GNUNET – A truly anonymous networking infrastructure

Krista Bennett, Christian Grothoff, Tzvetan Horozov, Ioana Patrascu, and Tiberiu Stef

Department of Computer Sciences, Purdue University
{klb,grothoff,horozov,patrascu,tstef}@cs.purdue.edu
http://gecko.cs.purdue.edu/GNUnet/

Abstract. This paper describes aspects of *GNUnet*, a framework for reliable anonymous distributed file-sharing with low traffic and CPU overhead and defenses against malicious hosts.
We describe a new technique to encode content such that it can be easily distributed, searched for and retrieved. The encryption scheme allows users to insert the same content under multiple keys; yet, multiple keys lead to identical copies in the system, reducing storage requirements. Keys can be chosen from natural language and can be combined to boolean queries. Queries and content cannot be decrypted by intermediaries without guessing the key. The encoding of the content produces many small *GBlocks*, which can be easily distributed over several hosts. This allows the network to balance load. Single hosts are never hit with requests that take a long time to process.
A new scheme that allows for anonymous transfer is described. This scheme achieves the same anonymity guarantees as traditional indirection schemes but is more efficient. While *GNUnet* aims to provide anonymity for its users, it requires authentication for the servers to guard against DoS attacks. A new model for micropayments was designed to make *GNUnet* resistant to attacks.

1 Introduction

Dynamic, wide area networks (WANs) present a challenge to any system that needs to provide security guarantees to its users. This is especially true because the widely used TCP/IP protocols [8] lack mechanisms for authentication or confidentiality [4]. The primary applications for distributed applications are computation and data transfer. In this paper, we focus on data transfer. The traditional methods of transferring files over networks have serious drawbacks which are partially caused by the underlying network technology. Some of these drawbacks include:

– Communications are not anonymous. In general, the sender, the receiver, intermediaries (e.g. routers), queries and replies are not protected. *GNUnet*

achieves full anonymity by protecting all of these entities. While participation, in general, can still be observed, all hosts can deny that they knew more than that a query took place and a reply was sent;
- Data transfers can be intercepted [1]. At present, the Internet provides neither good authentication nor confidentiality. Application-level protocols such as ssh [21] and ssl [13] try to address this issue. The *network* layer of the *GNUnet* framework provides these services for *GNUnet*'s applications;
- On the Internet, malicious hosts cannot be prevented from sending data to any other host. This makes it possible for adversaries to launch Denial-of-Service (DoS) attacks against networks [9]. Protecting a distributed network against abuse while keeping the network dynamic and open to new users is a problem that will be addressed in this paper. Using accounting mechanisms, *GNUnet* forces a host to contribute more to the network than the host can demand from the network, thus preventing certain types of DoS attacks and abuse, which might endanger the network [2];
- Many services, including certain distributed systems, rely on central servers that must be available and carry significant amounts of the load. The performance of *GNUnet* is worse than that of the most efficient data transfer mechanisms. Yet, *GNUnet* can decentralize the load better than other systems [18, 6]. Reducing the load of servers by distributing each file on the network is one of the prime goals of *GNUnet*.

The remainder of the paper is organized as follows. Section 2 presents a set of criteria that we use to evaluate *GNUnet* and describes common solutions. In section 3, we describe how files are distributed over the network. This section also describes the way *GNUnet* provides anonymity for users and guarantees that servers cannot exercise editorial control, thus allowing participants to avoid liability. Related work is discussed in section 5. Future work is described in section 6. Finally, section 7 concludes the paper.

2 Evaluation Criteria

We attempt to achieve several goals within the *GNUnet* framework. The *network* layer provides authentication and protects against particular attacks. In the *application* layer, we have implemented a file-sharing service called *GProxy* that provides full anonymity.

2.1 Privacy Guarantees

Distributed systems that require the user to expose his actions to other hosts of the system are very common. In most existing systems, the user must specify (to the network) the details of his query in plain-text [5, 14, 11]. Worse yet, the identity of the host sending the query is usually exposed as well.

At first sight, this seems inevitable: the query must be exposed in order for the network to find what the user is looking for. The host must be exposed in order to send the results back. Obvious solutions, such as broadcasting the result to every host, are extremely inefficient. All other solutions are, to some degree, susceptible to traffic analysis.

2.2 Lookup Mechanism

In existing systems, four major lookup mechanisms are deployed. The most widely used is a central server that is aware of the location of any file. While a hierarchical model can address scalability issues, a central authority must still make the ultimate decisions [23]. This has the drawback that a central point-of-failure cannot be avoided. Another extreme is one in which every host on the network is aware of all locations of all files [15]. This approach obviously does not scale well, though on small systems it may actually be worthwhile. The third approach performs in parallel a breadth-first search of the entire network [11]. This clearly has the potential to cause large amounts of network traffic. The last approach uses a depth-first single-threaded search of the network [7] which usually results in slow response times.

2.3 Protection Against Abuse

All distributed systems are exposed to the Internet and thus subject to attacks and abuse [2]. Most currently used systems do not monitor the behavior of hosts. This allows hosts to use the distributed network without contributing back to it. Thus, these networks become susceptible to DoS attacks. While this does not stop these networks from attracting millions of users, it does degrade performance. Worse, it discourages commercial use of these systems because they are not reliable.

2.4 Distribution Mechanism for Files

Distributed file-sharing networks fall into two categories. In the first category, hosts join the network and provide files to other participants. After a host leaves the network, its content disappears (except when content is explicitly copied to another host by a user[11]). This allows users to control which content they offer.

While some consider it an advantage to be able to control the content that one hosts, this also carries with it great disadvantages. Exercising editorial control implies liability for the content, whereas a lack of editorial control usually relieves the provider from responsibility for the nature of the data hosted by its service.[1]

In the second category of file-sharing networks, content migrates from host to host. Individual hosts can not see which content they are hosting, no one is in control. The network itself takes care of placing or discarding content.

[1] In a court trial, a provider was found liable for hosting a bulletin-board where offensive messages had been posted. This is because the provider had hired people to exercise editorial control. If the provider had merely published messages from users without interference, the provider would not have been held accountable[24].

2.5 Static or Dynamic Network

Most systems for distributed computing form a static network in which hosts cannot join or leave at any time [22]. Globally distributed file-sharing systems, on the other hand, are usually dynamic. However, these systems vary in how expensive joining and leaving the network is. Centralized systems often require some form prior authentication of the clients with the server before they are allowed to communicate with other clients ([16, 17, 10].

More distributed systems usually require only joining hosts to communicate only with an arbitrary other host on the network [12]. These networks without a central authority are especially prone to problems with respect to guarding the network against abuse. As nodes join and leave, their identity (IP) is often lost. New nodes cannot be trusted; still, it would be beneficial if they were able to use the network to some extent.

2.6 Efficiency

The efficiency of a distributed network can be judged by a few obvious criteria: the total amount of bandwidth that is required to transfer a file, the amount of storage space that is required, and the time it takes to obtain the file. Less obvious, but nevertheless important, is the maximum load that a request can cause for a single server. Clearly, the recipient will have to receive data of at least the size of the file. However, the maximum individual load of the other participants may be less than the highest load for hosts participating in a direct transfer of the entire file.

3 Content representation

The design of *GNUnet* storage and distribution of content attempts to achieve several goals:

1. Deniability for all participants;
2. Distribution of content and load;
3. Efficiency in terms of space and bandwidth.

In order to protect intermediaries, it is desirable that these hosts have no way to find out what they are serving. Content migration is used to ensure that the original sender cannot be located as well as for distribution of the load. The host that initially inserted the content can then forget the "key" and claim ignorance, even if the content is still stored locally: as long as the adversary has not performed full traffic analysis, the content could plausibly have come from another node.

Furthermore, the host sending the query must be protected. The other hosts should neither be able to find out what the query is about, nor what the content they send back contains.

In this section, we describe an encoding for content that achieves all three goals. While the receiver can decrypt the file, none of the intermediaries has a chance to look at the file. Also, the query is sent encrypted such that the other hosts on the network cannot compute what the user is looking for. Some minor goals have also influenced our design. First, we wanted to be able to perform boolean queries of the form $a\, AND\, b$, without revealing a or b. Second, content migration must be simple and fast, making it easier for *GNUnet* to migrate (or copy) files, achieving distribution and redundancy. Finally, users should be able to store a file in plaintext on their local drive (perhaps because they are using it), without doubling space requirements:

4. The encoding should allow files to be served that are stored in plaintext locally without requiring the storage of a second copy in encrypted form.[2]

To the best of our knowledge, no system exists that allows on-demand encryption without encrypting the whole file at the time of the request (or without sending the file in encrypted form), which is not practical because the encryption is too costly for the server.

The remainder of this section describes how these goals are achieved by our approach. In part 3.1 we describe how files are split into *GBlocks* which are easier to migrate. Part 3.2 describes how the *GBlocks* are encrypted in order to make it impossible for the intermediaries to determine what they are storing or serving. Part 3.3 describes how the user can query for content without exposing the content or the (plaintext) query. Finally, we describe how indirections are used in *GNUnet* to achieve full deniability. The following description details the full scheme that we develop in this section:

1. The user gives the local proxy the content C, a list of keywords K and a description D (and optionally the pseudonym P) to use.
2. The proxy splits C into blocks B_i, each of size 1k, and computes the hash values $H_i = H(B_i)$ and $H_i^2 = H(H(B_i))$. Random padding is added if needed.
3. Each block is then encrypted, yielding $E_i = E_{H_i}(B_i)$.
4. The proxy stores E_i under the name H_i^2.
5. If there is more than one H_i, the proxy groups 51 H_i values together with a CRC32 of the original data to a new block of 1k. Random padding is added if needed. All the 1k blocks obtained this way are then treated like in step 2.
6. If there is only one hashcode H_1, the proxy builds a root-node, containing H_1, the description D, the original length of C, a CRC checksum and optionally P and a signature. All this must be less than 1k in size (the length of the description may be shortened as needed). The resulting root-node R is padded and encrypted for each keyword K yielding $R_K = E_{H(K)}(R)$.
7. Finally, for each K, the result R_K is stored under $H(H(K))$.

[2] This would of course reduce deniability, yet we value giving users the choice between efficiency and security.

The space m required for a file of size n is

$$\begin{aligned} m &\leq n + 1k \cdot \sum_{i=0}^{\lfloor log_{51} \lceil \frac{n}{1k} \rceil \rfloor} 51^i \\ &\approx 1.02 \cdot n. \end{aligned}$$

3.1 *GBlocks*

In order to be able to migrate large files and to distribute load on the network, *GNUnet* splits large files into many small files, called *GBlocks*. Similar to UNIX INodes, special indirection-blocks called *IBlocks* encode how the leaf-nodes of the file tree, called *DBlocks*, can be reassembled into the original file (see figure 1).

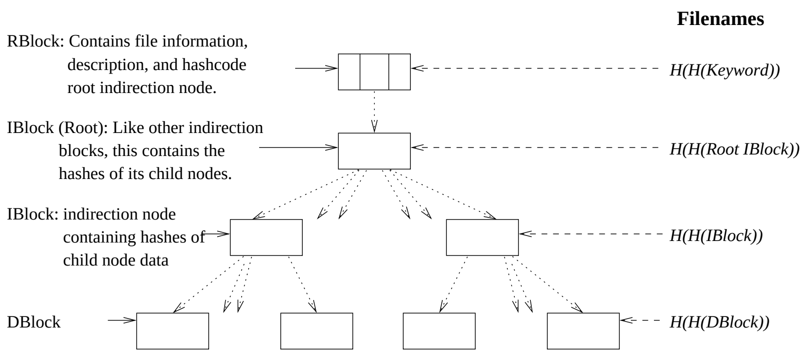
Filenames
RBlock: Contains file information, description, and hashcode root indirection node.
H(H(Keyword))
IBlock (Root): Like other indirection blocks, this contains the hashes of its child nodes.
H(H(Root IBlock))
IBlock: indirection node containing hashes of child node data
H(H(IBlock))
DBlock
H(H(DBlock))

Fig. 1. Encoding of the entire file

Splitting large files makes content migration cheap. For large files, it is unlikely that a node will be able to find another host that is willing (or able) to provide enough space. The traffic-burst that this transfer would impose on the network is impractical because both nodes would become very busy for a long time. Storing a large file on a single host also fails to facilitate distribution of load when a user requests this large file.

The size of all types of *GBlocks* in *GNUnet* is normalized to 1k. Thus, a 50k file will be split into 50 *DBlocks* f_i of length 1k each. Then, 50 RIPE-MD-160 hashcodes of the small files are composed to form an *IBlock* $H(f_1), \ldots, H(f_{50})$, which is stored with all the other *GBlocks* (see figure 2). *GBlocks* that are less than 1k in length are padded with random information. *IBlocks* can store up to 51 hashcodes ($51 * 20\, bytes = 1020\, bytes$) and one CRC32 checksum.

Larger files require more levels of indirection. A special *RBlock* on top of the file-tree contains the length of the file, a description and a hashcode. In this way, the user can download a single 1k *GBlock* and obtain information about the contents of the full-length corresponding file before choosing to download the rest.

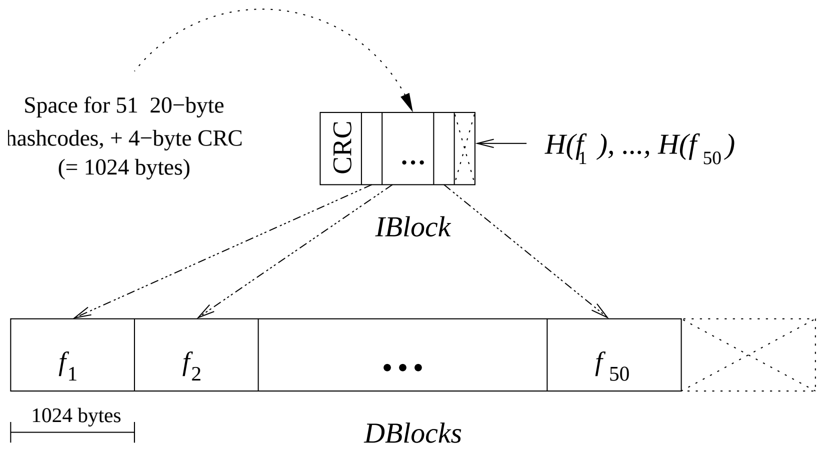

Fig. 2. Encoding of the 1k blocks

Obviously, this fine-grained splitting of the files increases the number of queries that must be made in order to obtain a file. Because the file is distributed in parts, it is entirely possible that a file cannot be fully reconstructed, even if some parts of the file are still available on the network. While this problem cannot be solved entirely, duplication of files in the network will make this problem less likely.

One could argue that a size of 1k is far too small. The rationale behind this filesize is that UDP, the transport mechanism used by *GNUnet* has (on Ethernet) an optimal packet size that is slightly above 1k. As *GNUnet* must add headers to the packets, the size approximates this number. Furthermore, many filesystems can be formatted with a block-size of 1k.

3.2 Encryption of the Content

While the previous section described the splitting of files into *GBlocks*, this technique is not sufficient to ensure that intermediaries are unable to guess which content they are transmitting. However, hiding content from intermediaries can be achieved by encrypting the *GBlocks* in a way such that their hosts cannot decrypt them. This will require some kind of secret key that the recipient must know. How the receiver obtains the secret key is discussed in the next section.

Encryption of the content with a secret key gives rise to another challenge. Suppose two parties insert the same file into the network, but under different keys. If one part of each version of the file is lost, neither of the files can be recovered with just one of the keys. This is true even if the parts lost in both files do not overlap and the file could otherwise have been reconstructed using both keys.

GNUnet encodes *GBlocks* in a special way that allows it to reassemble parts of two identical files that were inserted under different secret keys without anyone but the recipient even knowing that the *GBlocks* were related.

The key idea here is to use the hash of the *GBlock* as the key. More precisely, *GNUnet* encodes a 1k block B with the hash of B, $H(B)$. A RIPE-MD-160 hash provides 128 bits for a symmetric cipher, and another 32 bits for the initialization vector. Because the hashcode cannot be retrieved from the encrypted file $E_{H(B)}(B)$, the intermediaries cannot decrypt the content. Yet, even with this scheme in place, two identical plaintexts that were inserted independently will have the same keys and thus yield the same encrypted ciphertexts.

For the *RBlock*, a slightly different scheme is used to allow queries. Here, the user specifies the secret keys. The two different secret keys are used only to encrypt the *RBlock*s. Either of the keys can then be used to decrypt the entire file. A small problem arises from *GBlocks* that need to be padded to achieve the size of 1k. In these cases, only the actual data is hashed, not the random padding.

The encryption scheme for the *RBlock* is similar to the scheme used in FREENET as both use the hash of the keyword. Yet, our keywords are free-form and we only encrypt the *RBlock*; the rest of the file is encrypted differently.

3.3 Queries

The scheme described so far leaves one question open; that is, how to query for data. If the files were stored under the hash-codes of the original data, intermediaries could decrypt them. However, storing more than a single value per *DBlock* in the *IBlocks* would be inefficient.

In *GNUnet*, a block B is stored as $E_{H(B)}(B)$ under the name $H(H(B))$. Because H is a one-way function, the intermediaries cannot obtain the original hashcode from the filename. However, other nodes can compute $H(H(B))$ given $H(B)$.

The *RBlock* R of a file stores the hashcode $H(B)$ of the root *IBlock* B which is then encrypted with $H(K)$ where K is a user-supplied keyword. $E_{H(K)}(R)$ is then stored under $H(H(K))$. When a user searches for K, his client will send a request for $H(H(K))$ and decrypt the result with $H(K)$. No other node in the network can obtain any information about the data transferred except by guessing the keyword K (or by knowing the file that is returned, which usually implies that this node originally inserted the file). Although a dictionary attack is thus possible, such an attack can be avoided by carefully choosing K.

This scheme has another advantage. It allows users to specify boolean queries of the form $a\,AND\,b$. *GProxy* will then search for a and for b and return only those files that match both queries. Content providers can insert content under many keywords without using significantly more storage space, because only one extra *RBlock* per keyword is required, thus allowing users to search for content efficiently.

3.4 Avoiding Content from Malicious Hosts

As mentioned before, the actual query for a datum matching Q is hidden by hashing Q first. As $H(Q)$ is used as the key for the decryption, $H(H(Q))$ is the obvious choice for the published query. However, this approach has a small problem.

If N matches the query Q, the encoded node $E_{H(Q)}(N)$ no longer has hash Q. Thus, intermediaries (that do not know $H(Q)$) cannot verify that this node matches the query at all. A malicious node could return garbage to any query $H(H(Q))$, claiming that the garbage matches the query. The receiver would then have to propagate a message back through the chain that the original sender was malicious. As intermediaries cannot keep track of earlier connections for a long time, this feedback may not reach the malicious node. Thus, the malicious node could actually earn credit (see section 4) by sending out garbage to the network.

Such a problem can be prevented by using $H(H(H(Q)))$ for the query. The sender must then provide $H(H(Q))$ to demonstrate that the sender actually has matching data. Because the sender cannot guess $H(H(Q))$, it can be assumed that the sender had matching content at some point. Of course, this cannot prevent malicious hosts from creating garbage in general. Malicious nodes could *guess* that the keyword K is frequently used and just compute $H(H(K))$ and $H(H(H(K)))$ and return their garbage once a matching query comes by. However, this is similar to inserting garbage under that keyword into the network. Because no software can distinguish valuable content from garbage in general, this is not a design flaw. Still, it demonstrates that content moderation (feedback, ranking) is required. The triple-hash scheme simply makes it harder to reply to arbitrary queries with garbage; it cannot prevent users from inserting garbage under often-used keys.

3.5 Deniability

Traditionally, indirection has been frequently used in order to achieve anonymity. Forwarding queries from other hosts allows users to deny that a query originated from a particular host (assuming the adversary was not able to perform a full-traffic analysis).

The same applies for replies. If replies are not sent directly to the node that issued the request, the sender and receiver are unable to know each other. This is true even if all of the intermediaries are malicious and colaborating. The sender or recipient can still claim that it just indirected the reply. Again, this assumes that no complete traffic analysis is used by the attacker.

Systems deployed in practice previously took a binary approach; either they indirect all queries or none. Indirecting all queries is very costly, while using no indirection voids anonymity. *GNUnet* uses a different approach: intermediaries indirecting queries are *free to decide* whether or not they want to indirect the reply. If nodes want to indirect, they substitute the sender by their own identificaton. Otherwise, they simply keep the original query intact. *GNUnet* nodes base their choice on the current load of the local node and economic factors (see the next section). In order to avoid predictability, some randomness is added. Unlike Crowds [20], our system can thus adapt to the circumstances.

This adaptive scheme gives the same guarantees as the "always indirect" systems, while being nearly as efficient as the "never indirect" systems. If challenged, nodes can still claim that they indirected a query. Nodes that are very busy can still refuse to indirect, yielding the same load as for the "never indirect" system. For nodes that are not busy, the extra load imposed by the indirection is generally not a problem.

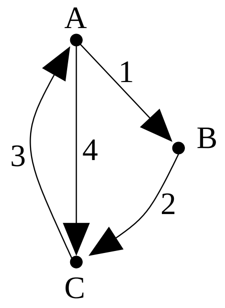

Fig. 3. Indirecting Replies

This insight has other serious implications. Suppose a node B receives a query from A and forwards it to C but preserves A as the receiver for the reply (see figure 3). In this case, C will send a reply directly to A. Thus, A gets to know C. In addition, A and C will have already performed a sessionkey exchange, making it likely that A will send another query directly to C. With *GNUnet*'s fine grained content it is likely that A usually performs many related queries. For the same document, it is reasonable to assume that C will often be closer to the location of the document than B is. This way, the number of hops between A and the content is decreased, speeding up the download process.

This raises the question, who profits from indirecting a query. The surprising answer is that if B indirects a query from A, the anonymity of A is not increased, but the anonymity of B. After all, if B is indirecting a number of queries, B can claim for a particular query that it was indirected. A and C do not profit from this; they are fully exposed to B. Of course, A and C can protect themselves by indirecting other queries, making it impossible for B to ensure that it is actually A who initiated the query.

Thus, it is impossible for malicious hosts to do harm by not indirecting queries. A host can only harm its own anonymity by not indirecting queries. In order to achieve anonymity, a host must indirect a *sufficient number* of queries from other hosts. If a host M never indirects, queries originating from that host with M as the receiver are certain to be initiated by M. Hosts that sometimes indirect, as well as hosts that always indirect, can claim the query was initiated elsewhere. Thus, indirecting *sometimes* gives us both anonymity and performance.

4 Trust Economy

Any distributed network is potentially vulnerable to attacks by malicious hosts that violate the protocols and rules set for the network. Malicious behavior includes attacks against the content in the network as well as against network resources, such as bandwidth.

In order to protect the network, malicious hosts must be detected and their impact limited. As the network is distributed and hosts are able to join the network at any time without signing up with a central authority, the detection of malicious hosts must also be decentralized.

In *GNUnet*, every node evaluates the behavior of the other nodes that it communicates with. New nodes that join the network start as *untrusted*. These nodes may send requests, but the established nodes will only reply if they have excess bandwidth. Even if the old hosts do not react to queries at all, a DoS attack by an overwhelming number of malicious hosts cannot be fought off if the malicious hosts have more bandwidth.[3] What can be done is to limit the ability of malicious hosts to consume network bandwidth; these hosts should not be able to produce *additional* traffic other than the traffic that originates from the malicious hosts themselves.

For example, in *Gnutella*, any host can start a search query. Each query multiplies in number as additional hosts are asked. In this way, a few malicious hosts may be able to bring down the entire network by making a large number of search queries. Thus, hosts that enter the network must be limited in their actions as follows:

– their requests should have a lower priority than the requests of established participating nodes;
– content brought into the network by these hosts should be discarded in favor of content from established hosts;
– they should be given content to store that is not important.

Once the new hosts have been in the network for some time, the other nodes should monitor their behavior:

– What is the availability and bandwidth provided?
– Do they obey the protocols?
– Is the content they provide valuable?

For example, the neighbors in the network may give the new node some files that they would otherwise discard. A few days later, a user may request the files from the new node. If the node still has the files, its rank is increased. Availability

[3] This is a general problem with the current Internet architecture.

and bandwidth of a node are kept as separate criteria for the node evaluation. In the future, *GNUnet* could use this information to decide where to store which content.

It is essential for *GNUnet* that it is possible to increase the ranking of well-behaved hosts without decreasing the ranking of other hosts to the same amount (i.e., no zero-sum in the rankings). Otherwise, all hosts would always have zero credit, because no one has credit to start with. Nevertheless, any activity that can be used for malicious behavior should decrease the ranking of the host. The scoring system must be designed to make sure that for malicious hosts the equation

$$contribution + \epsilon \geq damage - capacity \tag{1}$$

is satisfied where $\epsilon > 0$ should be small. Here, *capacity* is the bandwidth of the malicious host—even ignored search queries will do this much damage without contributing to the system. On the other hand, the ranking of well-behaved, participating hosts must increase over time.

If ϵ is sufficiently small, this system will ensure that, as long as a sufficient majority of the hosts are not malicious, the network "works". The accounting scheme in *GNUnet* is based on two principles. First, contributions must be rewarded and abuse be punished. Second, if hosts are idle, using their resources is not abuse.

Thus, when hosts perform queries, their ranking is decreased (they *pay* for the query); if they send (valid) replies, their ranking is increased. The amount of the increase/decrease depends on the priority of the query that was asked (or answered). This basic scheme is to be extended such that new nodes can earn credit and participate; if one node must always pay as much as another node receives, the system would be zero-sum and could not work. The source of new credit is *excess bandwidth.* The ϵ given in equation 1 is thus the *excess* capacity of the network.

If the node processing the query has excess bandwidth (and CPU time), it may decide not to charge the sender of the query. After all, the query did not cost it any performance. This is an important idea, because it allows nodes to *build up* credit. The system will produce credibility, and the nodes that provide more service than they use will rise in their ranking (the ranking is still increased even if excess bandwidth is used).

If the node processing the query is busy, it will discard queries with low priorities (and charge the nodes for asking questions). If a host sends queries with a priority higher than its own ranking, the policy decreases the ranking to the allowed ranking. Host rankings are kept for each pair of nodes that know each other.

Content Ranking

Content ranking is an important feature of *GNUnet.* Not only does content ranking help nodes to make better decisions about which content to discard and in what order, but it also helps to prevent malicious hosts from inserting garbage into the network (e.g. /dev/random).

Ranking of content occurs at two points. First, when the content is inserted, the user can specify how important it is. Other nodes may acknowledge that priority (based on whether or not they trust the sender of the data) or decrease it. Later, local nodes may decide to increase the ranking of content stored locally because it is requested. Increasing the ranking of the content by the priority of the request answered gives an acceptable heuristic.

The initial ranking of migrated content is limited by the trust level of the host inserting the content. However, one should bear in mind that any propagation of content from one node to another cannot be distinguished from insertion of fresh content. The user inserting the content into *GNUnet* should be able to tell the client how important the content is. The node can then ask other nodes to copy the content—and copying should lower the rank of a node.

Even with measures in place to keep highly-ranked content within the system, it is possible that over time this content may still disappear. Other content may achieve a higher ranking, either because it is frequently requested, or because hosts in the network insert content with an even higher ranking. In any case, this development occurs slowly. The general scheme that nodes are ranked will increase the interest of the node operators in keeping their nodes operational and the content stored on them intact.

5 Related Work

Currently, three major systems are used on the Internet with similar functionality to *GNUnet*. *Napster* [16] is a distributed file sharing system currently limited to `mp3` files where file distribution is coordinated through a central server. *Gnutella* [11] is a file sharing system based on the HTTP protocol without a centralized lookup mechanism or support for encryption. *Freenet* [6] is a distributed content sharing system that uses encryption on the hosts to protect servers from deciphering which content they serve. The remainder of this section discusses the technical issues of these implementations.

5.1 Napster

Napster's major drawback is that it is easy to discover everything about the communication by simply sniffing the traffic. Because *Napster* is centralized, it is particulary easy to shut down the system or to disclose user information[3]. The limitation to `mp3` files is just a corporate decision, not a technical issue. Content does not automatically migrate in Napster; the receiver has to manually add content to the export list.

The advantage of the *Napster* approach is the low overhead for the protocol and distribution and the fast and reliable lookup.

Napster has been used mostly to facilitate violations of contemporary copyright law by providing a service to users who want to share music.

5.2 Gnutella

Gnutella initially suffered from incompatible implementations and a small user community. The lack of a centralized lookup mechanism and the immense overhead to distribute queries is one of the major drawbacks of the system. Furthermore, the communication is not anonymous. The protocol leaks information on search queries and identifies the hosts providing the content [3].

On the other hand, the lack of centralization can be an advantage of *Gnutella*, which makes it harder to attack the network. As with *Napster*, content in *Gnutella* is explicitly provided by the participating hosts. As such, content does not magically disappear from the network. Like Napster, content does not automatically migrate in Gnutella.

Gnutella also has issues with "freeloading"; that is, users can download massive amounts of content without contributing any content of their own, effectively depleting bandwidth and storage resources without compensation. Such a situation could lead to nodes with more resources essentially becoming central servers, even though the network is "decentralized".

5.3 Freenet

Freenet does not suffer from the centralization issues associated with *Napster*. Additionally, the encryption scheme prevents individual servers from identifying the actual content of the data stored or transmitted using their resources. However, unlike *Napster* or *Gnutella*, it is possible for files stored in *Freenet* to disappear in favor of other files without user-intervention.

Freenet has the advantage that an individual server has no direct knowledge of the actual content of the data it is storing and distributing, which may shift some of the burden away from individual server owners in terms of liability. However, *GNUnet*'s solution, in which a single host generally has no means of reconstructing the whole file (even assuming the host can guess the key), increases the plausible deniability of the server owner. If the host would have to search the whole network to complete the file (in addition to guessing the key), it should be much harder to challenge the host operator for hosting small parts of the content.

Freenet encrypts the content using keys that identify the resource. If the key is known, then the associated file can be deciphered. *Freenet* has several different types of keys. The different key types are used to allow additional functionalities such as content signing, personal namespaces or splitting of content. As the key-structure is exposed directly to the user, use of the system requires a fair amount of knowledge. As far as we know, only the simplest key types (i.e. an unsigned, global namespace) are widely used. Keys that allow content updating have recently been introduced.

Queries in *Freenet* are serialized. Though this reduces the traffic overhead, it increases the time for a search to complete. On the other hand, content is propagated back on the search path. This increases the anonymity of the participants; a single communication might simply be a part of the search path in which neither of the participants as the ultimate sender or receiver. For long search paths, this content propagation may dramatically increase the overall traffic on the network. Unlike *GNUnet*, the content propagation path is fixed; individual nodes cannot decide if they should indirect the reply or short-cut the reply path. Thus, *Freenet* provides similar anonymity compared to *GNUnet*, but uses more bandwidth to achieve this goal.

A significant disadvantage of the current implementation of *Freenet* is that it does not allow direct sharing of files from the local drive without encrypting and inserting them first. Thus, to ensure content preservation, a node operator must keep a local copy of the unencrypted file in addition to the encrypted content on the *Freenet* server. Allowing the *Freenet* server to share local files directly may increase the stability and availablility of content in *Freenet* dramatically, especially for content where the node operator does not have to fear interference from outside authorities.[4]

Freenet is still under development. Recent versions had problems with excessive CPU usage and failed to acknowledge disk quotas set by the users (if not enforced by the operating system). One problem is that the *Freenet* server is implemented in Java. This requires every node to run a Java Virtual Machine (JVM) all the time. The memory requirements of a JVM are often not tolerable for many potential nodes.

5.4 Mojo Nation

Mojo Nation is a distributed file sharing system where hosts need to provide bandwidth and drive space to earn *Mojo*, or micro-credits. *Mojo* can then be used to request services from other hosts. This credit system protects the network against *freeloaders* (people who use the network but do not contribute to it). This approach is similar to GNUnet's ranking scheme, but does not allow use of excess capacities for new users, nor does it provide anonymity.

Mojo Nation is, like Napster, a commercial product. The available documentation does not make any specific claims about how authentication is achieved.

5.5 Free Haven

Free Haven [10] provides an infrastructure for anonymous publication by hiding the identity of the publisher, the clients and the location of the document. Free Haven is designed for anonymity and persistence of documents and not for frequent querying. Anonymity of senders and receivers is achieved using remailers and mixnets. Free Haven is decentralized; servers can be added dynamically. Malicious or dead servers can be detected and will be considered unreliable.

Free Haven expires documents after a fixed amount of time that was set by the author. After that time, the document is deleted. Documents are split into n shares, where k out of n shares are sufficient to reconstruct the entire document

[4] The content may still be valuable to the network as participants in other countries may not be allowed to access it using common Internet technologies.

[19]. Shares are transfered by trading. Servers make promises to store a share until it expires. In order to prevent hosts from breaking this promise, servers are audited using a 'buddy system'. For each share, a 'buddy' share on another host exists. This pair of hosts then monitors each other.

To retrieve a document, the user must obtain the key that was used to sign the document from other sources than Free Haven. Requests are broadcast from the server to all clients. The server may choose to sign the broadcast request, increasing the chance that other servers can perform accounting to guard against DoS attacks. In Free Haven, servers encrypt and send a share of a document after receiving a request.

Free Haven fails to account for requests, it only accounts for storage space. The concept that publishing costs while retrieving is free is unfortunate. Because shares seem to be stored in plaintext (only the transfer is encrypted), hosts can exercise editorial control. The 'buddy system' imposes a significant overhead. Nevertheless, it fails to give strong guarantees that hosts will actually behave well. Because hosts pay for storage space, the two buddies would both profit from dropping the share silently. Unlike the *GBlock* encoding, the k out of n share reconstruction scheme does not allow reconstructing, a file mixing shares from independent insertions (see section 3.2). Free Havens defence against flodding attacks (i.e. signing of requests) voids anonymity. The requirement that the users obtain Free Haven keys from other sources, puts an extreme burden on the user.

5.6 Generic Problems

All three implementations suffer from the problem that a single file is often stored as a whole or in large pieces in the network.[5] This is a particular problem for huge files which may require the server to provide excessive bandwidth on a single client. Furthermore, the distribution of search queries is a common problem. *Napster* and *Gnutella* search for filenames, which might not be appropriate. *Freenet* requires unique keys which may be non-trivial to guess. The keyservers inside of *Freenet* try to solve this problem by providing indices to all available keys. The disadvantage of the keyservers is that they must be maintained; additionally, they often index content which is no longer available. As far as we know, *GNUnet* is the first system that allows boolean queries and provides anonymity as well.

Finally, neither of the networks can make any guarantees on how long content will be available after the initial node which inserted the content goes offline. For *Napster* and *Gnutella*, this usually means the end of the content in the system. Even if the content has migrated to other servers in *Freenet*, the decision to delete the content from these servers may come at any time since replacement of content follows the Least Recently Used algorithm. *GNUnet* allows the nodes to operate somewhat less arbitrarily, as nodes are allowed to try to be clever about

[5] Freenet and Free Haven allow splitting of files, but this feature seems to be used only to split the file into huge blocks.

which content they are hosting. As rare content is usually more valuable, hosts could optimize their storage policies. However, heuristics for this must still be developed.

Achieving guarantees on how long content is preserved has been exploited by other projects. For example, *Free Haven* deploys a Buddy system where content is split into two parts and each one periodically checks that the other is still in the network. Still, no fully distributed system can guarantee that content will never be lost.

6 Future Work

User feedback is the only real way to determine with any degree of certainty that content is actually valuable. Yet, this is particulary hard to implement in an anonymous network. In the future we plan to propagate a user's evaluation of content back along the path the data originated from. Of course, back-propagation should be decided based upon the available bandwidth, the ranking of the hosts involved and the evaluation of the user. Since it is possible that the user who ranked the content is malicious, only content rankings from trusted hosts should be considered.

We also plan to conduct benchmarks to measure the actual performance of *GNUnet*. These results could be beneficial in optimizing several parameters of the network. The selection of the set of hosts a node is communicating with and a better heuristic to determine a subset of these hosts for forwarding queries are interesting places for optimizations.

7 Conclusion

GNUnet derives many of its conceptual foundations from file-sharing services like *Freenet*, *Gnutella*, and *Napster*. *GNUnet* gives stronger security guarantees than *Freenet* and is more resistant to attacks than *Freenet* or *GNutella*. *GNUnet* is the first system that simultaniously provides anonymity, searchability and accountability.

References

1. Carnivore.
2. E. Adar and B. Huberman. Free riding on gnutella. Technical report, Xerox Parc, Aug. 2000.
3. S. Bellovin. Security aspects of napster and gnutella, 2000.
4. S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, 1989.
5. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
6. I. Clark. A distributed decentralised information storage and retrieval system, 1999.

7. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*. International Computer Science Institute, 2000.
8. D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP Vol.II - Design, Implementation & Internals*. Prentice Hall, Englewood Cliffs, NJ, 1991.
9. P. Criscuolo. Distributed denial of service - trin00, tribe flood network, tribe flood network 2000, and stacheldraht. Technical Report CIAC-2319, Department of Energy - CIAC (Computer Incident Advisory Capability), Feb. 2000.
10. R. Dingledine. The free haven project. Master's thesis, Massachusetts Institute of Technology, 2000.
11. Clip2 DSS. Gnutella protocol specification v0.4.
12. P. Raghavan G. Pandurangan and Eli Upfal. Building low-diameter p2p networks. In *Proc. of the 33rd Annual Symposium on Theory of Computing (STOC 2001)*, 2001.
13. K. Hickman. The ssl protocol. internet draft rfc, 1995.
14. Mark Lewis. Metallica sues napster, universities, citing copyright infringement and rico violations, 2000.
15. Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.
16. Napster. Frequently asked questions.
17. Mojo Nation. Technology overview, Feb. 2000.
18. D. Plonka. Uw-madison napster traffic measurement, Mar. 2000.
19. Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
20. Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.
21. IETF Secure Shell (secsh) Working Group. Secure shell (secsh) charter., 1999.
22. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.
23. P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Rfc 2136: Dynamic updates in the domain name system, 1997.
24. Stratton Oakmont vs Prodigy Services Company, 1995 N.Y. Misc. Lexis 229, (N.Y. Sup. Ct. Nassau Co., 1995).