Freenet: A Distributed Anonymous Information Storage and Retrieval System

Ian Clarke[1], Oskar Sandberg[2], Brandon Wiley[3], and Theodore W. Hong[4*]

[1] Uprizer, Inc., 1007 Montana Avenue #323, Santa Monica, CA 90403, USA
ian@octayne.com
[2] Department of Numerical Analysis and Computer Science, Royal Institute of Technology, SE-100 44 Stockholm, Sweden
md98-osa@nada.kth.se
[3] College of Communication, University of Texas at Austin, Austin, TX 78712, USA
blanu@uts.cc.utexas.edu
[4] 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

Abstract. We describe Freenet, an adaptive 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

Networked computer systems are rapidly growing in importance as the medium of choice 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[28], and the undesirability of central points of failure which can be attacked by opponents wishing to remove data from the system[11, 27] 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

* Work of Theodore W. Hong was supported by grants from the Marshall Aid Commemoration Commission and the National Science Foundation.

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 a sufficient number of 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://www.freenetproject.org/`. It grew out of work originally done by the first author at the University of Edinburgh[12].

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[8] have been implemented for email by the Mixmaster remailer[13] and for general TCP/IP traffic by onion routing[19] and Freedom[32]. 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[6], 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[10] 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 many cases, the fact of contacting a particular server in itself can reveal much about the information retrieved, which can only be counteracted by having every server hold all information (naturally this scales poorly). The closest work to our own is Reiter and Rubin's Crowds system[25], 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[7], a stronger system that uses message padding and reordering and dummy messages to increase security, but again does not protect information producers.

The Rewebber[26] provides a measure of anonymity for producers of web information by means of an encrypted URL service that 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[18] extends this idea by using chains of nested encrypted URLs that 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[30] 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[5] 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[14] is an interesting anonymous publication system that uses a trust network and file trading mechanism to provide greater server accountability while maintaining anonymity.

`distributed.net`[15] 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[24] and Gnutella[17], although the former relies on a central server to locate files and the latter employs an inefficient broadcast search. Neither one replicates files. Intermemory[9] and India[16] 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 that 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 an adaptive 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`[15] 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 requests for keys are passed along from node to node through a chain of proxy requests in which each node makes a local decision about where to send the request next, in the style of IP (Internet Protocol) routing. Depending on the key requested, routes will vary. The routing algorithms for storing and retrieving data described in the following sections are designed to adaptively adjust routes over time to provide efficient performance while using only local, rather than global, knowledge. This is necessary since nodes only have knowledge of their immediate upstream and downstream neighbors in the proxy chain, to maintain privacy.

Each request is given a *hops-to-live* limit, analogous to IP's time-to-live, which is decremented at each node to prevent infinite chains. Each request is also assigned a pseudo-unique random identifier, so that nodes can prevent loops by rejecting requests they have seen before. When this happens, the immediately-preceding node simply chooses a different node to forward to. This process continues until the request 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 first discovering the address of one or more existing nodes through out-of-band means, then starting to send messages.

3.1 Keys and searching

Files in Freenet are identified by binary file keys obtained by applying a hash function. Currently we use the 160-bit SHA-1[4] function as our hash. Three different types of file keys are used, which vary in purpose and in the specifics of how they are constructed.

The simplest type of file key is the *keyword-signed key* (KSK), which is derived from a short descriptive text string chosen by the user when storing a file in the network. For example, a user inserting a treatise on warfare might assign it the description, `text/philosophy/sun-tzu/art-of-war`. This string is used as input to deterministically generate a public/private key pair. The public half is then hashed to yield the file key.

The private half of the asymmetric key pair is used to sign the file, providing a minimal integrity check that a retrieved file matches its file key. Note however that an attacker can use a dictionary attack against this signature by compiling a list of descriptive strings. The file is also encrypted using the descriptive string itself as a key, for reasons to be explained in section 3.4.

To allow others to retrieve the file, the user need only publish the descriptive string. This makes keyword-signed keys easy to remember and communicate to others. However, they form a flat global namespace, which is problematic. Nothing prevents two users from independently choosing the same descriptive string for different files, for example, or from engaging in "key-squatting"—inserting junk files under popular descriptions.

These problems are addressed by the *signed-subspace key* (SSK), which enables personal namespaces. A user creates a namespace by randomly generating a public/private key pair which will serve to identify her namespace. To insert a file, she chooses a short descriptive text string as before. The public namespace key and the descriptive string are hashed independently, XOR'ed together, and then hashed again to yield the file key.

As with the keyword-signed key, the private half of the asymmetric key pair is used to sign the file. This signature, generated from a random key pair, is more secure than the signatures used for keyword-signed keys. The file is also encrypted by the descriptive string as before.

To allow others to retrieve the file, the user publishes the descriptive string together with her subspace's public key. Storing data requires the private key, however, so only the owner of a subspace can add files to it.

The owner now has the ability to manage her own namespace. For example, she could simulate a hierarchical structure by creating directory-like files containing hypertext pointers to other files. A directory 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. Directories can also recursively point to other directories.

The third type of key is the *content-hash key* (CHK), which is useful for implementing updating and splitting. A content-hash key is simply derived by directly hashing the contents of the corresponding file. This gives every file a pseudo-unique file key. Files are also encrypted by a randomly-generated encryption key. To allow others to retrieve the file, the user publishes the content-hash key itself together with the decryption key. Note that the decryption key is never stored with the file but is only published with the file key, for reasons to be explained in section 3.4.

Content-hash keys are most useful in conjunction with signed-subspace keys using an indirection mechanism. To store an updatable file, a user first inserts it under its content-hash key. She then inserts an indirect file under a signed-subspace key whose contents are the content-hash key. This enables others to retrieve the file in two steps, given the signed-subspace key.

To update a file, the owner first inserts a new version under its content-hash key, which should be different from the old version's content hash. She then inserts a new indirect file under the original signed-subspace key pointing to the updated version. 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. Thus the signed-subspace key will lead to the most recent version of the file, while old versions can continue to be accessed directly by content-hash key if desired. (If not requested, however, these old versions will eventually be removed from the network—see section 3.4.) This mechanism can be used to manage directories as well as regular files.

Content-hash keys can also be used for splitting files into multiple parts. For large files, splitting can be desirable because of storage and bandwidth limitations. Splitting even medium-sized files into standard-sized parts (e.g. 2^n kilobytes) also has advantages in combating traffic analysis. This is easily accomplished by inserting each part separately under a content-hash key, and creating an indirect file (or multiple levels of indirect files) to point to the individual parts.

All of this still leaves the problem of finding keys in the first place. The most straightforward way to add a search capability to Freenet 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 each containing a pointer to the real file, named according to search keywords chosen by her. These indirect files would differ from normal files in that multiple files with the same key (i.e. search keyword) would be permitted to exist, and requests for such keys would keep going until a specified number of results were accumulated instead of stopping at the first file found. Managing the likely large volume of such indirect files is an open problem.

An alternative mechanism is to encourage individuals to create their own compilations of favorite keys and publicize the keys of these compilations. This is an approach also in common use on the world-wide web.

3.2 Retrieving data

To retrieve a file, a user must first obtain or calculate its binary file key. 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 limit is exceeded, a

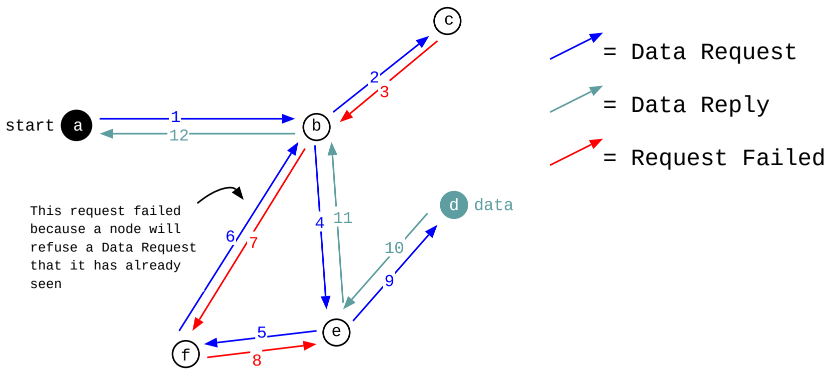

Fig. 1. A typical request sequence.

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*, 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 geographic 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, the announcement mechanism described in section 3.5 is necessary). Note that direct links to data sources 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.

Since keys are derived from hashes, lexicographic closeness of keys does not imply any closeness of the original descriptive strings and presumably, no closeness of subject matter of the corresponding files. This lack of semantic closeness is not important, however, as the routing algorithm is based on knowing here keys are located, not where subjects are located. That is, supposing a string such as `text/philosophy/sun-tzu/art-of-war` yields a file key `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 hashes 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. The same is true for personal subspaces—files belonging to the same subspace will be scattered across different nodes.

3.3 Storing data

Inserts follow a parallel strategy to requests. To insert a file, a user first calculates a binary file key for it, using one of the procedures described in section 3.1. 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. 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 situation for requests. 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 use inserts as a supplementary means of announcing their existence to the rest of the network. Third, attempts by attackers to supplant existing files by inserting junk files under existing keys are likely to simply spread the real files further, since the originals are propagated on collision. (Note, however, that this is mostly only relevant to keyword-signed keys, as the other types of keys are more strongly verifiable.)

3.4 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[29] in which data items are 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 resulting impact on availability is mitigated 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. That is, there is no "permanent" copy which is being replicated in a cache. 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 systems such as Eternity and Free Haven which seek to provide guarantees of file lifetimes.

The expiration mechanism has an advantageous aspect, however, in that it allows outdated documents to fade away naturally after being superseded by newer documents. 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. This is why all stored files are encrypted. The encryption procedures used are not intended to 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 file key, not the encryption key. The encryption keys for keyword-signed and signed-subspace data can only be obtained by reversing a hash, and the encryption keys for content-hash data are completely unrelated. With effort, of course, a dictionary attack will reveal which keys are present—as it must in order for requests to work at all—but the burden such an effort would require is intended to provide a measure of cover for node operators.

3.5 Adding nodes

A new node can join the network by discovering the address of one or more existing nodes through out-of-band means, then starting to send messages. As mentioned previously, the request mechanism naturally enables new nodes to learn about more of the network over time. However, in order for existing nodes to discover *them*, new nodes must somehow announce their presence. This process is complicated by two somewhat conflicting requirements. On one hand, to promote efficient routing, we would like all the existing nodes to be consistent in deciding which keys to send a new node (i.e. what key to assign it in their routing tables). On the other hand, it would cause a security problem if any one node could choose the routing key, which rules out the most straightforward way of achieving consistency.

We use a cryptographic protocol to satisfy both of these requirements. A new node joining the network chooses a random seed and sends an announcement message containing its address and the hash of that seed to some existing node. When a node receives a new-node announcement, it generates a random seed, XOR's that with the hash it received and hashes the result again to create a commitment. It then forwards the new hash to some node chosen randomly from its routing table. This process continues until the hops-to-live of the announcement runs out. The last node to receive the announcement just generates a seed. Now all nodes in the chain reveal their seeds and the key for the new node is assigned as the XOR of all the seeds. Checking the commitments enables each node to confirm that everyone revealed their seeds truthfully. This yields a consistent random key which cannot be influenced by a malicious participant. Each node then adds an entry for the new node in its routing table under that key.

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 using a persistent channel such as a TCP connection may also send multiple messages over the same connection. 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`. Nodes which change addresses frequently may also use virtual addresses stored under *address-resolution keys* (ARK's), which are signed-subspace keys updated to contain the current real address.

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 sender's return address may be impossible to determine automatically from the transport layer, or the sender may wish to receive replies at a different address 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 limit, 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 3.2. Using the chosen hops-to-live limit, 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.

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

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 3.3. 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 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 Performance analysis

We performed simulations on a model of this system to give some indications about its performance. Here we summarize the most important results; for full details, see [21].

5.1 Network convergence

To test the adaptivity of the network routing, we created a test network of 1000 nodes. Each node had a datastore size of 50 items and a routing table size of 250 addresses. The datastores were initialized to be empty, and the routing tables were initialized to connect the network in a regular ring-lattice topology in which

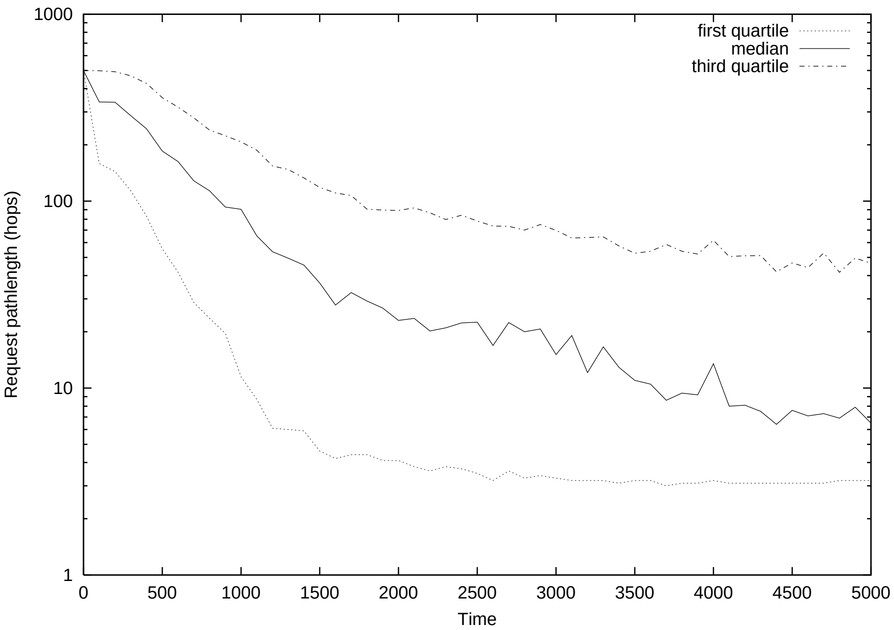

Fig. 2. Time evolution of the request pathlength.

each node had routing entries for its two nearest neighbors on either side. The keys associated with these routing entries were set to be hashes of the destination nodes' addresses. Using hashes has the useful property that the resulting keys are both random and consistent (that is, all references to a given node will use the same key).

Inserts of random keys were sent to random nodes in the network, interspersed randomly with requests for randomly-chosen keys known to have been previously inserted, using a hops-to-live of 20 for both. Every 100 timesteps, a snapshot of the network was taken and its performance measured using a set of probe requests. Each probe consisted of 300 random requests for previously-inserted keys, using a hops-to-live of 500. We recorded the resulting distribution of *request pathlengths*, the number of hops actually taken before finding the data. If the request did not find the data, the pathlength was taken to be 500.

Figure 2 shows the evolution of the first, second, and third quartiles of the request pathlength over time, averaged over ten trials. We can see that the initially high pathlengths decrease rapidly over time. In the beginning, few requests succeed at all, but as the network converges, the median request pathlength drops to just six.

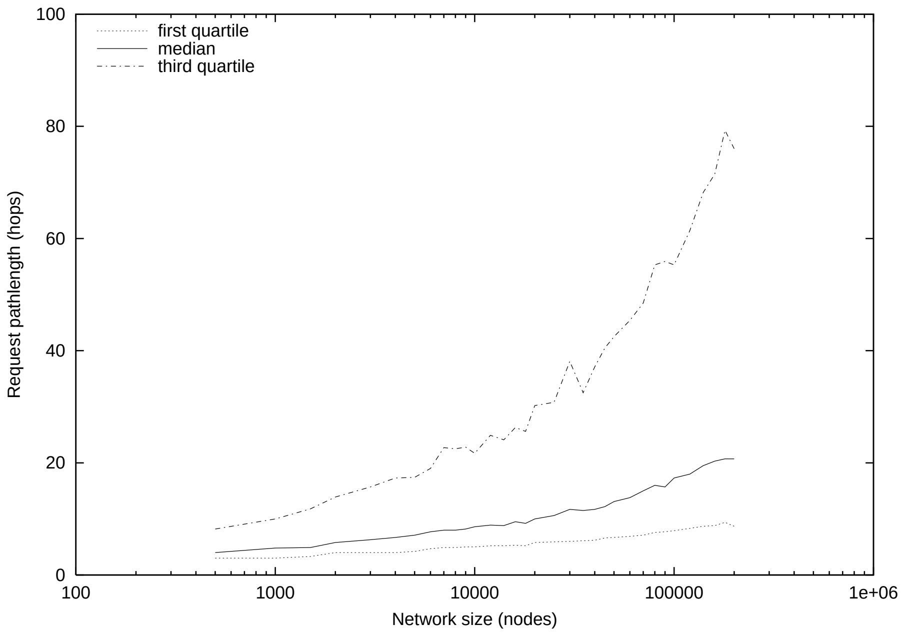

Fig. 3. Request pathlength versus network size.

5.2 Scalability

Next, we examined the scalability of a growing network. Starting from a small network of 20 nodes initialized in the same manner as the previous section, we added new nodes over time and measured the change in the request pathlength.

Inserts and requests were simulated randomly as before. Every five timesteps, a new node was created and added to the network by simulating a node announcement message with hops-to-live of 10 sent from it to a randomly-chosen existing node. The key assigned by this announcement was taken to be the hash of the new node's address. Note that this procedure does not necessarily imply a linear rate of network growth, but rather a linear relationship between the request rate and the growth rate. Since it seems likely that both rates will be proportional to network size (yielding an exponential growth rate in real, as opposed to simulated, time), we believe that this model is justifiable.

Figure 3 shows the evolution of the first, second, and third quartiles of the request pathlength versus network size, averaged over ten trials. We can see that the pathlength scales approximately logarithmically, with a change of slope near 40,000 nodes. We posit that the slope change is a result of routing tables becoming filled and could be improved by adding a small number of nodes with larger routing tables. Section 5.4 discusses this issue in more depth. Where our routing tables were limited to 250 entries by the memory requirements of the simulation, real Freenet nodes should easily be able to hold thousands of entries.

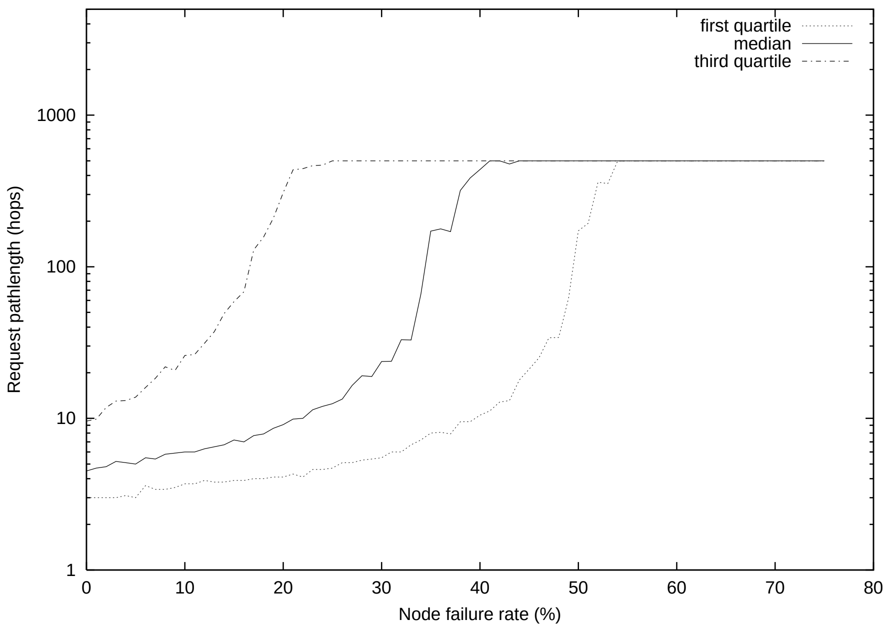

Fig. 4. Change in request pathlength under network failure.

Nonetheless, even this limited network appears capable of scaling to one million nodes with a median pathlength of just 30. Note also that the network was grown continuously, without any steady-state convergence period.

5.3 Fault-tolerance

Finally, we considered the fault-tolerance of the network. Starting with a network grown to 1000 nodes by the previous method, we progressively removed randomly-chosen nodes from the network to simulate node failures. Figure 4 shows the resulting evolution of the request pathlength, averaged over ten trials. The network is surprisingly robust against quite large failures. The median pathlength remains below 20 even when up to 30% of nodes fail.

5.4 Small-world model

The scalability and fault-tolerance characteristics of Freenet can be explained in terms of a *small-world* network model[23, 31, 22, 3]. In a small-world network, the majority of nodes have only relatively few, local, connections to other nodes, while a small number of nodes have large, wide-ranging sets of connections. Small-world networks permit efficient short paths between arbitrary points because of the shortcuts provided by the well-connected nodes, as evidenced by

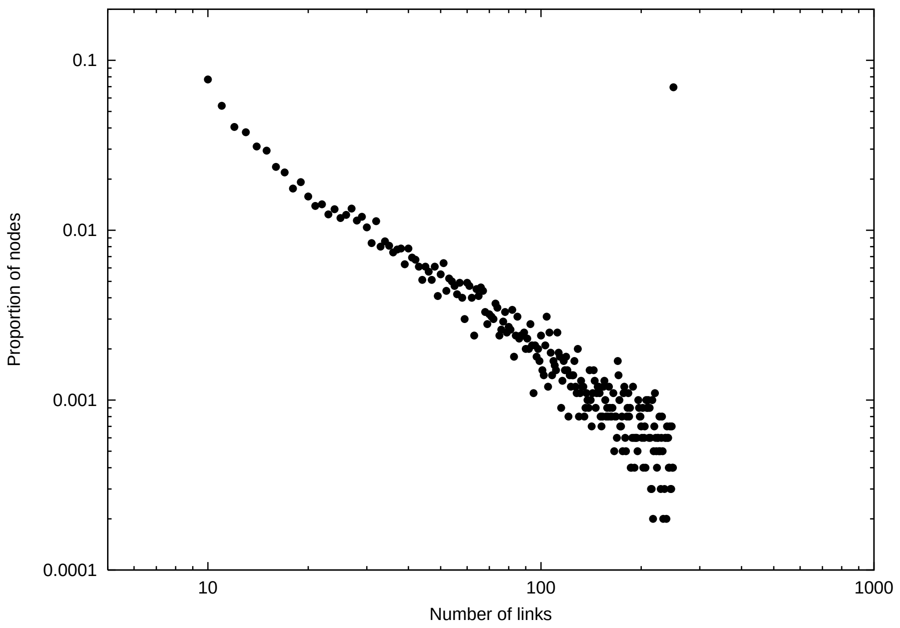

Fig. 5. Distribution of link number among Freenet nodes.

examination of Milgram's letter-passing experiment[23] and the Erdös number game cited by Watts and Strogatz[31].

Is Freenet a small world? A key factor in the identification of a small-world network is the existence of a scale-free power-law distribution of links within the network, as the tail of such distributions provides the highly-connected nodes needed to create short paths. Figure 5 shows the average distribution of links (i.e. routing table entries) in the 1000-node Freenet networks used in the previous section. We see that the distribution closely approximates a power law, except for the anomalous point representing nodes with filled 250-entry routing tables. When we used differently-sized routing tables, this cutoff point moved but the power-law character of the distribution remained the same.

In addition to providing short paths, the power-law distribution also gives small-world networks a high degree of fault-tolerance. Random failures are most likely to knock out nodes from the majority that possess only a small number of connections. The loss of poorly-connected nodes will not greatly affect routing in the network. It is only when the number of random failures becomes high enough to knock out a significant number of well-connected nodes that routing performance will be noticeably affected.

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.

6 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[25] 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 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 their unhashed versions 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. [25] 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 of requested files by a malicious node in a request chain is an important threat, and not only because of the corruption of the files themselves. 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 signed-subspace 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 keyword-signed keys, however, is vulnerable to dictionary attack since signatures can be made by anyone knowing the original descriptive string.

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 junk files. An interesting possibility for countering this attack is a scheme such as Hash Cash[20]. Essentially, this 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 junk inserts might temporarily paralyze insert operations but would not displace existing files. It is difficult for an attacker to artificially legitimize her own junk 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 displace existing files by inserting alternate versions under the same keys. Such an attack is not possible against a content-hash key or signed-subspace key, since it requires finding a hash collision or successfully forging a cryptographic signature. An attack against a keyword-signed 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 3.3) 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.

7 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 adaptive 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 tens of thousands of copies downloaded and many interesting files in circulation. Because of the anonymous 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 nodes joining and leaving simultaneously, variation in node capacity and bandwidth, and larger network sizes. We would also like to implement a public-key infrastructure to authenticate nodes and create a searching mechanism.

8 Acknowledgements

This material is partly 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," *Linux Gazette* issue 38, March 1999.
2. Akamai, `http://www.akamai.com/` (2000).
3. R. Albert, H. Jeong, and A. Barabási, "Error and attack tolerance of complex networks," *Nature* **406**, 378-382 (2000).
4. American National Standards Institute, American National Standard X9.30.2-1997: *Public Key Cryptography for the Financial Services Industry - Part 2: The Secure Hash Algorithm (SHA-1)* (1997).
5. 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).
6. Anonymizer, `http://www.anonymizer.com/` (2000).
7. O. Berthold, H. Federrath, and S. Köpsell, "Web MIXes: a system for anonymous and unobservable Internet access," in *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, USA. Springer: New York (2001).
8. D.L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM* **24**(2), 84-88 (1981).
9. 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).
10. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," *Journal of the ACM* **45**(6), 965-982 (1998).
11. Church of Spiritual Technology (Scientology) v. Dataweb *et al.*, Cause No. 96/1048, District Court of the Hague, The Netherlands (1999).
12. I. Clarke, "A distributed decentralised information storage and retrieval system," unpublished report, Division of Informatics, University of Edinburgh (1999). Available at `http://www.freenetproject.org/` (2000).
13. L. Cottrell, "Frequently asked questions about Mixmaster remailers," `http://www.obscura.com/~loki/remailer/mixmaster-faq.html` (2000).

14. R. Dingledine, M.J. Freedman, and D. Molnar, "The Free Haven project: distributed anonymous storage service," in *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, USA. Springer: New York (2001).
15. Distributed.net, `http://www.distributed.net/` (2000).
16. D.J. Ellard, J.M. Megquier, and L. Park, "The INDIA protocol," `http://www.eecs.harvard.edu/~ellard/India-WWW/` (2000).
17. Gnutella, `http://gnutella.wego.com/` (2000).
18. I. Goldberg and D. Wagner, "TAZ servers and the rewebber network: enabling anonymous publishing on the world wide web," *First Monday* **3**(4) (1998).
19. D. Goldschlag, M. Reed, and P. Syverson, "Onion routing for anonymous and private Internet connections," *Communications of the ACM* **42**(2), 39-41 (1999).
20. Hash Cash, `http://www.cypherspace.org/~adam/hashcash/` (2000).
21. T. Hong, "Performance," in *Peer-to-Peer*, ed. by A. Oram. O'Reilly: Sebastopol, CA, USA (2001).
22. B.A. Huberman and L.A. Adamic, "Internet: growth dynamics of the world-wide web," *Nature* **401**, 131 (1999).
23. S. Milgram, "The small world problem," *Psychology Today* **1**(1), 60-67 (1967).
24. Napster, `http://www.napster.com/` (2000).
25. M.K. Reiter and A.D. Rubin, "Anonymous web transactions with Crowds," *Communications of the ACM* **42**(2), 32-38 (1999).
26. The Rewebber, `http://www.rewebber.de/` (2000).
27. M. Richtel and S. Robinson, "Several web sites are attacked on day after assault shut Yahoo," *The New York Times*, February 9, 2000.
28. J. Rosen, "The eroded self," *The New York Times*, April 30, 2000.
29. A.S. Tanenbaum, *Modern Operating Systems*. Prentice-Hall: Upper Saddle River, NJ, USA (1992).
30. M. Waldman, A.D. Rubin, and L.F. Cranor, "Publius: a robust, tamper-evident, censorship-resistant, web publishing system," in *Proceedings of the Ninth USENIX Security Symposium*, Denver, CO, USA (2000).
31. D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature* **393**, 440-442 (1998).
32. Zero-Knowledge Systems, `http://www.zks.net/` (2000).