traffic-shaping algorithm have been planned and discussed, but are currently disabled due to engineering difficulty and load on the servers. ZKS recognizes that statistical traffic analysis is possible [39].

Third, Freedom loses anonymity for the primary reason that it is a commercial network operated for profit. Users must purchase the nyms used in pseudonymous communications. Purchasing is performed out-of-band via an online Web store, through credit-card or cash payments. ZKS uses a protocol of issuing serial numbers, which are reclaimed for nym tokens, which in turn are used to anonymously purchase nyms. However, this system relies on "trusted third party" security: the user must trust that ZKS is not logging IP information or recording serial–token exchanges that would allow them to correlate nyms to users [38].

B.4.9 Web Mixes

Another more recent effort to apply a Mix network to web browsing is due to Federrath et. al.[6] who call their system, appropriately enough, "Web Mixes." From Chaum's mix model, similar to other real-time systems, they use: layered public-key encryption, prevention of replay, constant message length within a certain time period, and reordering outgoing messages.

The Web Mixes system incorporates several new concepts. First, they use an adaptive "chop-and-slice" algorithm that adjusts the length used for all messages between time periods according to the amount of network traffic. Second, dummy messages are sent from user clients as long as the clients are connected to the Mix network. This cover traffic makes it harder for an adversary to perform traffic analysis and determine when a user sends an anonymous message, although the adversary can still tell when a client is connected to the mixnet. Third, Web Mixes attempt to restrict insider and outsider flooding attacks by limited either available bandwidth or the number of used time slices for each user. To do this, users are issued a set number of blind signature tickets for each time slice, which are spent to send anonymous messages. Lastly, this effort includes an attempt to build a statistical model which characterizes the knowledge of an adversary attempting to perform traffic analysis.

B.5 Other Anonymous Channels

B.5.1 The Dining Cryptographers

The Dining Cryptographers protocol was introduced by David Chaum[8] and later improved by Pfitzmann and Waidner[] as a means of guaranteeing untraceability for the sender and receiver of a message, even against a computationally all-powerful adversary. The protocol converts any broadcast channel into an anonymous broadcast channel. In the context of Free Haven, however, we have a problem : the participants in the protocol are identified, even though the sender and receiver of any given message is not. If the only long-term participants in the protocol are likely to be Free Haven servnet nodes, then we do not achieve the server-anonymity we desire. Less serious, but still important, problems are the efficiency of the protocol and the difficulty of correct implementation.

Therefore we have not seriously considered using the dining cryptographers protocol to provide Free Haven's anonymous channel. If we were to do so, we might consider running a dining cryptographer protocol using Mixes to hide the legal identity of each participant. In that case, while a failure of the Mix would reveal a participant's identity, the anonymous broadcast would prevent him or her from being linked to any particular message.

B.5.2 Crowds

The Crowds system was proposed and implemented by AT&T Research, named for collections of users that are used to achieve partial anonymity for Web browsing [36]. A user initially joins some crowd and her system begins acting as a node, or anonymous *jondo*, within that crowd. In order to instantiate communications, the user creates some path through the crowd by a random-walk of *jondos*, in which each *jondo* has some small probability of sending the actual `http` request to the end server. Once established, this path remains static as long as the user remains a member of that crowd. The Crowds system does not use dynamic path creation so that colluding crowd eavesdroppers are not able to probabilistically determine the initiator (i.e., the actual sender) of requests, given repeated requests through a crowd. The *jondos* in a given path also share a secret *path key*, such that local listeners, not part of the path, only see an encrypted end server address until the request is finally sent off. The Crowds system also includes some optimizations to handle timing attacks against repeated requests, as certain HTML tags cause browsers to automatically issue re-requests.

Similar to other real-time anonymous communication channels (Onion Routing, the Freedom Network, Web Mixes), Crowds is used for senders to communicate with a known destination. The system attempts to achieve sender-anonymity from the receiver and a third-party adversary. Receiver-anonymity is only meant to be kept from adversaries, not from the sender herself.

The Crowds system serves primarily to achieve sender and receiver anonymity from an attacker, not provide unlinkability between the two agents. Due to high availibility of data – real-time access is faster that mix-nets as Crowds does not use public key encryption – an adversary can more easily use traffic analysis or timing attacks. However, Crowds differs from all other systems we have discussed, as users are *members* of the communications channel, rather than merely communicating *through* it. Sender-anonymity is still lost to a local eavesdropper that can observe all communications to and from a node. However, other colluding *jondos* along the sender's path – even the first-hop – cannot expose the sender as originated the message. Reiter and Rubin show that as the number of crowd members goes to infinity, the probable innocence of the last-hop being the sender approaches one.

B.5.3 Ostrovsky's Anonymous Broadcast via XOR-Trees

In CRYPTO '97, Ostrovsky considered a slightly different model of anonymous broadcast[]. In this model, there are n servers broadcasting into a shared broadcast channel. One of the servers is a special "Command and Control" server; the rest are broadcasting dummy traffic. Then there is an adversary who has control of some of the servers and wants to know which server is the

"Command and Control." Ostrovsky shows how to use correlated pseudo-random number generators whose output reveals a certain message when XORed together to create a protocol which prevents the adversary from discovering which server is the correct one, even if he can eavesdrop on all communications and corrupt up to k servers, where k is a security parameter which affects the efficiency of the protocol.

Appendix C

Comm Module Code

Handling feeder and haven sockets: comm.c

```
#include "../shared/common.h"
#include "../shared/protos.h"
#include "comm.h"
#include "protos.h"

static char haven_can_host_name[MAX_HOSTLEN+1]; /* canonical form */

int haven_socket;
struct message_queue_t *haven_queue;
struct feeder_t *feeder;

int main(int argc, char **argv) {
  int listen_socket;

  if (argc < 3 || !str_is_number(argv[1]) || !str_is_number(argv[2])) {
    printf("Usage: %s incoming-port haven-port\n", argv[0]);
    exit_comm();
  }

  umask(022);

  INFO("Loading configuration options...\n");
  load_conf("../config/comm.conf");

  INFO("Initializing node database...\n");
  initialize_nodedb();

  INFO("Starting comm probe: incoming port %d, haven module on port %d\n",atoi(argv[1]), atoi(argv[2

  handle_ports(atoi(argv[1]), listen_socket, atoi(argv[2]));

  return (1);
}
```

```
void handle_ports(int incoming_port, int listen_socket, int haven_port) {
  fd_set rfds;
  int highest_fd; /* for use with select. contains one more than max(fd's)*/
  int select_result;
  struct timeval tv; /* for sleeping in select */

  int tmp_haven_socket;
  int haven_connected;
  int haven_waiting;
  static struct sockaddr_in haven_in;

  int incoming_socket;
  struct sockaddr_in incoming;
  int incominglen;
  char *sec_timeout, *usec_timeout;

  int new_feeder_socket;
  //  char feeder_buf[FEEDER_BUFSIZE];
  char *feeder_buf;
  struct feeder_t *prev_feeder, *curr_feeder;
  int feeder_read_result, written_buf;

  struct message_queue_t *haven_queue_entry;
  char *busiest_node_PK;


  //  char feederfile_name[LENGTH_OF_COMM_FILE_PATH+1];

  /* Resolve name of controller name into a socket address. */
  haven_in.sin_family = AF_INET;
  if ((haven_in.sin_addr.s_addr = resolve_name("localhost",
       haven_can_host_name, MAX_HOSTLEN)) == 0)
  {
      printf("Comm: unknown host %s\n", "localhost");
      exit_comm();
  }

  /* Fill in the controller port. */
  haven_in.sin_port = htons(haven_port);

  /* Create a socket of type stream (instead of datagram). */
  if ((tmp_haven_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Comm: socket failed");
    exit_comm();
  }

  /* haven initially not connected */
  haven_connected = 0;
  /* initially no feeder sockets connected */
  feeder = NULL;

  /* Wait until listen_connection gets incoming connection */
  incoming_socket = create_listening_tcpsocket(incoming_port);
  /* Set incoming socket to non-blocking */
  fcntl(incoming_socket, F_SETFL, O_NONBLOCK);


  /* main loop for comm, which never ends. */
  for(;;) {

    /* Haven module has possibly gone down */
    /* Open connection with haven module   */
    if (!haven_connected) {

      INFO("Comm: attempting haven connection...\n");

      /* Create a socket of type stream (instead of datagram). */
      if ((tmp_haven_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("Comm: haven socket failed");
          exit_comm();
      }
      haven_waiting = 0;
       /* Set up a connection with the controller. */
      if (connect(tmp_haven_socket, (struct sockaddr *) &haven_in, sizeof(haven_in)) < 0) {
if (errno == ECONNREFUSED) {
  INFO("Comm: haven connection failed, sleeping %i sec...\n", DEFAULT_TIME_SLEEP_FOR_HAVEN);
  sleep(DEFAULT_TIME_SLEEP_FOR_HAVEN);
  haven_waiting = 1;
} else {
          perror("Comm: connect to haven failed");
  exit_comm();
}
      }

      if (!haven_waiting) {
/* now haven_socket is what we're going to use. */
haven_socket = tmp_haven_socket;
haven_connected = 1;

INFO("Comm: connect to haven succeeded to %s on port %d\n",
     (char *)inet_ntoa(haven_in.sin_addr), haven_in.sin_port);
      }
    } /* end !haven_connented */


    /* set select timeout */
    if ((sec_timeout = get_conf("comm_socket_sec_timeout"))) {
      tv.tv_sec = atoi(sec_timeout);
    } else {
      tv.tv_sec = 0;
    }
    if ((usec_timeout = get_conf("comm_socket_usec_timeout"))) {
      tv.tv_usec = atoi(usec_timeout);
    } else {
      tv.tv_usec = 0;
    }

    /* prepare to select among possible socket connections */
    FD_ZERO(&rfds);
    FD_SET(incoming_socket, &rfds);

    if (!haven_waiting) {
      FD_SET(haven_socket, &rfds);
      highest_fd = (haven_socket > incoming_socket) ? haven_socket : incoming_socket;
    } else {
      highest_fd = incoming_socket;
    }

    /* add each feeder in linked list to select rfds */
    curr_feeder = feeder;
    while (curr_feeder) {
      FD_SET(curr_feeder->socket, &rfds);

      if (curr_feeder->socket > highest_fd) {
highest_fd = curr_feeder->socket;
      }
      curr_feeder = curr_feeder->next;
    }

    highest_fd++;


    /*******************************************/
    /* select and wait for socket transmissions */
    select_result = select(highest_fd, &rfds, NULL, NULL, &tv);

    if (select_result < 0) {
      perror("Comm: select failed");
      exit_comm();
    }

    /* one of more socket fd are selected.  Check rfds */
    if (select_result) {

      /* data on haven socket? */
      if (FD_ISSET(haven_socket, &rfds)) {
haven_connected = process_from_socket(haven_socket, get_conf("comm_tempfile_outgoing"));
      }



      /* data on incoming socket? */
      if (FD_ISSET(incoming_socket, &rfds)) {

if ((new_feeder_socket = accept(incoming_socket,
(struct sockaddr *) &incoming, &incominglen)) < 0) {
  // EAGAIN acceptable: non-blocking and no connection waiting */
  if (errno != EAGAIN) {
    perror("Comm: accepting incoming connection");
    exit_comm();
  }
}


if (new_feeder_socket >= 0) {  /* else nonblocking socket EAGAIN error */

  create_feeder_entry(new_feeder_socket);

  INFO("Comm: incoming socket from %s on port %d, using feeder file %s\n",
       (char *)inet_ntoa(incoming.sin_addr), incoming.sin_port, feeder->filename);
}
      }


      /* data on feeder sockets? */
      prev_feeder = NULL;
      curr_feeder = feeder;
      while (curr_feeder) {

if (FD_ISSET(curr_feeder->socket, &rfds)) {
  feeder_buf = (char *)malloc(FEEDER_BUFSIZE*sizeof(char));
  feeder_read_result = read(curr_feeder->socket, feeder_buf, FEEDER_BUFSIZE);

  INFO("Comm: reading feeder socket %i\n", curr_feeder->socket);

  /* Error encountered */
  if (feeder_read_result < 0) {
    perror("Comm: reading from feeder socket");
    exit_comm(); /* this is a bad idea, should fix later */
  }

  /* EOF has been reached */
  if (feeder_read_result == 0) {
    INFO("Comm: finished reading on feeder socket, written to feeder file %s\n",
 feeder->filename);
    process_feeder_entry(curr_feeder, prev_feeder);
    /* prepare for next iteration of feeder while loop */
    curr_feeder = prev_feeder;
  }

  while (feeder_read_result > 0) {
      written_buf = write(curr_feeder->fd, feeder_buf, feeder_read_result);
      feeder_buf += written_buf;
      feeder_read_result -= written_buf;
  }
} /* end if FD_ISSET */

prev_feeder = curr_feeder;
if (curr_feeder) {
  curr_feeder = curr_feeder->next;
}
```

```
      } /* end while (feeder) */

    } /* end if (select_result) */


    /* Dequeue and send file to haven */
    if (haven_queue && haven_connected) {
      /* Possibly handle multiple transfers during each cycle:
       *     send the entire queue  i.e, while (haven_queue) {...}
       */
      haven_queue_entry = haven_queue->next;
      send_file_to_haven(haven_queue->filename);
      free(haven_queue);
      haven_queue = haven_queue_entry;
    }


    /* check to process haven request, sending thing into mix */
    if ((busiest_node_PK = node_get_busiest_PK())) {
      do_transmit(busiest_node_PK);
    }

  } /* end main comm for(;;) loop */

}


void create_feeder_entry(int new_feeder_socket) {
  /* requires:  new_feeder_socket be valid socket > 0 */
  /* effects:   add new feeder entry to feeder queue  */
  struct feeder_t *feeder_entry;

  feeder_entry = malloc(sizeof(struct feeder_t));
  feeder_entry->socket = new_feeder_socket;

  sprintf(feeder_entry->filename, "%s/tmp-incoming-XXXXXX", get_conf("comm_incoming_directory_root")
  /* XXX Creates with mode 0666, or handled with umask? */
  feeder_entry->fd = mkstemp(feeder_entry->filename);

  if (feeder_entry->fd < 0) { /* error */
    perror("Comm: feeder mkstemp");
    exit_comm();
  }

  /* connect to head of feeder list */
  /* this is a FILO queue...is this okay?  Should we waste time and make FIFO? */
  feeder_entry->next = feeder;
  feeder = feeder_entry;

}

void process_feeder_entry(struct feeder_t *curr_feeder, struct feeder_t *prev_feeder) {
  close(curr_feeder->fd);
  enqueue_haven_message(curr_feeder->filename);
  close(curr_feeder->socket);

  if (prev_feeder) {
    /* remove struct from feeder queue */
    prev_feeder->next = curr_feeder->next;
  } else {
    /* top of feeder list */
    feeder = NULL;
  }
  free(curr_feeder);
}


/* Inter-freehaven communications:  comm -> mixnet -> comm */
void process_freehaven_message(char *filename) {

  enqueue_haven_message(filename);

}


void enqueue_haven_message(char *filename) {
  struct message_queue_t *queue = haven_queue, *queue_entry;

  queue_entry = malloc(sizeof(struct message_queue_t));
  strcpy(queue_entry->filename, filename);

  /* Push freehaven message into end of FIFO incoming_queue */
  if (queue) {
    while (queue->next) {
      queue = queue->next;
    }
    queue->next = queue_entry;
  } else {
    /* queue is empty */
    haven_queue = queue_entry;
  }
}

/* Intra-freehaven communications:  haven -> comm */
void process_internal_message(char *filename) {
  struct tag_t *tag_list;
  char *transaction_type;

  /* un-base-64(filename);          */
  /* decrypt message with pk_comm   */

  tag_list = build_tag_list_from_file(filename);

  if (!(transaction_type = get_tag(tag_list, "transaction"))) {
    /* message has no transaction type, dropping */
    free_tag_list(tag_list, TAGLIST_DELETE_SHAREFILES_TOO);
    return;
  }

  if (!strcmp(transaction_type, "broadcast")) {
    broadcast(tag_list);
  }

  if (!strcmp(transaction_type, "transmit")) {
    transmit(tag_list);
  }

  free_tag_list(tag_list, TAGLIST_DELETE_SHAREFILES_TOO);

}


/* already in proper tagged format */
void send_file_to_haven(char *filename) {
  /* open filename, shove through haven_socket */
  /* We might want to sign it, encrypt it, base64 it, etc. */

  /* Need to place <internal-message> wrappers around file */
  /* This will be handled in push_file_through_socket */
  INFO("Comm: sending file %s to haven.\n", filename);
  push_string_through_socket(haven_socket, strcat(INTERNAL_OPEN_MSG,"\n"));

  /* Need to kill tmp-file... */
  push_string_through_socket(haven_socket, "<internal-message>\n");
  push_file_through_socket(haven_socket, filename);
  push_string_through_socket(haven_socket, "</internal-message>\n");
}


void exit_comm() {

  /* save state of incoming / outgoing buffers */
  exit(1);
}
```

Broadcast messages from haven: broadcast.c

```
#include "../shared/common.h"
#include "comm.h"
#include "protos.h"

extern GDBM_FILE nodedb;

int broadcast(struct tag_t *tag_list) {

  char *sharefile;
  datum key, value;
  struct nodedb_entry_t node;
  struct tag_t *node_tag_list;

  /* do initial test to halt send execution if sharefile invalid */
  if (!(sharefile = get_tag(tag_list, "sharefile"))) {
    return(0);
  }

  INFO("Comm: broadcast message called...\n");

  key = gdbm_firstkey(nodedb);
  while (key.dptr) {

    value = gdbm_fetch(nodedb, key);
    if (value.dptr) {
      /* explicitly true - can't check for .dsize */
      node = reconstruct_nodedb_entry(value.dptr);
      /* transmit and broadcast appear same to receiver */
      node_tag_list = add_tag(node_tag_list, strdup("transaction"), strdup("transmit"));
      node_tag_list = add_tag(node_tag_list, strdup("sharefile"), strdup(sharefile));
      node_tag_list = add_tag(node_tag_list, strdup("PK"), strdup(key.dptr));
      node_tag_list = add_tag(node_tag_list, strdup("mixnet"), strdup(node.mixnet));
      node_tag_list = add_tag(node_tag_list, strdup("address"), strdup(node.address));

      transmit(node_tag_list);

      free_tag_list(node_tag_list, TAGLIST_LEAVE_SHAREFILES);
      free_node(node);
    } /* if false, keep iterating, do not return */

    free(value.dptr);
    key = gdbm_nextkey(nodedb, key);
  }
  return (1);

}
```

Transmit messages to specific nodes: transmit.c

```
#include "../shared/common.h"
#include "comm.h"
#include "protos.h"

extern GDBM_FILE nodedb;

int transmit(struct tag_t *tag_list) {

  char *sharefile, *hPKdest, *PK, *address, *mixnet, *filename;
  int fd, waiting;

  if(!(sharefile = get_tag(tag_list, "sharefile"))) {
    WARN("Transmit:  sharefile not present in tag_list.\n");
    return (0);
  }
  if(!(hPKdest = get_tag(tag_list, "hPKdest"))) {
    WARN("Transmit:  hPKdest not present in tag_list.\n");
    return (0);
  }
  if(!(PK = get_tag(tag_list, "PK"))) {
    PK = node_get_PK(hPKdest);
    if (!PK) {
      WARN("Transmit: node %s missing PK entry.\n", hPKdest);
      return (0);
    }
    tag_list = add_tag(tag_list, strdup("PK"), strdup(PK));
  }
  if(!(address = get_tag(tag_list, "address"))) {
    address = node_get_address(hPKdest);
    if (!address) {
      WARN("Transmit: node %s missing address entry.\n", hPKdest);
      return (0);
    }
    tag_list = add_tag(tag_list, strdup("address"), strdup(address));
  }
  if(!(mixnet = get_tag(tag_list, "mixnet"))) {
    mixnet = node_get_mixnet(hPKdest);
    if (!mixnet) {
      WARN("Transmit: node %s missing mixnet entry.\n", hPKdest);
      return (0);
    }
    tag_list = add_tag(tag_list, strdup("mixnet"), strdup(mixnet));
  }


  sprintf(filename, "%s/tmp-outgoing-XXXXXX", get_conf("comm_outgoing_directory_root"));
  fd = mkstemp(filename);

  if (fd < 0) { /* error */
    perror("Comm: transmit mkstemp");
    exit_comm();
  }

  build_file_from_tag_list(filename, tag_list);
  waiting = node_add_waiting_msg(hPKdest, filename);

  INFO("Comm: Enqueuing sharefile %s to PK %s:  %i waiting messages, file %s\n",
       sharefile, PK, waiting, filename);

  free(sharefile);
  free(hPKdest);
  free(PK);
  free(mixnet);
  free(address);
  return (1);
}


int do_transmit(char *hPK) {
  /* Extract the waiting file list for hPK node.  Glom all the
   * waiting files into one large file, and send this in one large
   * chunk through the remailer.
   */

  struct message_queue_t *message_queue;

  message_queue = node_get_message_queue(hPK);

  if (!message_queue) {
    INFO("Comm: Node %s to transmit has no waiting messages.\n", hPK);
    return (0);
  }

  /* cat sharefile | mix -S --subject="sharedesc" --to="towherever" */
  /* system() the command line                                      */
  /* for now just creates a command line and sends a test message.  */

   if (!(strcmp(mixnet, "mixmaster"))) {
     char * sendtestfile = "cat testfile | mix -S --subject=\"Iamfreehaven\" --to=\"freehaven-dev@se
     system(sendtestfile);
   }

  INFO("Comm: Busiest node %s transmitted...\n", hPK);

  return (1);
}
```

Node Database interface: nodedb.c

```
#include "../shared/common.h"
#include "comm.h"
#include "protos.h"

GDBM_FILE nodedb;


void initialize_nodedb() {

  nodedb = gdbm_open("nodedb", 512, GDBM_WRCREAT, 0777, 0);
  INFO("Node DB opened and initialized.\n");

}

datum construct_gdbm_entry(struct nodedb_entry_t *entry) {
  /* effects:  construct gdbm value from nodedb_entry_t struct
   * return:   gdbm value datum
   */
  datum value;
  void *val, *val_start;
  int len_PK = strlen(entry->PK)+1;
  int len_mixnet = strlen(entry->mixnet)+1;
  int len_address = strlen(entry->address)+1;
  int len_messages = 0, i = 0;
  struct message_queue_t *message;
  size_t dsize;

  WARN("Constructing gdbm entry...\n");

  message = entry->message_queue;
  while (message) {
    len_messages += strlen(message->filename)+1;

    WARN("file: %s\n", message->filename);

    message = message->next;
    i++;
  }

  if (i != entry->waiting_msgs) {
    WARN("Nodedb: waitings_msgs [%i] != number of filenames [%i]\n",
 entry->waiting_msgs, i);
  }

  dsize = 3*sizeof(int) +
    len_PK*sizeof(char) +
    len_messages*sizeof(char) +
    len_mixnet*sizeof(char) +
    len_address*sizeof(char);
  val = (void *)malloc(dsize);
  val_start = val;

  // XXX Maybe a portability problem for systems with different endian-ness
  memcpy(val, &entry->statistics, sizeof(int));
  val += sizeof(int);
  memcpy(val, &entry->cost, sizeof(int));
  val += sizeof(int);
  memcpy(val, &entry->waiting_msgs, sizeof(int));
  val += sizeof(int);

  message = entry->message_queue;
  while (message) {
    /* shouldn't this be &message->filename */
    strcpy(val, message->filename);
    val += (strlen(message->filename)+1)*sizeof(char);
    message = message->next;
  }


  strcpy(val, entry->PK);
  val += len_PK*sizeof(char);
  strcpy(val, entry->mixnet);
  val += len_mixnet*sizeof(char);
  strcpy(val, entry->address);

  value.dptr = (char *)val_start;
  value.dsize = dsize;

  return (value);
}


struct nodedb_entry_t reconstruct_nodedb_entry(void *nodedb_value) {
  /* effects: reconstruct struct nodedb_entry_t from
   *          gdbm value.  Keeps FILO (head - new, tail - old)
   *          ordering of message queue.
   * return:  nodedb_entry_t set to gdbm info
   */

  struct nodedb_entry_t nodedb_entry;
  struct message_queue_t *message, *prev_message;
  int i;
  char *loc;

  memcpy((int *) nodedb_entry.statistics, nodedb_value, sizeof(int));
  nodedb_value += sizeof(int);
  memcpy((int *) nodedb_entry.cost, nodedb_value, sizeof(int));
  nodedb_value += sizeof(int);
  memcpy((int *) nodedb_entry.waiting_msgs, nodedb_value, sizeof(int));
  nodedb_value += sizeof(int);

  loc = (char *)nodedb_value;
  if (nodedb_entry.waiting_msgs > 0) {
    nodedb_entry.message_queue = malloc(sizeof(struct message_queue_t));

    strcpy(nodedb_entry.message_queue->filename, loc);

    prev_message = nodedb_entry.message_queue;

    for (i=1; i < nodedb_entry.waiting_msgs; i++) {
      /* more than one message waiting */
      loc += strlen(prev_message->filename) + 1;
      message = malloc(sizeof(struct message_queue_t));
      strcpy(message->filename, (char *)loc);
      prev_message->next = message;
      prev_message = message;
    }
  }
  strcpy(nodedb_entry.PK, loc);
  loc += strlen(nodedb_entry.PK) + 1;
  strcpy(nodedb_entry.mixnet, loc);
  loc += strlen(nodedb_entry.mixnet) + 1;
  //  loc = strchr(loc, '\0') + 1;
  strcpy(nodedb_entry.address, loc);

  return (nodedb_entry);
}



int node_change_PK(char *hPK, char *newPK) {
  /* update a node PK       */
  /* returns < 0 if fails   */

  datum key, value;
  struct nodedb_entry_t nodedb_entry;
  int ret_val = 1;

  if (!hPK || !newPK) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return(-1);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);
  free(nodedb_entry.PK);
  nodedb_entry.PK = newPK;
  value = construct_gdbm_entry(&nodedb_entry);

  if (gdbm_store(nodedb, key, value, GDBM_INSERT)) {
    WARN("Nodedb: Hash(PK) %s not changed.\n", hPK);
    ret_val = -2;
  }

  if (gdbm_delete(nodedb, key)) {
    WARN("Nodedb: Old PK not deleted properly from nodedb: hPK %s!\n", hPK);
    ret_val = -3;
  }

  free(value.dptr);
  free_node(nodedb_entry);

  return (ret_val);
}



int node_add_new_node(char *hPK, char *PK, char *address, char *mixnet, int statistics) {
  /* no statistics initially? Or poll from web? */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  nodedb_entry.statistics = statistics;
  nodedb_entry.cost = 0;
  nodedb_entry.waiting_msgs = 0;
  nodedb_entry.message_queue = NULL;
  nodedb_entry.PK = strdup(PK);
  nodedb_entry.mixnet = strdup(mixnet);
  nodedb_entry.address = strdup(address);

  value = construct_gdbm_entry(&nodedb_entry);

  if (gdbm_store(nodedb, key, value, GDBM_INSERT)) {
    WARN("PK %s already exists in nodedb when attempting to add.\n", PK);
    free(value.dptr);
    return(-1);
  }

  free(value.dptr);
  return(1);

}


struct message_queue_t* node_get_message_queue(char *hPK) {
  /* look it up, return the front of struct queue */
  /* return NULL if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (NULL);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if (!value.dptr) {
    return (NULL);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free(nodedb_entry.PK);
  free(nodedb_entry.address);
  free(nodedb_entry.mixnet);

  return (nodedb_entry.message_queue);
}



char* node_get_PK(char *hPK) {
  /* look it up, return the PK     */
  /* return NULL if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (NULL);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return (NULL);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free_node_msgs(nodedb_entry);
  free(nodedb_entry.mixnet);
  free(nodedb_entry.address);

  return (nodedb_entry.PK);
}


char* node_get_mixnet(char *hPK) {
  /* look it up, return the mixnet name */
  /* return NULL if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (NULL);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return (NULL);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free_node_msgs(nodedb_entry);
  free(nodedb_entry.PK);
  free(nodedb_entry.address);

  return (nodedb_entry.mixnet);
}


char* node_get_address(char *hPK) {
  /* look it up, return the address */
  /* return NULL if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (NULL);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return(NULL);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free_node_msgs(nodedb_entry);
  free(nodedb_entry.PK);
  free(nodedb_entry.mixnet);

  return (nodedb_entry.address);
}


char* node_get_busiest_PK() {
  /* return the PK of the node with the greatest */
  /* number of waiting messages to be processed. */
  /* if no messages are waiting, NULL returned.  */

  datum key, value;
  struct nodedb_entry_t nodedb_entry;
  int max_cost = 0;
  char *busiest_hPK = NULL;

  key = gdbm_firstkey(nodedb);

  while (key.dptr) {
    value = gdbm_fetch(nodedb, key);
    /* XXX if (value.dptr) explicitly true ? */
    nodedb_entry = reconstruct_nodedb_entry(value.dptr);

    if (nodedb_entry.cost > max_cost) {
      max_cost = nodedb_entry.cost;
      if (busiest_hPK) free(busiest_hPK);
      busiest_hPK = strdup(key.dptr);
    }

    free(value.dptr);
    free_node(nodedb_entry);

    key = gdbm_nextkey(nodedb, key);
  }

  return (busiest_hPK);
}


int node_add_waiting_msg(char *hPK, char *filename) {
  /* add filename to PK's queue of waiting messages  */
  /* return the number of waiting_msgs, -1 if failure */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;
  struct message_queue_t *new_message;

  if (!hPK || !filename) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if (!value.dptr) {
    return (-1);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);
  nodedb_entry.waiting_msgs++;

  new_message = malloc(sizeof(struct message_queue_t));
  strcpy(new_message->filename, filename);

  new_message->next = nodedb_entry.message_queue;
  nodedb_entry.message_queue = new_message;

  free(value.dptr);

  value = construct_gdbm_entry(&nodedb_entry);
  gdbm_store(nodedb, key, value, GDBM_REPLACE);

  free(value.dptr);
  free_node(nodedb_entry);

  return (nodedb_entry.waiting_msgs);
}


int node_get_waiting_msgs(char *hPK) {
  /* look it up, return number of waiting msgs */
  /* return -1 if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return (-1);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free_node(nodedb_entry);

  return (nodedb_entry.waiting_msgs);
}


int node_set_statistics(char *hPK, int statistics) {
  /* look it up, set the statistics */
  /* return -1 if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return (-1);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);
  nodedb_entry.statistics = statistics;

  free(value.dptr);
  value = construct_gdbm_entry(&nodedb_entry);
  gdbm_store(nodedb, key, value, GDBM_REPLACE);

  free(value.dptr);
  free_node(nodedb_entry);

  return (1);
}


int node_get_statistics(char *hPK) {
  /* look it up, return statistics */
  /* return -1 if it's not there */
  datum key, value;
  struct nodedb_entry_t nodedb_entry;

  if (!hPK) return (-1);
  key.dptr = hPK;
  key.dsize = strlen(hPK)+1;

  value = gdbm_fetch(nodedb, key);
  if(!value.dptr) {
    return (-1);
  }

  nodedb_entry = reconstruct_nodedb_entry(value.dptr);

  free(value.dptr);
  free_node(nodedb_entry);

  return (nodedb_entry.statistics);
}


void free_node_msgs(struct nodedb_entry_t nodedb_entry) {

  struct message_queue_t *message, *next_message;

  message = nodedb_entry.message_queue;
  while (message) {
    next_message = message->next;
    free(message);
    message = next_message;
  }

}


void close_nodedb() {
  gdbm_close(nodedb);
}
```