GNET

Christian Grothoff[*] Ioana Patrascu[†]
Krista Bennett[‡] Tiberiu Stef [§] Tzvetan Horozov[¶]

Version 0.5.2
June 13, 2002

Abstract

This paper describes *GNet*, a reliable anonymous distributed backup system with reasonable defenses against malicious hosts and low overhead in traffic and CPU time. The system design is described and compared to other publicly used services with similar goals. Additionally, the implementation and the protocols of *GNet* are presented.

1 Introduction

GNet strives to provide a reliable anonymous distributed backup system. It consists of several layers. The communication layer provides certain guarantees for the higher protocols; it provides authentication of the participating parties and confidentiality for the data (similar to SSH). On top of this layer, a first simple service—distributed anonymous file sharing—is implemented.

1.1 Design Goals

GNet central goals are anonymity for the users, deniability for all participants and decentralization. In order to achieve this, *GNet* aims towards maximal distribution of content. Efficiency is a secondary goal, which is partially achieved by not specifying the exact behaviour of nodes. Instead, the *GNet* protocols say what is allowed and the network rewards nodes for successful behaviour. This way, optimizations in the routing of queries can be performed in a host-specific way, without violating the network's protocols.

As the optimal solution may depend on the requirements of the higher-level problem (e.g. anonymity may be improved using indirections), the higher level protocols must be involved in routing decisions, the nodes

[*]grothoff@cs.purdue.edu
[†]patrascu@cs.purdue.edu
[‡]klb@cs.purdue.edu
[§]tstef@cs.purdue.edu
[¶]horozov@cs.purdue.edu

must to some extend know what they are dealing with. Thus the pure communication layer is extended by a layer that knows a little bit about the data that it is handling. The higher level protocols should also be able to detect misuse of *GNet* (attacks) and react, crediting hosts for good behavior and limiting access for malicious nodes.

CPU time is a critical resource, especially on the busiest servers. Thus *GNet* aims to reduce encryption overhead for the servers and intermediaries. For clients executing the user-level part of the protocols, reasonable amounts of CPU time should be available.

As available data usually exceeds available disk space, data must be discarded somteimes. *GNet* should be capable of distinguishing important (i.e. frequently accessed) data from garbage. Furthermore, data provided by participating hosts should have a higher chance of survival than content from non-contributing (potentially malicious) hosts.

If a host receives more queries than can be answered with available bandwidth, it should drop the queries from hosts that have earned less credit.

1.2 Outline of the paper

Related work is presented in section 2. In section 3 we describe the low-level communication protocol and how identification and confidentiality are achieved. Section 4 describes the high-level file sharing system. In Section 5 we discuss rating of hosts and defenses against attacks. Section 6 describes the implementation. Section 7 gives details on the protocols used in *GNet*. Section 8 gives some data gathered from initial test runs of *GNet* and section 9 concludes the paper.

2 Related Work

Currently, three major systems are used on the Internet with similar functionality to *GNet*. *Napster* [1] is a distributed file sharing system currently limited to `mp3` files where file distribution is coordinated through a central server. *Gnutella* [2] is a file sharing system based on the HTTP protocol without a centralized lookup mechanism or support for encryption. *Freenet* [5] 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 in these implementations.

2.1 Napster

Napster's major drawback is that it is easy to find out everything about the communication by simply sniffing the traffic. As *Napster* is centralized, it is particulary easy to shut down the system or to disclose user information. 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 add content manually 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 for users who want to share music.

2.2 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 main drawbacks of the system. Furthermore, the communication is not anonymous. The protocol leaks information on search queries and identifies the hosts providing the content.

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. This could lead to nodes with more resources essentially becoming central servers, even though the network is "decentralized".

2.3 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 favour 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. This may shift some of the burden away from individual server owners in terms of liability. Yet, *GNet*'s solution where a single host usually has no means of reconstructing the whole file (even assuming the host can guess the key) seems to be better. 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, 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. Recently, keys that allow content updating have been introduced.

Search 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 with 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 *GNet*, the content propagation path is fixed, individual nodes can not decide if they should indirect the reply or short-cut the reply path. Thus *Freenet* provides similar anonymity compared to *GNet*, but uses more bandwidth to achieve this.

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.[1]

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.

2.4 Mojo Nation

Mojo Nation is a distributed file sharing system where hosts need to provide bandwidth and drive space to earn *Mojo*, micro-credits. *Mojo* can then be used to request services from other hosts. This protects the network against *freeloaders* (people that use the network but do not contribute). This approach is similar to GNet's ranking, but it does not allow using excess capacities for new users and does not provide anonymity.

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

2.5 Generic Problems

All three implementations suffer from the problem that a single file is always stored as a whole in the network.[2] 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 be not appropriate. *Freenet* requires unique keys which might be non-trivial to guess.

[1]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.

[2]Freenet allows splitting of files, but this feature seems to be rarely used.

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, *GNet* is the first system that allows boolean queries **and** provides anonymity.

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 been migrated to other servers in *Freenet*, the decision to delete the content may come at any time since replacement of content follows the Least Recently Used algorithm. *GNet* allows the nodes to operate slightly better as nodes are allowed to try to be clever about which content they are hosting. As rare content is usually more valuable, hosts could optimize their storage policies. Yet, heuristics for this must still be developed.

Achieving guarantees on how long content is preserved has been exploited by other projects. For example, the *Freehaven Project* deploys a **Buddy** system where content is split into two parts and each one checks periodically that the other is still in the network. Still, no fully distributed system can guarantee that content will never be lost.[3]

3 Distributed safe communication

The basic communication mechanism used in *GNet* is very similar to SSH. 2-key RSA encryption is used to exchange a session key between two nodes. Each node in the network is identified by its public key (or as a short handle for the public key; a hashcode of the public key is used).

The major difference between SSH and *GNet*'s lowest layer is the use of UDP instead of TCP. TCP suffers from overhead introduced by the initial handshake, guarantees of order of packets, and packet loss. *GNet* does not make such guarantees; packets may be lost. This is because most higher-level GNet protocols do not rely on these guarantees. When a search-query is broadcast to 20 hosts, it does not matter if only 19 receive it. When a reply does not arrive, one of the two things may happen. If the host has already received the requested information from another host, it will simply ignore the failed request. Otherwise, the host may re-issue the query.

In *GNet*, the higher-level services have the responsibility to cope with lost packets.

3.1 Issues

Man-in-the-middle

One of the main issues with SSH is the possibility of a man-in-the-middle attack when the public keys are exchanged. Interestingly, this attack should not have an impact on *GNet*. Hosts are identified by their secret key, and that is all that matters. IP addresses, port numbers, locations,

[3]Actually, this probably also applies for centralized systems. Make your backup now. :-)

are all irrelevant properties. If Mallory intercepts the communication between Alice and Bob, they will both exchange data with Mallory—and judge him by his behavior (potentially affecting his reputation). If he answers queries and behaves well, they will give Mallory credit for that. If Mallory floods their nodes with requests, they will at some point refuse to connect with him as his reputation will deteriorate.

As long as Alice and Bob just want to communicate with someone (and get to know someone), Mallory cannot stop them. In *GNet*, nodes never want to communicate with a **specific** host in the sense of an IP or other network address. They only want to communicate with a node that has a particular secret key, and these secret keys are learned over time.

3.1.1 UDP

Using TCP instead of UDP has the big advantage that feedback ciphers can be used without problems. With UDP this is not practical, as packets may be lost or arrive out-of-order. Thus, for each packet, a new initial sequence number is chosen.

The lifetime of a session key is also well-defined for TCP: it lasts as long as the connection lasts. For UDP, there is no connection. Thus, session keys expire after a fixed amount of time. Distributing new keys may be done at any time by either of the two hosts sharing a key. Of course, this may lead to a few lost packets. For example, if one host sends a new session key and the other host continues to use the old key, packets sent by the second host will look like garbage until both hosts successfully use the same session key. The same will happen if a packet with the new session key is lost. As stated previously, however, there are no guaranteed deliveries.

3.1.2 Dynamic IPs

GNet nodes may have a dialup connection and change their IPs rather frequently. Thus, each *GNet* node must not only know the public key of other nodes, but also their current IP and port. When a node (re-)joins *GNet*, it sends a message to other known nodes containing its current IP and port and a timestamp indicating how long this address will be valid. This data is then signed with the private key of the joining node. This signature is required because otherwise an attack is possible where malicious hosts send out incorrect sender-addresses and thus hinder communication between well-behaved nodes.

3.2 Encryption primitives

GNet uses RSA for the asymmetric encryption and Blowfish for the symetric exchange. RSA was chosen because it seems to be the most suitable (i.e. difficult to break and patent-free) choice for our purposes. Blowfish was chosen because the protocol is intended for implementations in software, it is freely available and is fast.

RIPE160 was chosen for the hash primitive because this is the longest (in terms of the output) hash-function supported by OpenSSL. The 160 bits make collissions very unlikely, even with lots of content in the network. CRC32 was chosen as a checksum for the data because there are exactly 32 bit left in the 1024 byte inodes if each inode stores 51 RIPE160 hashes.

3.3 Efficient communication

Since UDP is used, the delay of the network connection can not be measured by simply sending packets (especially as the receiver may not reply instantly or at all). Thus, the best way to obtain routing information may be to look at the IP addresses and assume that closer addresses are in fact closer to the local node. The recent *Code Red* worm was fairly successful using this technique.

Of course, this heuristic does not preclude better implementations gathering further information, (e.g. by using `ping` (ICMP) or new *GNet* subprotocols). It is the responsibility of every node to optimize its behavior. After all, each node is evaluated by other hosts' perceptions of its performance.

Each packet sent in *GNet* has a certain overhead. This includes encryption and decryption, communication of the packet header and the actual processing of the packet. In order to keep the number of packets low, *GNet* buffers outputs for each target node, trying to achieve optimal packet sizes. By default, the optimal packet size is 1472 bytes (the optimal size for ethernet).

Buffering outgoing data per host is particulary useful for requests, as the node can freely choose to whom to send the request. Good candidates for requests are hosts with pending output requests that have not been sent because the minimum packet size was not reached.

The heuristic used to decide whom to send data is open for improvements. Changes to these implementation details in a node in order to improve performance are **not** a violation of the *GNet* protocol.

3.4 Joining GNet

A node that wants to join *GNet* must know at least one other node. This may be achieved by providing a list of initial nodes together with the distribution or by publishing lists of known nodes in newsgroups or on public webpages. *GNet* uses "HELO" messages to exchange information about nodes.

As "HELO" messages contain the connection information (IP and Port), these messges must be signed to avoid attacks by malicious hosts that put fake sender addresses. As addresses may change (e.g. in case of Dial-up connections), these signed "HELO" messages must also be timestamped with an expiration date. This prevents malicious hosts from forwarding outdated "HELO"s.

After sending an initial "HELO" message to the known nodes, these nodes will include the new node in their queries and eventually forward other "HELO" messages. Forwarding "HELO" messages is performed randomly (the node chooses randomly two hosts that it knows and forwards one of them the address of the other).

4 Ranking

4.1 Malicious Host Detection

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 should be able to join the network at any time without signing up with a central authority, the detection of malicious hosts must also be decentralized.

As a first step, every node must evaluate the behavior of the other nodes that it communicates with. New nodes that join the network start as *untrusted*. Those 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 Denial-of-Service attack by an overwhelming number of malicious hosts cannot be fought off if the malicious hosts have more bandwidth.[4] 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 are in the network for a while, the other nodes should monitor their behavior:

- Did they keep the content they were given?
- 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 are hardly ever requested. These established nodes keep copies of the files. A few days later, they request the files from the new node. If the node still has the files, its rank is increased. Next, parts of the files may be dropped from the database of the established nodes. Only a few pieces of the file are kept. If a few weeks later the new node still has these pieces, its rank is increased again. Availability and bandwidth of a node

[4]This is a general problem with the current internet architecture.

should be kept as separate criteria for the node evaluation. *GNet* can use these to decide where to store which content.

It is essential for *GNet* that it is possible to increase the ranking of well-behaved hosts without decreasing the ranking of other hosts to the same amount (no zero-sum of the rankings). Otherwise, all hosts would always have credit "zero" as nobody has credit to start with. Still, 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

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

is satisfied where $\epsilon > 0$ should be small. *capacity* here 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 is not malicious, the network "works".

The current implementation

Hosts in *GNet* do not have a global credibility. Instead, each node in *GNet* keeps track of its *opinion* about all the other nodes it has contact with, based on their previous behavior.

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 must 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.*

If the node processing the query has excess bandwidth (and CPU time), it may decide to **not** charge the sender of the query. After all, the query did not cost it any performance. This is important 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 very busy, it should discard queries with low priorities (and charge the nodes for asking questions). Hosts asking queries with a priority higher than their own ranking, the policy decreases the ranking to the allowed ranking. Host rankings are kept for each pair of nodes that know each other.

4.2 Content ranking

In order to prevent malicious hosts from inserting garbage into the network (e.g. /dev/random), content must be ranked. The only way to determine if the content was valuable is to ask the user. The nodes may then decide to propagate the evaluation back on the path the data came from. Of course, the back-propagation should be decided on the available bandwidth, ranking of the hosts involved and the evaluation of the user.

Also, the user that ranked the content may be malicious. Thus, only content rankings from trusted hosts should be considered.

Apart from user feedback, the number of requests for the content should be considered. If several keys on a host match (e.g. because the same key has been reused for different files), the hosts may want to consider discarding content that—even if not ranked lowest—has low feedback scores.

The initial ranking of content should be limited by the trust level of the host inserting the content. Mind that any propagation of content from one node to another cannot be distinguished from content insertion. The user inserting the content into *GNet* should be able to tell the client how important the content is for the user. The node can then ask other nodes to copy the content—and copying should lower the rank of a node.[5]

Ranking Content

Content ranking is an important feature of *GNet*. It allows *GNet* to make better decisions on which content to discard, even if there can still be no strict guarantees that content will be preserved. As content with low ranking is discarded, this content will have a high survival rate, regardless of whether it is requested frequently or not.

Of course, over time, the 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 to keep their nodes operational and the content intact.

Ranking of content occurs at two points. First, when the content is inserted, the user can specify how important the content is. Other nodes may acknowledge that priority (based on the question of whether or not they trust the origin 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 (and maybe some low value for requests answered with priority zero) should give an acceptable heuristic.

4.3 Inserting Content

In order to distribute content on *GNet*, the first step is to provide a node to the network. If the content is important, it is required that the node stays active in the network for a while before distributing the content in order to achieve a certain trust level.

Even with the lowest level of trust (i.e. host ranked as malicious), the new node can ask other nodes to copy its content. Depending on available bandwidth and disk storage, the other nodes may or may not do so. Established nodes might want to take the content in order to maximize the number of queries they can answer (since correct answers to queries

[5]exchanging content (trade) should not

increases their ranking). Still, they would probably rank the content at the lowest end and discard it if it is not requested.

If the host where the content is inserted has a higher ranking, it may be able to decide between asking other hosts to copy the content and exchanging the content for different data instead. Exchanging content is important in order to increase the entropy in the content distribution. The more randomly the content is distributed, the harder it will be to determine the original provider of the content.

Content with a poor rating may be discarded if a node runs out of space and more promising content is offered. Nodes may keep track of where the content they have originated from, of course, after a time, that information may become irrelevant to judge the host that send the content and then the information where the content came from may be discarded.

5 Anonymous file sharing

5.1 What is anonymity?

Anonymous communication is commonly perceived as communication for which it is impossible for third parties to identify the participants involved. For us, anonymous communication is supposed to guarantee that a data transfer cannot be connected with the real sender or receiver but only with the immediate hosts participating (which might just be intermediaries). Furthermore, the communication should be confidential in the sense that only the receiver knows the content of the message. The sender and the intermediaries should not be able to determine the actual content. Also, the original submitter of the content should be able to plausibly deny that the content originated from him or her, even if all nodes (except for the submitter's node) that the content was going through were malicious and kept records of all their transactions.

This anonymity requirement is difficult to achieve, especially if the communicating parties are supposed to identify themselves for the lower-level protocols that provide identification and confidentiality. This is particularly true when malicious hosts are involved that have the ability to expose the packets that they can decrypt.

5.2 Encrypting Queries

GNet tries to achieve anonymity by making it nearly impossible for content providers to identify what kind of content they are serving. Furthermore, queries are encrypted by the client and can not be deciphered by the intermediaries or the final server that hosts the data.

Making it impossible for intermediaries to decrypt the content is achieved by hashing the query string using a one-way function [6] and using the hashcode as an index for the requested file. In order to allow reasonable search queries, hashcodes can be combined using the logical operators `and`, `or` and `not`. This increases the searchability of *GNet* over systems like *Freenet*. On the other hand, using a combination of short words makes dictionary attacks much easier.[6] Yet, the choice for the keywords is up to the users; they must decide between usability (short keywords) and security (long keywords).

5.3 Distributing the content

The data stored in *GNet* is split into small chunks which are individually hashed. These hashcodes then serve as keys for the distributed parts of the file. The hashcodes are grouped into indirection nodes, similar to UNIX inodes. Generally speaking, since content is torn apart, no single provider (except for the one inserting the content at the beginning) will host files in their entirety. Furthermore, as indirection nodes and data nodes are not easily distinguished, it is hard for the content provider to determine if clients are requesting data or meta data.

Splitting the content into uniform pieces of 1k makes it also a lot easier to exchange content. Unlike other systems where always whole files must migrate (limiting the ability of large files to get moved) migrating parts of a file in *GNet* takes just a single packet to be send.

In general, content should migrate towards requesting sites. Frequently requested content should be stored on high-bandwidth servers, while rarely accessed data should migrate to low-capacity hosts.[7]

Splitting up the data into small pieces also has the advantage that downloading files from *GNet* actually makes use of the distribution; the file can be downloaded in parallel from many hosts, incurring little cost to each host (except for the receiver). The disadvantage is that content must be multiplied as hosts may not be available (they may be offline). On the other hand, this also helps to ensure that content is not immediately lost when a node goes offline.

5.4 Content storage

Storing content in *GNet* must satisfy three basic requirements:

1. storage space should be kept small
2. hosts providing space should have no means to decrypt content obtained from other hosts
3. content stored unencrypted locally should be accessible to *GNet* on demand

The first requirement is obvious. If it takes 100 MB to store a 1k file, the system is useless. The second requirement is supposed to protect participants from being helt liable for content transmitted through their nodes. The third requirement is supposed to address an entirely different issue. In some cases, host operators may want to just share files that are on their harddrives anyway, without need for protection against adversaries that may obtain access to the host. For these cases, *GNet* should be able to encrypt this data at the time where it is requested. This should of course not be visible to other users, so they will still obtain small pieces

[6] Compare "h(Marx) `AND` h(Kapital) `AND` h(Das)" against "h(Marx: Das Kapital)".
[7] Again, the *should* indicates that this is up to the nodes involved to decide.

of the file each time. As the file is stored in plaintext and encrypted only for the transmission, the provider can use the unencrypted files and safe storage space. Other systems, like *Freenet* would require a second copy of the data. *GNet* only requires indexing of the data and can then encrypt and serve the files on-demand.

To achieve the above goals, three modes of operations are planned. The first mode, which is the default, provides a reasonable tradeoff between the three goals and also serves as the basic model for the other modes. As described in section 5.3, *GNet* splits the files into chunks of 1k. Each of these chunks is than individually distributed.

In order to store a 1k block B, *GNet* first computes $H(B)$ and $H(H(B))$. The block is then encrypted (using a block-cipher) using $H(B)$ as the key (and another part of the hash as the initialization vector). The block is then stored under a file with the name $H(H(B))$. In this form, the data can be forwarded, but the local node (not knowing $H(B)$) can not decrypt it. The only way to decrypt the file is by guessing $H(B)$. This way of encrypting the file also guarantees that if the same file is inserted by multiple parties, they will yield the same encrypted files and thus just increase the availability of the data.[8]

The indirection nodes ("inodes") would then be

$$I = H(B_1), \ldots, H(B_{51}), CRC32(B_1, \ldots, B_{51})$$

and I would be again treated as a normal block, that is encrypted with $H(I)$ and stored under $H(H(I))$. The root-node of the indirection tree would then be encapsulated in a special node that is encrypted with the hash of the keyword that was supplied by the user and stored under the hash of the hash of that keyword. The root node can additionally contain a description of the contents of the file, allowing users to decide between multiple results for the same query depending on the description.

This scheme requires storage space m that is only 2% above the size of the initial file n:

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

It yet provides security for the sender and receiver, as the encrypted content can not be decrypted by anybody but the receiver (assuming that the hashcodes can not be guessed).

The second mode of operation is a simple extention to the first mode. By computing the hashes of the 1k blocks, a file can be indexed.[9] If an incoming query matches the index, *GNet* can encrypt the local file on-the-fly. Even better, *GNet* only needs to read and encrypt the matching 1k block, the rest of the file (if not requested by other queries) is not required for the encryption.

[8]The encryption scheme in *Freenet* would lead to duplication of the content. It would be impossible to detect duplicates, storage space would be wasted.

[9]The inodes should be stored using the first mode, but their size is negligible.

Both, the first and the second mode have the advantage that the space and communication overhead are fairly low. Yet, an attack is possible. If the adversary can **guess** the exact (!) contents of the file, the adversary can compute the encryption and find out that the host was storing that file (or that the receiver was asking for that particular file). The user has no means to protect himself against this kind of attack–as opposed to the adversary guessing the keyword where the user may choose a stronger keyword, the user may not have the choice to manipulate the content to make it impossible for the adversary to find out about it.

Yet, this problem can be solved with the third mode. By xor-ing the file that should be stored in *GNet* with a one-time-pad and then basically storing the one-time-pad and the xor-ed file into *GNet* using the first or second mode, this kind of attack is impossible. In this case, the root-node contains two inodes, one for the one-time-pad and one for the xor-ed file. The problem with this approach is that it at least doubles the storage space required. If another user inserts the file, the duplicates can no longer be detected and even more space is wasted. Also the amount of network traffic is doubled. Thus this mode should only be used if this level of security is really required.

Another way to increase security is to requiring multiple root-nodes with different keywords to decode the actual file. This should not affect the user if the query actually only matches an **and**-query. Except for mode 2, the *GNet*-node is not concerned with these issues, as they are only taking place in the *gproxy*-layer for anonymization.

5.5 More on queries

As mentioned before, the actual query for a datum matching Q could be hidden by hashing Q first. As $H(Q)$ is used as the key for the decryption (as described in section 5.4), $H(H(Q))$ is the obvious choice for the query published. Yet, this approach has a slight 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 $H(Q)$) can not verify if 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 back through the chain that the original sender was malicious. As intermediaries can not 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 by sending out garbage to the network.

This 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. As the sender can not guess $H(H(Q))$ it can be assumed that the sender had matching content at some point. This can of course not prevent malicious hosts to create 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. Yet, this is similar to inserting garbage under that keyword into the network. As no software can distinguish valuable content from garbage in general, this is not a design flaw. Yet, it demonstrates that content moderation (feedback, ranking) is required. The triple-hash scheme just makes it more difficult to insert garbage, it can not make it impossible.

5.6 Query propagation

Query propagation must solve two issues. First of all, queries must be forwarded in a manner such that they do not loop. Secondly, the routing of queries should try to be efficient and should scale. Elaborate schemes have been designed in routing content and queries in a way that lookup is fast. A particularly interesting approach is chosen in *Freenet*, where similar content (in terms of similarity the hashed key) is stored on topologically close nodes. Thus nodes can guess in which area of the network content may be stored. Currently, *GNet* does not attempt to use such advanced schemes, but they may be implemented in the future. [10]

In order to forward queries, three issues must be resolved:

1. selecting a subset of the known nodes as targets for forwarding
2. selecting the priority of the forwarded query
3. book-keeping of forwarded queries that are routed back through the forwarding node

In *GNet*, each query has associated with it two integer values that help the node decide each of these questions. The first integer value is the **priority** of the request. It indicates how important the request is for the node sending (or forwarding) the request. A value of zero indicates that this request should only be honored if excess bandwidth is available.

As the first step of processing, the node that receives the request considers the ranking of the node that sends the request and adjusts the priority. If the priority was higher than the credit of the sender, the priority is reduced to the credit the sender had. Then, the credit of the sender is decreased by the remaining priority of the query (charge for service) if the bandwidth available at the moment is low.

Now the node checks if the content is available in the local storage and eventually sends a reply. If the node is busy and the priority is low, the node may decide to not send a reply even if it has the data itself.

If the node does not have the content, it considers the second integer that accompanies the request. This second integer gives the **time-to-live** (TTL) for the request. The originating node may have set it arbitrarily high, so the local node may first reduce it to a reasonable value. If the TTL is zero, the request has expired and is dropped. If the TTL is greater than zero, the node adds the query (sender, hashcode and time-to-live) to its local query table. If a query with higher TTL already exists in the table, the new query is added but not forwarded: a reply from that earlier query may come in at any time—or the query has looped back to the local

[10]Storing content on topologically close nodes has a disadvantage, however; if topologically close nodes are also geographically close, the network is then vulnerable to localized failures (i.e. power surges, earthquakes, etc.). In *GNet*, this could be a particular problem, as portions of files might become entirely unavailable if stored in a geographically centralized location. This also makes it harder to infer which kind of content a node stores.

node. In general, the forwarding table is used to store information about forwarded requests such that the node can forward replies later.[11]

Depending on the new priority of the request and the available bandwidth, the node selects n random[12] other hosts for forwarding. The node than decreases the TTL by a reasonable amount (greater than the time it expects for network delay and processing of requests). It decreases the priority of the request to $\frac{p}{n+1}$ and forwards the query with the adjusted TTL and priority to the n hosts.[13] The number for n depends on the current network load. If the load is high, the node may decide to forward the query to fewer hosts.

The TTL of the packet is decremented over time on each node (e.g. $TTL--$ every $30\,s$). Once the TTL reaches zero, the entry is removed from the forwarding table (the query is considered unsuccessful).

5.7 Benefits of this approach

GNet allows applications that go far beyond the possibilities of *Napster*, *Gnutella* or *Freenet*. The searchability allows large portions of the WWW to migrate to *GNet*, corporations may decide to store important data decentralized in *GNet* and users will like the service because it offers privacy that will otherwise be difficult to achieve given the current configuration of the Internet.

WWW^2

Why should Internet sites migrate to *GNet*? First of all, *GNet* will allow them to distribute their content over the globe. There will be no need for a central server that gets millions of hits in a short time. Instead the content will be automatically distributed, reducing download times for the users. Furthermore, for providing content that is frequently accessed, the nodes of the content providers will earn credit in *GNet*.

Users will also appreciate the ability to retrieve Internet content without concerns that third parties will build profiles of their browsing habits. With the advent of targeted Internet advertising companies (e.g. *Doubleclick*, *Passport*), consumers are increasingly concerned about the privacy of their personal data. *GNet* assures that the link between provider and consumer is completely anonymous, thus severing the ability of marketers to use private data for their own purposes without the explicit consent of the user.

[11]If the node (depending on the available bandwidth decides not to indirect replies, it may skip this step.

[12]Better strategies for selecting nodes may be used in the future.

[13]Dividing by $n + 1$ guarantees that forwarding nodes are not charged more than they charged the node that send the query. It also guarantees that even $n + 1$ cooperating hosts can not keep the node busy by indirecting queries via the node (DOS attack all against one) because the local node makes $\frac{p}{n+1}$ profit in credit for each request.

Domain Name Service (DNS)

Small content providers currently face another issue: DNS. In order to achieve visibility, they not only have to pay a fee to the DNS monopolies, but must also face lawyers trying to sue them for the use of potentially unknown *trademarked* names. Domain squatting is sometimes a nuisance even for mid-sized companies. The site *www.gatt.org* is a great example. In *GNet*, domain names are meaningless. The more often a site is requested, the faster it will show up. Multiple results for the same query are possible.

ISPs

GNet credit could be traded—for money. For example ISPs may be ranked according to their *GNet* ranking. As the servers of the ISPs will forward queries from the ISPs clients with that ranking, the clients of the ISPs will get service in accordance to the ranking of their ISP. ISPs will have to ways to achieve higher rankings. Either by increasing their bandwidth or storage space to directly earn credit from *GNet* or by trading credit (e.g. with content providers or with other ISPs that decide that they have a higher ranking than they need).

Backup

Companies could use *GNet* to backup important data. They profit from the anonymity (Microsoft can not decide to delete data stored for IBM) and the distribution: there is no single point of failure. Instead of building redundant systems, the network will achieve distribution and redundancy automatically.

This is particularly important since local redundancy does not always help; considering the recent power surges in California, high availability of content is best achieved on a global scale.

6 Implementation

A system like *GNet* must solve several implementation issues of varying sizes. This section tries to address a few of them.

6.1 System architecture

In order to allow small machines to act as *GNet* nodes, the central server process of *GNet* is a small C program. Its main functionality is matching incoming requests (keys) with the local data, sending the data and controlling limitations set by the user (bandwidth, memory, CPU). The only encryption the server process is involved in is the identification of the remote hosts and the encryption of the communication channels, similar to SSH. Furthermore, the server should provide metadata of the communications to a control process. This control process should then rank the content, detect malicious hosts, and then define the policies for the main server process.

6.2 User interface

The user interface (UI) should be split into three parts. A simple text-based UI should allow insertion of content into the local node. Another text-based UI should allow content requests and ranking of the results. Gproxy is a proxy that allows browsing *GNet*. Gproxy performs requests and provides a WWW interface. Gproxy is responsible for the decryption of content that is retrieved from the network.

6.3 Protocol tunnels

In order to avoid *GNet* becoming a victim of modern firewalls, *GNet* hosts should be able to tunnel their communications in protocols that are unlikely to be abandoned by the Internet community. Examples for these protocols are HTTP, SMTP, and FTP. If *GNet* nodes can substitute the underlying UDP protocol layer and communicate through these common channels, firewalls and other restrictions may be rendered useless.

The first generation implementation of *GNet* will not incorporate these features, but programmers are asked to keep them in mind when designing the system.

7 GNet Protocols

This section describes the protocols used by *GNet*. As a general rule for inter-nodal communication, any node may refuse to reply at any point. This may be because packages were lost, because the node is busy, or because the node refuses to answer because of distrust (or any combination of these).

The underlying assertions for the protocols are as follows:

- All nodes have a synchronized time (UTC), but small time-differences must be tolerated (up to a few minutes). GNet does not provide means for time-synchronization.
- Any node may be malicious and fail at any time; the network is not reliable and does not provide any authentication.
- The private hostkeys are always secret and the implementation of the cryptographic primitives is sound.

7.1 UDP layer: confidentiality and authentication protocols

This section describes the HELO and SKEY messages of the GNet protocol. They are supposed to establish a confidential and authenticated communication channel.

HELO

This protocol describes how a new node joins *GNet*. It is assumed that the node knows as least one other participating node of the network. This is achieved by giving the node the IP and portnumber of an existing node in its data files. A is the new node, B is an existing node in *GNet*.

1. $A \rightarrow B$: $E_B(A, S_A(@A, t))$ where A is A's public key, S_A is the message signed with A's private key, $@A$ is the current network address of A and t is the time for which this address is valid. E_B is encryption with B's public key, B is known in advance.

   Cryptographic primitive: public key crypto, 2048 bits, RSA, same key for signing and encryption.
2. B verifies the signature and adds A to his list of known hosts, together with A's address, the timestamp and the signature.

Later, B can decide to tell another node C about A. The timestamp t that A used in the original message limits the timeframe in which these propagations are allowed (A may want to change $@A$).

3. $B \rightarrow C$: $E_C(A, S_A(@A, t))$.

SKEY

After A and B have exchanged their public keys using the HELO protocol, they can exchange session keys K. This may be initiated by either A or B:

1. $B \rightarrow A$: $B, E_A(K), t, S_B(H(E_A(K), t))$ where K is session key that was chosen by B and that was created at time t.
2. $A \rightarrow B$: $E_K(H(K))$ (acknowledgement)

A and B store K in their current connection table. Further communication is encrypted with K:

1. $A \rightarrow B$: $E_K(M, t)$.
2. $B \rightarrow A$: $E_K(M, t)$.

Each message contains a timestamp t that describes how long this message is supposed to be valid. This way, an attacker can not do much harm with replay-attacks.

7.2 GProxy

GProxy is a service on top of the basic GNet layer. GProxy allows users to share files. The central goals of GProxy's design are:

- anonymity: neither the sender nor the intermediaries should be able to find out what the receiver is looking for (except if they can guess it).
- distribution: no single host should be forced to share the whole file, small (1k) pieces must be enough.
- security: the host storing the data should have no way to decrypt the data and thus be able to deny knowledge of the contents.
- efficiency: even if encrypted, the same content (eventually inserted independently by two parties under different keys) should yield the same filenames with the same content, reducing storage requirements.

- on-the-fly encryption: it must be possible to share parts of unencrypted files directly from the local disk, without having to encrypt the whole file each time.[14]

Inserting content

Content insertion describes the steps it takes to initially insert content into a local node. It does not describe steps to distribute the content over the network or how to retrieve the content.

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 required.
3. Then it encrypts each block (1-key crypto) 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 required. All the 1k blocks obtained this way are than treated as under 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 $H(H(K))$.

Retrieving content

Invert "Inserting content".

8 Benchmarks

Encoding a 30 MB file for *GNet* takes about 5-10 minutes on a modern machine, the encoding time is dominated by the symmetric cipher used.

References

[1] Napster, `http://www.napster.com/`

[2] Gnutella, `http://gnutella.wego.com/`

[3] Mojo Nation, `http://www.mojonation.com/`

[4] OpenSSL, `http://www.openssl.org/`

[14]This is not possible in Freenet. If the provider of the content keeps a local copy, the file will be stored twice, encrypted for Freenet and unencrypted for normal use.

[5] Clarke, Sandberg, Wiley, Hong: *Freenet: A Distributed Anonymous Information Storage and Retrieval System.*, `http://www.freenetproject.org/`

[6] *RIPEMD160*