Freenet: A Distributed Anonymous Information Storage and Retrieval System

Ian Clarke
124C Lyham Road, Clapham Park
London SW2 5QA
United Kingdom
i.clarke@dynamicblue.com

Oskar Sandberg
Mörbydalen 12
18252 Stockholm
Sweden
md98-osa@nada.kth.se

Brandon Wiley
2305 Rio Grande St.
Austin, TX 78705
USA
blanu@uts.cc.utexas.edu

Theodore W. Hong [*]
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
United Kingdom
t.hong@doc.ic.ac.uk

July 1, 2000

Abstract

We describe Freenet, a peer-to-peer network application that permits the publication, replication, and retrieval of data while protecting the anonymity of both authors and readers. Freenet operates as a network of identical nodes that collectively pool their storage space to store data files, and cooperate to route requests to the most likely physical location of data. No broadcast search or centralized location index is employed. Files are referred to in a location-independent manner, and are dynamically replicated in locations near requestors and deleted from locations where there is no interest. It is infeasible to discover the true origin or destination of a file passing through the network, and difficult for a node operator to determine or be held responsible for the actual physical contents of her own node.

1 Introduction

Computer networks are rapidly growing in importance as a medium for the storage and exchange of information. However, current systems afford little privacy to their users, and typically store any given data item in only one or a few fixed places, creating a central point of failure. Because of a continued desire among individuals to protect the privacy of their authorship or readership of various types of sensitive information[24], and the undesirability of central points of failure

[*] Supported by grants from the Marshall Aid Commemoration Commission and the National Science Foundation.

which can be attacked by opponents wishing to remove data from the system[10, 23] or simply overloaded by too much interest[1], systems offering greater security and reliability are needed.

We are developing Freenet, a distributed information storage and retrieval system designed to address these concerns of privacy and availability. The system operates as a location-independent distributed file system across many individual computers that allows files to be inserted, stored, and requested anonymously. There are five main design goals:

- Anonymity for both producers and consumers of information
- Deniability for storers of information
- Resistance to attempts by third parties to deny access to information
- Efficient dynamic storage and routing of information
- Decentralization of all network functions

The system is designed to respond adaptively to usage patterns, transparently moving, replicating, and deleting files as necessary to provide efficient service without resorting to broadcast searches or centralized location indexes. It is not intended to guarantee permanent file storage, although it is hoped that enough nodes will join with enough storage capacity that most files will be able to remain indefinitely. In addition, the system operates at the application layer and assumes the existence of a secure transport layer, although it is transport-independent. It does not seek to provide anonymity for general network usage, only for Freenet file transactions.

Freenet is currently being developed as a free software project on Sourceforge, and a preliminary implementation can be downloaded from `http://freenet.sourceforge.net/`. It grew out of work originally done by the first author at the University of Edinburgh[11].

2 Related work

Several strands of related work in this area can be distinguished. Anonymous point-to-point channels based on Chaum's mix-net scheme[7] have been implemented for email by the Mixmaster remailer[19] and for general TCP/IP traffic by onion routing[17] and Freedom[27]. Such channels are not in themselves easily suited to one-to-many publication, however, and are best viewed as a complement to Freenet since they do not provide file access and storage.

Anonymity for consumers of information in the web context is provided by browser proxy services such as the Anonymizer[5], although they provide no protection for producers of information and do not protect consumers against logs kept by the services themselves. Private information retrieval schemes[9] provide much stronger guarantees for information consumers, but only to the extent of hiding which piece of information was retrieved from a particular server. In most cases, the fact of contacting a particular server in itself reveals much about the information retrieved, which can only be counteracted by having each server hold all information (naturally this scales poorly). The closest work to our own is Reiter and Rubin's Crowds system[21], which uses a similar method of proxying requests for consumers, although Crowds does not itself store information and does not protect information producers. Berthold *et al.* propose Web Mixes[6], a stronger system which uses message padding and reordering and dummy messages to increase security, but again does not protect information producers.

The Rewebber[22] provides a measure of anonymity for producers of web information by means of an encrypted URL service which is essentially the inverse of an anonymizing browser proxy, but has the same difficulty of providing no protection against the operator of the service itself. TAZ[16] extends this idea by using chains of nested encrypted URLs which successively point to different rewebber servers to be contacted, although this is vulnerable to traffic analysis using replay. Both rely on a single server as the ultimate source of information. Publius[26] enhances availability by distributing files as redundant shares among n webservers, only k of which are needed to reconstruct a file; however, since the identity of the servers themselves is not anonymized, an attacker might remove information by forcing the closure of n–k+1 servers. The Eternity proposal[4] seeks to archive information permanently and anonymously, although it lacks specifics on how to efficiently locate stored files, making it more akin to an anonymous backup service. Free Haven[12] is an interesting anonymous publication system which uses a trust network and file trading mechanism to provide greater server accountability while maintaining anonymity.

`distributed.net`[13] demonstrated the concept of pooling computer resources among multiple users on a large scale for CPU cycles; other systems which do the same for disk space are Napster[20] and Gnutella[15], although the former relies on a central server to locate files and the latter employs an inefficient broadcast search. Neither one replicates files. Intermemory[8] and India[14] are cooperative distributed fileserver systems intended for long-term archival storage along the lines of Eternity, in which files are split into redundant shares and distributed among many participants. Akamai[2] provides a service which replicates files at locations near information consumers, but is not suitable for producers who are individuals (as opposed to corporations). None of these systems attempt to provide anonymity.

3 Architecture

Freenet is implemented as a peer-to-peer network of nodes that query one another to store and retrieve data files, which are named by location-independent keys. Each node maintains its own local datastore which it makes available to the network for reading and writing, as well as a dynamic routing table containing addresses of other nodes and the keys that they are thought to hold. It is intended that most users of the system will run nodes, both to provide security guarantees against inadvertently using a hostile foreign node and to increase the storage capacity available to the network as a whole.

The system can be regarded as a cooperative distributed filesystem incorporating location independence and transparent lazy replication. Just as systems such as `distributed.net`[13] enable ordinary users to share unused CPU cycles on their machines, Freenet enables users to share unused disk space. However, where `distributed.net` uses those CPU cycles for its own purposes, Freenet is directly useful to users themselves, acting as an extension to their own hard drives.

The basic model is that queries are passed along from node to node in a chain of proxy requests, with each node making a local routing decision in the style of IP routing about where to send the query next (this varies from query to query). Nodes know only their immediate upstream and downstream neighbors in the chain. Each query is given a "hops-to-live" count which is decremented at each node to prevent infinite chains (analogous to IP's time-to-live). Each query is also assigned a pseudo-unique random identifier, so that nodes can prevent loops by rejecting queries they have seen before. When this happens, the immediately preceding node simply chooses a different node to forward to. This process continues until the query is either satisfied or exceeds its hops-to-live limit. Then, the success or failure result is passed back up the chain to the sending node.

No node is privileged over any other node, so no hierarchy or central point of failure exists. Joining the network is simply a matter of discovering the address of one or more existing nodes through out-of-band means, then starting to send messages. Files are immutable at present, although file updatability is on the agenda for future releases. In addition, the namespace is currently flat, consisting of the space of 160 bit SHA-1[3] hashes of descriptive text strings, although support for a richer namespace is a priority for development. See section 5 for further discussion of updating and naming.

3.1 Retrieving data

To retrieve data, a user hashes a short descriptive string (for example, `text/philosophy/sun-tzu/art-of-war`) to obtain a file key. (See section 5 for details on how descriptive strings corresponding to files can be discovered.) She then sends a request message to her own node specifying that key and a hops-to-live value. When a node receives a request, it first checks its own store for the data and returns it if found, together with a note saying it was the source of the data. If not found, it looks up the nearest key in its routing table to the key requested and forwards the request to the corresponding node. If that request is ultimately successful and returns with the data, the node will pass the data back to the upstream requestor, cache the file in its own datastore, and create a new entry in its routing table associating the actual data source with the requested key. A subsequent request for the same key will be immediately satisfied from the local cache; a request for a "similar" key (determined by lexicographic distance) will be forwarded to the previously successful data source. Because maintaining a table of data sources is a potential security concern, any node along the way can unilaterally decide to change the reply message to claim itself or another arbitrarily-chosen node as the data source.

If a node cannot forward a request to its preferred downstream node because the target is down or a loop would be created, the node having the second-nearest key will be tried, then the third-nearest, and so on. If a node runs out of candidates to try, it reports failure back to its upstream neighbor, which will then try *its* second choice, etc. In this way, a request operates as a steepest-ascent hill-climbing search with backtracking. If the hops-to-live count is exceeded, a failure result is propagated back to the original requestor without any further nodes being tried. Nodes may unilaterally curtail excessive hops-to-live values to reduce network load. They may also forget about pending requests after a period of time to keep message memory free.

Figure 1 depicts a typical sequence of request messages. The user initiates a request at node a. Node a forwards the request to node b, which forwards it to node c. Node c is unable to contact any other nodes and returns a backtracking "request failed" message to b. Node b then tries its second choice, e, which forwards the request to f. Node f forwards the request to b,

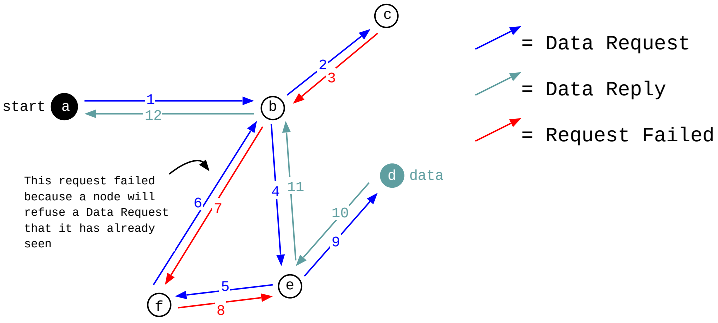
start
a
b
c
d
data
e
f
1
2
3
4
5
6
7
8
9
10
11
12
This request failed because a node will refuse a Data Request that it has already seen
= Data Request
= Data Reply
= Request Failed

Figure 1: A typical request sequence.

which detects the loop and returns a backtracking failure message. Node f is unable to contact any other nodes and backtracks one step further back to e. Node e forwards the request to its second choice, d, which has the data. The data is returned from d via e and b back to a, which sends it back to the user. The data is also cached on e, b, and a.

This mechanism has a number of effects. Most importantly, we hypothesize that the quality of the routing should improve over time, for two reasons. First, nodes should come to specialize in locating sets of similar keys. If a node is listed in routing tables under a particular key, it will tend to receive mostly requests for keys similar to that key. It is therefore likely to gain more "experience" in answering those queries and become better informed in its routing tables about which other nodes carry those keys. Second, nodes should become similarly specialized in storing clusters of files having similar keys. Because forwarding a request successfully will result in the node itself gaining a copy of the requested file, and most requests will be for similar keys, the node will mostly acquire files with similar keys. Taken together, these two effects should improve the efficiency of future requests in a self-reinforcing cycle, as nodes build up routing tables and datastores focusing on particular sets of keys, which will be precisely those keys that they are asked about.

In addition, the request mechanism will cause popular data to be transparently replicated by the system and mirrored closer to requestors. For example, if a file that is originally located in London is requested in Berkeley, it will become cached locally and provide faster response to subsequent Berkeley requests. It also becomes copied onto each computer along the way, providing redundancy if the London node fails or is shut down. (Note that "along the way" is determined by key closeness and does not necessarily have geographical relevance.)

Finally, as nodes process requests, they create new routing table entries for previously-unknown nodes that supply files, increasing connectivity. This helps new nodes to discover more of the network (although it does not help the rest of the network to discover *them*; for that, performing inserts is necessary). Note that direct links are created, bypassing the intermediate nodes used. Thus, nodes that successfully supply data will gain routing table entries and be contacted more often than nodes that do not.

Because hashes are used as keys, lexicographic closeness of keys does not imply any closeness of the original descriptive strings and presumably, no closeness of subject matter of the files.

This lack of semantic closeness is not important, however, as the routing algorithm is based on knowing where keys are located, not where subjects are located. That is, supposing `text/philosophy/sun-tzu/art-of-war` hashes to `AH5JK2`, requests for this file can be routed more effectively by creating clusters containing `AH5JK1`, `AH5JK2`, and `AH5JK3`, not by creating clusters for works of philosophy. Indeed, the use of a hash is desirable precisely because philosophical works will be scattered across the network, lessening the chances that failure of a single node will make all philosophy unavailable.

3.2 Storing data

Inserts follow a parallel strategy to requests. To insert data, a user picks an appropriate descriptive text string and hashes it to create a file key. She then sends an insert message to her own node specifying the proposed key and a hops-to-live value (this will determine the number of nodes to store it on). When a node receives an insert proposal, it first checks its own store to see if the key is already taken. If the key is found, the node returns the pre-existing file as if a request had been made for it. The user will thus know that a collision was encountered and can try again using a different key, i.e. different descriptive text. (Although potentially the use of a hash might cause extra collisions, in practice a high-quality hash of sufficient length should only cause collisions if the same initial descriptive text was chosen.) If the key is not found, the node looks up the nearest key in its routing table to the key proposed and forwards the insert to the corresponding node. If that insert causes a collision and returns with the data, the node will pass the data back to the upstream inserter and again behave as if a request had been made (i.e. cache the file locally and create a routing table entry for the data source).

If the hops-to-live limit is reached without a key collision being detected, an "all clear" result will be propagated back to the original inserter. Note that for inserts, this is a successful result, in contrast to the request case. The user then sends the data to insert, which will be propagated along the path established by the initial query and stored in each node along the way. Each node will also create an entry in its routing table associating the inserter (as the data source) with the new key. To avoid the obvious security problem, any node along the way can unilaterally decide to change the insert message to claim itself or another arbitrarily-chosen node as the data source.

If a node cannot forward an insert to its preferred downstream node because the target is down or a loop would be created, the insert backtracks to the second-nearest key, then the third-nearest, and so on in the same way as for requests. If the backtracking returns all the way back to the original inserter, it indicates that fewer nodes than asked for could be contacted. As with requests, nodes may curtail excessive hops-to-live values and/or forget about pending inserts after a period of time.

This mechanism has three effects. First, newly inserted files are selectively placed on nodes already possessing files with similar keys. This reinforces the clustering of keys set up by the request mechanism. Second, new nodes can tell the rest of the network of their existence by inserting data. Third, an attempt by an attacker to supplant an existing file by deliberate insert collision (e.g., by inserting a corrupted or empty file under the same key) is likely to simply spread the real file further, since the original file is propagated back on collision. (The use of a content-hash key, described in section 5, makes such an attack infeasible altogether.)

3.3 Managing data

All information storage systems must deal with the problem of finite storage capacity. Individual Freenet node operators can configure the amount of storage to dedicate to their datastores. Node storage is managed as an LRU (Least Recently Used) cache[25], with data items kept sorted in decreasing order by time of most recent request (or time of insert, if an item has never been requested). When a new file arrives (from either a new insert or a successful request) which would cause the datastore to exceed the designated size, the least recently used files are evicted in order until there is room. The impact on availability is mitigated somewhat by the fact that the routing table entries created when the evicted files first arrived will remain for a time, potentially allowing the node to later get new copies from the original data sources. (Routing table entries are also eventually deleted in a similar fashion as the table fills up, although they will be retained longer since they are smaller.)

Strictly speaking, the datastore is not a cache, since the set of datastores is all the storage that there is, i.e. there is no “permanent” copy which is being duplicated. Once all the nodes have decided, collectively speaking, to drop a particular file, it will no longer be available to the network. In this respect, Freenet differs from the Eternity service, which seeks to provide guarantees of file lifetime.

This mechanism has an advantageous side, however, since it allows outdated documents to fade away after being superseded by newer documents, thus alleviating some of the problem of immutable files. If an outdated document is still used and considered valuable for historical reasons, it will stay alive precisely as long as it continues to be requested.

For political or legal reasons, it may be desirable for node operators not to explicitly know the contents of their datastores. Therefore, it is recommended that all inserted files be encrypted by their original unhashed descriptive text strings in order to obscure their contents. Of course, this does not secure the file—that would be impossible since a requestor (potentially anyone) must be capable of decrypting the file once retrieved. Rather, the objective is that the node operator can plausibly deny any knowledge of the contents of her datastore, since all she knows *a priori* is the hashed key and its associated encrypted file. The hash cannot feasibly be reversed to reveal the unhashed description and decrypt the file. With effort, of course, a dictionary attack will reveal which keys are present—as it must in order for requests to work at all.

4 Protocol details

The Freenet protocol is packet-oriented and uses self-contained messages. Each message includes a transaction ID so that nodes can track the state of inserts and requests. This design is intended to permit flexibility in the choice of transport mechanisms for messages, whether they be TCP, UDP, or other technologies such as packet radio. For efficiency, nodes are also able to send multiple messages over a persistent channel such as a TCP connection, if available. Node addresses consist of a transport method plus a transport-specific identifier (such as an IP address and port number), e.g. `tcp/192.168.1.1:19114`.

A Freenet transaction begins with a Request.Handshake message from one node to another, specifying the desired return address of the sending[1] node. (The return address may be impossible to determine from the transport layer alone, or may use a different transport from that used to send the message.) If the remote node is active and responding to requests, it will reply with a Reply.Handshake specifying the protocol version number that it understands. Handshakes are remembered for a few hours, and subsequent transactions between the same nodes during this time may omit this step.

All messages contain a randomly-generated 64-bit transaction ID, a hops-to-live counter, and a depth counter. Although the ID cannot be guaranteed to be unique, the likelihood of a collision occurring during the transaction lifetime among the limited set of nodes that it sees is extremely low. Hops-to-live is set by the originator of a message and is decremented at each hop to prevent messages being forwarded indefinitely. To reduce the information that an attacker can obtain from the hops-to-live value, messages do not automatically terminate after hops-to-live reaches 1 but are forwarded on with finite probability (with hops-to-live again 1). Depth is incremented at each hop and is used by a replying node to set hops-to-live high enough to reach a requestor. Requestors should initialize it to a small random value to obscure their location. As with hops-to-live, a depth of 1 is not automatically incremented but is passed unchanged with finite probability.

To request data, the sending node sends a Request.Data message specifying a transaction ID, initial hops-to-live and depth, and a search key. The remote node will check its datastore for the key and if not found, will forward the request to another node as described in section 3.1. Using the chosen hops-to-live count, the sending node starts a timer for the expected amount of time it should take to contact that many nodes, after which it will assume failure. While the request is being processed, the remote node may periodically send back Reply.Restart messages indicating that messages were stalled waiting on network timeouts, so that the sending node knows to extend its timer.

If the request is ultimately successful, the remote node will reply with a Send.Data message containing the data requested and the address of the node which supplied it (possibly faked). If the request is ultimately unsuccessful and its hops-to-live are completely used up trying to satisfy it, the remote node will reply with a Reply.NotFound. The sending node will then decrement the hops-to-live of the Send.Data (or Reply.NotFound) and pass it along upstream, unless it is the actual originator of the request. Both of these messages terminate the transaction and release any resources held. However, if there are still hops-to-live remaining, usually because the request ran into a dead end where no viable non-looping paths could be found, the remote node will reply with a Request.Continue giving the number of hops-to-live left. The sending node will then try to contact the next-most likely node from its routing table. It will also send a Reply.Restart upstream.

To insert data, the sending node sends a Request.Insert message specifying a randomly-generated transaction ID, an initial hops-to-live and depth, and a proposed key. The remote node will check its datastore for the key and if not found, forward the insert to another node as described in section 3.2. Timers and Reply.Restart messages are also used in the same way as for requests.

If the insert ultimately results in a key collision, the remote node will reply with either

[1] Note that the sending node may not be the original requestor.

a Send.Data message containing the existing data or a Reply.NotFound (if existing data was not actually found, but routing table references to it were). If the insert does not encounter a collision, yet runs out of nodes with nonzero hops-to-live remaining, the remote node will reply with a Request.Continue. In this case, Request.Continue is a failure result meaning that not as many nodes could be contacted as asked for. These messages will be passed along upstream as in the request case. Both messages terminate the transaction and release any resources held. However, if the insert expires without encountering a collision, the remote node will reply with a Reply.Insert, indicating that the insert can go ahead. The sending node will pass along the Reply.Insert upstream and wait for its predecessor to send a Send.Insert containing the data. When it receives the data, it will store it locally and forward the Send.Insert downstream, concluding the transaction.

5 Naming, searching, and updating

Freenet's basic flat namespace has obvious disadvantages in terms of discovering documents, name collisions, etc. Several mechanisms of providing more structure within the current scheme are possible. For example, directory-like documents containing hypertext pointers to other files could be created. A directory file under the key `text/philosophy` could contain a list of keys such as `text/philosophy/sun-tzu/art-of-war`, `text/philosophy/confucius/analects`, and `text/philosophy/nozick/anarchy-state-utopia`, using appropriate syntax interpretable by a client. The difficulty, however, is in deciding how to permit changes to the directory to update entries, while preventing the directory from being corrupted or spammed. An alternative mechanism is to encourage individuals to maintain and share their own compilations of keys—subjective bookmark lists, rather than authoritative directories. This is the approach in common use on the world-wide web.

Name collisions are still a problem with both bookmark lists and directories. One way of addressing collisions is to introduce a two-level structure similar to that used by most traditional file systems. Real files could be stored under a pseudo-unique binary key, such as a hash of the files' contents (a *content-hash key*). Users would access these files by first retrieving an indirect file stored under a semantically meaningful name. The indirect file would consist solely of a list of binary keys corresponding to that name (possibly only one), along with other information useful for differentiating among the possible choices, such as author, creation time, and endorsements by other users. Content-hash keys would also protect against files being maliciously tampered with or replaced. However, indirect files are essentially like low-level directories, and share the same problem of managing updates. Another approach is to skip the indirect files altogether and have bookmark lists pointing to content-hash keys, rather than names.

Introducing a search capability in conjunction with binary keys offers a way to side-step the need to maintain directories. The most straightforward way to add search is to run a hypertext spider such as those used to search the web. While an attractive solution in many ways, this conflicts with the design goal of avoiding centralization. A possible alternative is to create a special class of lightweight indirect files. When a real file is inserted, the author could also insert a number of indirect files containing a single pointer to the real file, named according to search keywords chosen by her. These indirect files would differ from normal files in that collisions would be permitted on insert, and requests for an indirect file key (i.e. a keyword) would keep going until a specified number of indirect files (i.e. search results) were accumulated. Managing the likely large volume of these indirect files is an open problem.

Updates by a single user can be handled in a reasonably straightforward manner by using a variation on indirection. When an author inserts a file which she later intends to update, she first generates a public-private key pair and signs the file with the private key. The file is inserted under a binary key, but instead of using the hash of the file's contents, the literal public key itself is used (a *signature-verifying key*). As with inserting under a content-hash key, inserting under a signature-verifying key provides a pseudo-unique binary key for a file, which can also be used to verify that the file's contents have not been tampered with. To update the file, the new version is signed by the private key and inserted under the public signature-verifying key. When the insert reaches a node which possesses the old version, a key collision will occur. The node will check the signature on the new version, verify that it is both valid and more recent, and replace the old version.

In order to allow old versions of files to remain in existence for historical reasons and to prevent the possibility of authors being compelled to "update" their own files out of existence, an additional level of indirection can be used. In this scheme, the real file is inserted under its content-hash key. Then, an indirect file containing a pointer to that content-hash key is created, but inserted under a signature-verifying key. This signature-verifying key is given out as the binary key for the file and entered, for example, in directories and bookmark lists (making them double indirect files). When the author creates a new version, it is inserted under its own content-hash key, distinct from the old version's key. The signature-verified indirect file is then updated to point to the new version (or possibly to keep pointers to all versions). Thus the signature-verifying key will always lead to the most recent version of the file, while old versions can continue to be accessed by content-hash key if desired. (If not requested, however, these old versions will eventually disappear like any other unused file.)

For large files, splitting files into multiple parts is desirable because of storage and bandwidth limitations. Splitting even medium files into standard-sized parts (e.g. 16 kilobytes) also has advantages in combating traffic analysis. This is easily accomplished by inserting each part separately under a content-hash key, and including pointers to the other parts.

Combining all these ideas together, a user might look through a bookmark list or perform a search on a keyword to get a list of signature-verifying binary keys for files dealing with a particular topic. Retrieving one of these keys gives an indirect file containing a set of content-hash keys corresponding to different versions, ordered by date. Retrieving the most recent content-hash key gives the first part of a multipart file, together with pointers to two other content-hash keys. Finally, retrieving those last two content-hash keys and concatenating the three parts together yields the desired file.

6 Performance simulation

Simulations were carried out on an early version of this system to give some indications about its performance. Here we summarize the most important results; for full details, see [11].

The scenario for these simulations was a network with between 500 and 900 nodes. Each

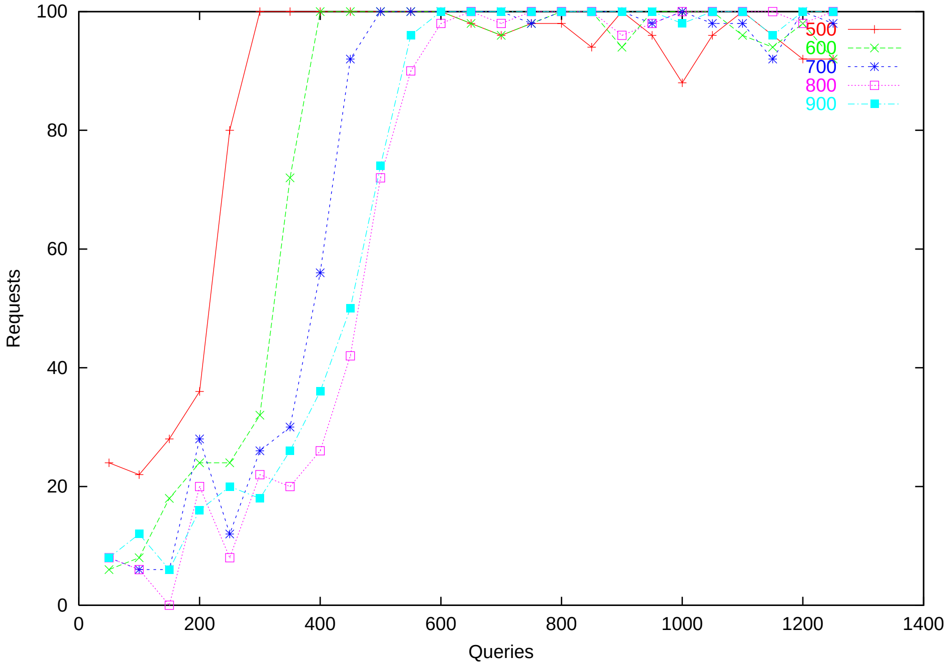

Figure 2: Percentage of successful requests over time.

node had a datastore size of 40 items and a routing table size of 50 addresses (relatively low, because of limitations on available hardware), and was given 10 unique items to store locally. The network was initially connected in a linear fashion, where each node started with routing references to one node on either side.

6.1 Retrieval success rate

Queries for random keys were sent to random nodes in the network, in batches of 50 parallel queries at a time, and the percentage of successful requests recorded over time. Figure 2 shows the percentage of successful requests versus the total number of queries since initialization, for several network sizes. We can see that the initially low success rate rises rapidly until over 95% of requests are successful. The number of queries until network convergence is approximately half the total size of the network.

6.2 Retrieval time

Queries for random keys were sent to random nodes in the network, in batches of 50 parallel queries at a time, and the average number of hops needed for a successful request recorded over time. Figure 3 shows the number of request hops versus the total number of queries since initialization, for several network sizes. We can see that the initially high number of hops

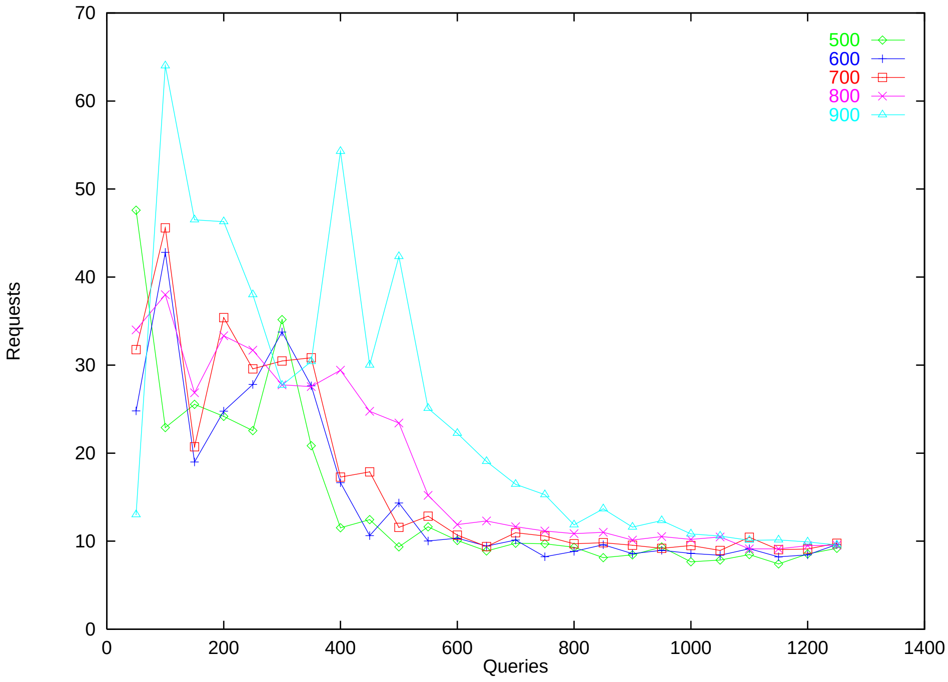

Figure 3: Number of hops per request over time.

required drops until it stabilizes at approximately 10 hops. This value changes remarkably little with network size, suggesting that the time required for requests should scale quite well. The number of queries until network convergence is approximately equal to the total size of the network.

7 Security

The primary goal for Freenet security is protecting the anonymity of requestors and inserters of files. It is also important to protect the identity of storers of files. Although trivially anyone can turn a node into a storer by requesting a file through it, thus “identifying” it as a storer, what is important is that there remain other, unidentified, holders of the file so that an adversary cannot remove a file by attacking all of the nodes that hold it. Files must be protected against malicious modification, and finally, the system must be resistant to denial-of-service attacks.

Reiter and Rubin[21] present a useful taxonomy of anonymous communication properties on three axes. The first axis is the type of anonymity: sender anonymity or receiver anonymity, which mean respectively that an adversary cannot determine either who originated a message, or to whom it was sent. The second axis is the adversary in question: a local eavesdropper, a malicious node or collaboration of malicious nodes, or a web server (not applicable to Freenet). The third axis is the degree of anonymity, which ranges from absolute privacy (the presence of

System	Attacker	Sender anonymity	Key anonymity
Basic Freenet	local eavesdropper	exposed	exposed
	collaborating nodes	beyond suspicion	exposed
Freenet + pre-routing	local eavesdropper	exposed	beyond suspicion
	collaborating nodes	beyond suspicion	exposed

Table 1: Anonymity properties of Freenet.

communication cannot be perceived) to beyond suspicion (the sender appears no more likely to have originated the message than any other potential sender), probable innocence (the sender is no more likely to be the originator than not), possible innocence, exposed, and provably exposed (the adversary can prove to others who the sender was).

As Freenet communication is not directed towards specific receivers, receiver anonymity is more accurately viewed as key anonymity, that is, hiding the key which is being requested or inserted. Unfortunately, since routing depends on knowledge of the key, key anonymity is not possible in the basic Freenet scheme (but see the discussion of "pre-routing" below). The use of hashes as keys provides a measure of obscurity against casual eavesdropping, but is of course vulnerable to a dictionary attack since unhashed keys must be widely known in order to be useful.

Freenet's anonymity properties under this taxonomy are shown in Table 1. Against a collaboration of malicious nodes, sender anonymity is preserved beyond suspicion since a node in a request path cannot tell whether its predecessor in the path initiated the request or is merely forwarding it. [21] describes a probabilistic attack which might compromise sender anonymity, using a statistical analysis of the probability that a request arriving at a node a is forwarded on or handled directly, and the probability that a chooses a particular node b to forward to. This analysis is not immediately applicable to Freenet, however, since request paths are not constructed probabilistically. Forwarding depends on whether or not a has the requested data in its datastore, rather than chance. If a request is forwarded, the routing tables determine where it is sent to, and could be such that a forwards every request to b, or never forwards any requests to b, or anywhere in between. Nevertheless, the depth value may provide some indication as to how many hops away the originator was, although this is obscured by the random selection of an initial depth and the probabilistic means of incrementing it (see section 4). Similar considerations apply to hops-to-live. Further investigation is required to clarify these issues.

Against a local eavesdropper there is no protection on messages between the user and the first node contacted. Since the first node contacted can act as a local eavesdropper, it is recommended that the user only use a node on her own machine as the first point of entry into the Freenet network. Messages between nodes are encrypted against local eavesdropping, although traffic analysis may still be performed (e.g. an eavesdropper may observe a message going out without a previous message coming in and conclude that the target originated it).

Key anonymity and stronger sender anonymity can be achieved by adding mix-style "pre-routing" of messages. In this scheme, basic Freenet messages are encrypted by a succession of public keys which determine the route that the encrypted message will follow (overriding the normal routing mechanism). Nodes along this portion of the route are unable to determine either the originator of the message or its contents (including the request key), as per the mix-net anonymity properties. When the message reaches the endpoint of the pre-routing phase, it will be injected into the normal Freenet network and behave as though the endpoint were the originator of the message.

Protection for data sources is provided by the occasional resetting of the data source field in replies. The fact that a node is listed as the data source for a particular key does not necessarily imply that it actually supplied that data, or was even contacted in the course of the request. It is not possible to tell whether the downstream node provided the file or was merely forwarding a reply sent by someone else. In fact, the very act of successfully requesting a file places it on the downstream node if it was not already there, so a subsequent examination of that node on suspicion reveals nothing about the prior state of affairs, and provides a plausible legal ground that the data was not there until the act of investigation placed it there. Requesting a particular file with a hops-to-live of 1 does not directly reveal whether or not the node was previously storing the file in question, since nodes continue to forward messages having hops-to-live of 1 with finite probability. The success of a large number of requests for related files, however, may provide grounds for suspicion that those files were being stored there previously.

Modification or outright replacement of files by a hostile node is an important threat, and not only because of the corruption of the file itself. Since routing tables are based on replies to requests, a node might attempt to steer traffic towards itself by pretending to have files when it does not and simply returning fictitious data. For data stored under content-hash keys or signature-verifying keys, this is not feasible since inauthentic data can be detected unless a node finds a hash collision or successfully forges a cryptographic signature. Data stored under ordinary descriptive text keys, however, is vulnerable. Some protection is afforded by the expectation that files will be encrypted by the descriptive text, since the node must respond to a hashed key request with data encrypted by the original text key, but a dictionary attack is possible using a table of text keys and their hashes. Even partial dictionaries cause problems since the node can behave normally when an unknown key is requested and forge data when the key is known.

Finally, a number of denial-of-service attacks can be envisioned. The most significant threat is that an attacker will attempt to fill all of the network's storage capacity by inserting a large number of garbage files. An interesting possibility for countering this attack is the Hash Cash scheme[18]. Essentially, the scheme requires the inserter to perform a lengthy computation as "payment" before an insert is accepted, thus slowing down an attack. Another alternative is to divide the datastore into two sections, one for new inserts and one for "established" files (defined as files having received at least a certain number of requests). New inserts can only displace other new inserts, not established files. In this way a flood of garbage inserts might temporarily paralyze insert operations but would not displace existing files. It is difficult for an attacker to artificially legitimize her own (garbage) files by requesting them many times since her requests will be satisfied by the first node to hold the data and not proceed any further. She cannot send requests directly to the other downstream nodes holding her files since their identities are hidden from her. However, adopting this scheme may make it difficult for genuine new inserts to survive long enough to be requested by others and become established.

Attackers may attempt to replace existing files by inserting alternate versions under the same keys. Such an attack is not possible against a content-hash key or signature-verifying key, since it requires finding a hash collision or successfully forging a cryptographic signature. An attack against a descriptive text key, on the other hand, may result in both versions coexisting in the network. The way in which nodes react to insert collisions (detailed in section 3.2) is intended to make such attacks more difficult. The success of a replacement attack can be measured by the ratio of corrupt versus genuine versions resulting in the system. However, the more corrupt copies the attacker attempts to circulate (by setting a higher hops-to-live on insert), the greater the chance that an insert collision will be encountered, which would cause an increase in the number of genuine copies.

8 Conclusions

The Freenet network provides an effective means of anonymous information storage and retrieval. By using cooperating nodes spread over many computers in conjunction with an efficient routing algorithm, it keeps information anonymous and available while remaining highly scalable. Initial deployment of a test version is underway, and is so far proving successful, with over 15,000 copies downloaded and many interesting files in circulation. Because of the nature of the system, it is impossible to tell exactly how many users there are or how well the insert and request mechanisms are working, but anecdotal evidence is so far positive. We are working on implementing a simulation and visualization suite which will enable more rigorous tests of the protocol and routing algorithm. More realistic simulation is necessary which models the effects of inserts taking place alongside requests, nodes joining and leaving, variation in node capacity, and larger network sizes.

9 Acknowledgements

Portions of this material are based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] S. Adler, "The Slashdot effect: an analysis of three Internet publications," `http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html` (2000).

[2] Akamai, `http://www.akamai.com/` (2000).

[3] American National Standards Institute, Draft: American National Standard X9.30-199X: *Public-Key Cryptography Using Irreversible Algorithms for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA).* American Bankers Association (1993).

[4] R.J. Anderson, "The Eternity service," in *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology (PRAGOCRYPT '96)*, Prague, Czech Republic (1996).

[5] Anonymizer, `http://www.anonymizer.com/` (2000).

[6] O. Berthold, H. Federrath, and M. Köhntopp, "Project 'Anonymity and unobservability in the Internet'," in *Computers Freedom and Privacy Conference 2000 (CFP 2000) Workshop on Freedom and Privacy by Design* (to appear).

[7] D.L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM* **24**(2), 84-88 (1981).

[8] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos, "A prototype implementation of archival intermemory," in *Proceedings of the Fourth ACM Conference on Digital Libraries (DL '99)*, Berkeley, CA, USA. ACM Press: New York (1999).

[9] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," *Journal of the ACM* **45**(6), 965-982 (1998).

[10] Church of Spiritual Technology (Scientology) v. Dataweb *et al.*, Cause No. 96/1048, District Court of the Hague, The Netherlands (1999).

[11] I. Clarke, "A distributed decentralised information storage and retrieval system," unpublished report, Division of Informatics, University of Edinburgh (1999). Available at `http://freenet.sourceforge.net/` (2000).

[12] R.R. Dingledine, "The Free Haven Project: Design and Deployment of an Anonymous Secure Data Haven," M.Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2000). Available at `http://www.freehaven.net/` (2000).

[13] Distributed.net, `http://www.distributed.net/` (2000).

[14] D.J. Ellard, J.M. Megquier, and L. Park, "The INDIA protocol," `http://www.eecs.harvard.edu/~ellard/India-WWW/` (2000).

[15] Gnutella, `http://gnutella.wego.com/` (2000).

[16] I. Goldberg and D. 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` (2000).

[17] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing for anonymous and private Internet connections," *Communications of the ACM* **42**(2), 39-41 (1999).

[18] Hash Cash, `http://www.cypherspace.org/~adam/hashcash/` (2000).

[19] Mixmaster, `http://www.obscura.com/~loki/remailer/mixmaster-faq.html` (2000).

[20] Napster, `http://www.napster.com/` (2000).

[21] M.K. Reiter and A.D. Rubin, "Anonymous web transactions with Crowds," *Communications of the ACM* **42**(2), 32-38 (1999).

[22] The Rewebber, `http://www.rewebber.de/` (2000).

[23] M. Richtel and S. Robinson, “Several web sites are attacked on day after assault shut Yahoo,” *The New York Times*, February 9, 2000.

[24] J. Rosen, “The eroded self,” *The New York Times*, April 30, 2000.

[25] A.S. Tanenbaum, *Modern Operating Systems.* Prentice-Hall: Upper Saddle River, NJ, USA (1992).

[26] M. Waldman, A.D. Rubin, and L.F. Cranor, “Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system,” in *Ninth USENIX Security Symposium*, Denver, CO, USA (to appear).

[27] Zero-Knowledge Systems, `http://www.zks.net/` (2000).