

An End-to-End Encrypted Anonymous File-sharing Network

Gabriel F. Stocco

May 5, 2008

A Thesis submitted to the Faculty
in partial fulfillment
of the requirements for the
BACHELOR OF ARTS

Accepted

Paul Shields, Thesis Advisor

Charlie Derr, Second Reader

Mike Bergman, Third Reader

Mary Marcy, Provost

Bard College at Simon's Rock

Great Barrington, MA

2008

Acknowledgments

Thank you to my thesis committee members, Paul Shields, Charlie Derr and Mike Bergman for your assistance with the thesis process. Your encouragement throughout the project and diligence with grammatical minutia is much appreciated.

Thanks also to Rose Li who took the time to proofread the semi-final draft and to Susan Mlodzianoski for your encouragement and editing.

Finally, thank you to my mother for your continued support.

Abstract

File-sharing networks are quickly growing in popularity and prominence.

Unfortunately, while advances have been made creating efficient and highly scalable networks, less attention has been paid to privacy and security issues which affected even the earliest file-sharing networks.

This thesis proposes and designs a new file-sharing network which protects the privacy of users in two ways: by encrypting all data sent over the network, and by anonymizing users so that they cannot be identified. I compare this network with existing file-sharing solutions.

Contents

1	Introduction	1
2	Existing Solutions	3
3	Design	8
4	Operation	11
5	Specification	14
6	Code	23
7	Conclusion	37
8	Bibliography	39

1 Introduction

The objective of this thesis is to create a working specification for an end-to-end encrypted anonymous file sharing network, and to implement that specification in a working prototype. An anonymous network allows for the anonymous distribution of files, preventing the identities of both sharer and receiver from being revealed to any party. An end-to-end encrypted network is secured from the monitoring of content as it is shared on the network. By combining the two, a network can provide for a technically secure method of data transfer, thus encouraging sharing. Furthermore, because of the design of the network, some attacks that can be especially effective against existing networks are fruitless against this network. This document lays out how the technical decisions were made which resulted in the final specification for GRRP, the protocol that I have developed. The full protocol specification is included in section 5.

In order to guarantee both end-to-end encryption and anonymity, several fundamental choices are made. To provide anonymity, peers are randomly assigned other peers by a central server. Any data connections are routed indirectly through multiple levels of peers before reaching their destination. In this way, the sender cannot identify the receiver, the receiver cannot identify the sender, and the peers transmitting data in the middle can't identify either sender or receiver. A similar idea is implemented in MUTE (see section 2), a file sharing network which provides anonymity but without end-to-end encryption. MUTE lacks encryption because previously it had not been considered possible to create a network which provides for both anonymity and reliable strong encryption.¹ One way to address the problem, though, is to have the server, assuming it is trustworthy, function as a key repository for the clients using public key encryption. Working from that premise, the first draft of the protocol specification was written to use only public key cryptography.

¹Problems arise when key exchange must be done in the clear and no trust can be established. A man-in-the-middle can scan packets passing by it for encryption keys, substituting its own when appropriate. By doing so, the man-in-the-middle is able to decrypt entire data stream.

As testing progressed, however, it became clear that public key cryptography (which works on the principle that large prime numbers are incredibly hard to factor) would be far too slow for use on all data packets. Nevertheless, public key cryptography could still serve as an excellent layer of protection for exchanging a symmetric encryption cipher. Symmetric encryption ciphers encrypt and decrypt at a much greater speed and thus are better suited for exchange of large amounts of information when the key can be exchanged in a secure manner. Thus, in the second draft of the protocol, all connections, peer-to-peer or peer-to-server, are first used to exchange public keys. That done, a symmetric cipher is generated, encrypted using public key encryption, and transferred. Further communication between the end points is encrypted using the symmetric cipher.

The final protocol design still has several limitations. There are a minimum number of peers required, around 5, for it to function at all. However, without a significantly larger pool of peers, around 50, it would be quite difficult to create any meaningful anonymity.² When a network is small enough, it might be possible to make reasonable inferences about the content that a peer is receiving by examining all of the data that is being transferred. Secondly, there must be a level of implied trust in the server to provide for a truly random allotment of peers and the exchange of encryption keys. Like a Bittorrent tracker (section 2), the server has in memory a complete list of all the peers connected to the network. Unlike a Bittorrent tracker, however, the GRRP server does not know anything about which files the peers on its network are transferring. A discussion of possible attacks against the network, along with some possible fixes, is provided in section 7.

²I expect GRRP to scale to the thousands of users, at which point it will provide excellent anonymity, with the number of users only limited by the processing power of the server. However, GRRP has not been tested at such a scale.

2 Existing Solutions

It all started with Napster

Napster was the first popular peer-to-peer filesharing network. Created in 1999, Napster utilized a network architecture that had a server which handled search requests, with clients informing the server of the details of what files they were sharing. The files were then transferred directly between clients. Without the central server, the Napster network could not function. The US Judicial System ruled Napster illegal in 2001 and the server was shut down.³ Even so, the ideas Napster brought to the table have spurred years of technological development. Following the death of Napster came Fasttrack, Guntella, MUTE, Bittorrent and Tor, among others, each with its own goals, purposes and weaknesses.

Bittorrent is King

Created in 2001, Bittorrent is a peer-to-peer file sharing network whose structure contains a number of peers and a server, called a 'tracker'. Users who want to download a file download a .torrent file which contains meta-data about that file, including hashes to verify integrity, and the IP address of the tracker to connect to. The tracker keeps track of, and publicly shares, the IP addresses of every connected client, the files they are sharing or downloading, and the pieces of the file which they have obtained. Bittorrent uses a 'tit-for-tat' algorithm to determine which users will get priority in situations with limited upload bandwidth, prioritizing users with the best upload speeds.⁴

Peer-to-peer traffic is estimated to make up between 48% and 80% of all traffic on the internet, with Bittorrent composing roughly two thirds of that.⁵ Most of this traffic is likely composed of sharing copyrighted materials. However, Bittorrent clients are also used for distributing open source software, such as Linux distributions, greatly reducing the dis-

³<http://news.bbc.co.uk/2/hi/business/852283.stm>

⁴Bittorrent's tit-for-tat algorithm works by throttling upload speeds to clients who are themselves uploading little data. By doing so, the creators of Bittorrent hoped to avoid some problems which plagued earlier filesharing networks, including a lack of peers willing to upload data.

⁵http://www.ipoque.com/news_&_events/internet_studies/internet_study_2007

tribution costs. Bittorrent is also increasingly being used to legally distribute commercial software, video, music and other custom applications.

For example, World of Warcraft, the most popular online role-playing game, with well over 5 million active users, uses a Bittorrent based client to distribute patches. Bittorrent can also be used to help automate patching systems with huge bandwidth considerations. INHOLLAND University found that by using Bittorrent to distribute patches they could push patches to 6,500 computers in 4 hours with only 4 servers. Using conventional methods the process had previously taken them four days.⁶

Bittorrent revolutionized file-sharing, significantly increasing the efficiency of file transfers by increasing the granularity and quality of hash files. Most importantly, Bittorrent greatly increased transfer speeds by allowing clients to begin uploading data once they have obtained even a single chunk of the file.

Bittorrent also separated the meta-data from the network by the use of .torrent files, Bittorrent removed the costly and inefficient searching from the network, allowing clients to be able to connect to multiple networks without the added overhead of routing search traffic for each. The trade-off in this approach is that the 'tracker' must be entirely trustworthy, since it can track what every user connected to it is doing. By extension, every user connected to the tracker must be trustworthy, because they can obtain a copy of all the information tracked by the tracker as a consequence of the Bittorrent network design.

FastTrack and Gnutella

FastTrack and Gnutella, first launched in 2000, are two networks which are based on a network structure consisting of normal peers, and 'superpeers' or 'supernodes'. Peers on the FastTrack and Gnutella networks simply share and download data. Superpeers on

⁶<http://arstechnica.com/news.ars/post/20080309-dropping-22tb-of-patches-on-6500-pcs-in-4-hours-bittorrentdropping-22tb-of-patches-on-6500-pcs-in-4-hours-bittorrent.html>

the other hand, create file indices to help speed searching, are crucial in the bootstrapping process, and route data when peers are behind a firewall or NAT, which prevents the peer from opening an incoming port.⁷

At the peak of its popularity, the FastTrack network boasted 2.4 million users while Bit-torrent served an estimated 10 million in March 2006.⁸

The primary difference which separates FastTrack and Gnutella from the original Napster network is their further decentralized nature. Instead of one central server which assists searches and indexes files, the supernodes take over that role. By moving the burden of indexing and searching into the network, FastTrack and Gnutella are able to scale better than Napster. Unfortunately, their search reliability is low because results are based on outdated indices stored on the supernodes and transfer speeds are slowed when routed through the supernodes.

MUTE

MUTE is an anonymous file sharing network. Created in December 2003, MUTE utilizes a network structure that is similar to the FastTrack/Gnutella networks, but instead of direct connections between peers, MUTE uses extra peers as intermediaries to ferry information from end node to end node in order to provide anonymity to the end peers. While the anonymity provided by MUTE is a significant security improvement over the other networks mentioned above, including Bittorrent, MUTE does not employ any end-to-end encryption. Thus, any peer engaged in the transfer, even one shuttling information between end peers, is able to read the packets in full. This reintroduces, in theory, the possibility of tracking down the sender and recipient of the data by analysing the content of the transfer.

⁷Bootstrapping is the process by which a client connects to the network. Network Address Translation (NAT) is a router based technology which allows multiple clients to share one external IP address. The tradeoff is greatly increased difficulty in directly contacting to clients behind a NAT.

⁸<http://en.wikipedia.org/wiki/FastTrack>, http://www.businessweek.com/technology/content/may2006/tc20060508_693082.htm

Tor - The Onion Router

Tor is a peer-to-peer application more than it is a file-sharing program. Nonetheless, it can serve as one solution to the problem of secure file sharing. Tor uses a distributed network of peers that shuttles around encrypted copies of the data, obfuscating identities and physical locations. Tor does this by having self-selected peers, which act as exit nodes that can decrypt the data, send it over the internet as if the packet had originated from the exit node. All communications to the greater internet can be sent through Tor, effectively anonymizing access with a double-blind approach.⁹

Unfortunately, while the data is hidden from peers in the network, it is revealed in full to the exit node which sends it on to the internet at large. This exposes the user's data to the exit node, though without the user's identity. This can still pose a large security risk unless all network traffic sent through Tor is not personally identifiable.

Tor has also run into serious problems due to a dearth of available bandwidth from exit nodes. Exit nodes must be run on a volunteer basis, costing the user bandwidth and potentially leaving them open to lawsuit.¹⁰ Finally, Tor is a network through which you must tunnel other traffic. By itself, Tor does not provide any network functionality, it merely helps to protect the user of an unsecured protocol against many common surveillance tactics. Unfortunately, the bandwidth limitations that Tor faces greatly restricts the range of applications, and number of users, able to effectively run on the network.

HTTP

Wikileaks, a website dedicated to developing an “uncensorable system for untraceable mass document leaking and public analysis”,¹¹ is shaping up to be a viable replacement for many uses of file sharing networks. But Wikileaks is vulnerable to attack on its DNS

⁹Tor hides the identity of the user from the network they access using Tor, as well as hiding it from the exit node which sends that data over the network.

¹⁰http://www.cnet.com/8301-13739_1-9779225-46.html

¹¹<http://wikileaks.org/>

entries, as is the case with any website. This vulnerability was exposed when a court in California ordered the DNS entries for Wikileaks.org to be taken down, leaving the Wikileaks site up, but largely inaccessible.¹²

¹²<http://news.bbc.co.uk/2/hi/technology/7250916.stm>

3 Design

Encryption

Choosing the right encryption algorithm and the appropriate number of bits to use with the algorithm was a difficult process. A careful balance must be struck between security and feasibility of implementation. The first revision of the specification used public key encryption exclusively so that the encryption keys could be shared in the open even without a safe tunnel to exchange keys. After a review of the available public key encryption algorithms, I settled on RSA, which is one of the most widely used and trusted public key encryption schemes.

Upon testing the speed of the encryption algorithm it became clear that public key encryption was unsuitable for such an application. Its benefits however could be used to exchange encryption keys for a much faster algorithm. NIST¹³ ran a competition in an attempt to create an encryption standard that would be secure enough to store classified government documents. It was called the AES.¹⁴

Five finalists were chosen for the AES competition. Four of the five were good enough to be considered for the purposes required by the GRRP network: Rijndael, which became the AES standard; RC6, a successor to the popular RC4 encryption that is currently used by some Bittorrent clients to obfuscate headers; Twofish, a cipher written by Blowfish creator Bruce Schneier; and Serpent, a more secure, but slower version of Rijndael. The 5th AES candidate, Mars, was not considered, as it scored exceptionally low on all the tests run by NIST.

Twofish was the best for a number of reasons. First, one of the primary deciding factors for choosing Rijndael over Twofish for the AES competition was the larger memory

¹³National Institute of Standards and Technology, a non-regulatory agency of the United States Department of Commerce

¹⁴Advanced Encryption Standard

footprint of Twofish. The larger memory footprint was a concern for the competition, as the standard chosen had to be easily implemented in hardware smart cards.¹⁵ This is not a factor in the choice for GRRP.

A common metric used to evaluate encryption algorithms is the number of 'rounds' that they implement. A round is a iteration through the algorithm. The "safety factor" of an algorithm is a good indicator of how likely it is to be cracked, and correlates to the number of rounds. The safety factor is defined as the maximum number of rounds divided by the number of rounds that have been broken. Using this analysis, a safety factor of 1 indicates a completely cracked algorithm. Twofish has a safety factor of 2.67, while Rjindael is significantly weaker with a safety factor of only 1.56. Serpent has a safety factor of 3.56, but it is much slower than either Twofish or Rjindael.

Anonymity

Creating an anonymous network requires hiding the identities of all of the peers on the network such that each peer cannot match traffic up with its source or decipher its content. While both anonymous networks and end-to-end encrypted networks exist, some considered combining the two features into one network to be an intractable problem. Further research on the topic indicated that adding encryption to a MUTE type network had already been considered, but the creator of MUTE himself decided that it was not a problem which could be solved.¹⁶

GRRP combines the network behavior of MUTE, which creates anonymity using middle peers to establish data connections and search connections with the external file-based hashes and tracker of Bittorrent. All the while, encryption is available at every link, assisted by the server. The server itself is authenticated by distributing its public encryption key through a third party. Once the secure link from the client to the server is established,

¹⁵Smart Cards are hardware devices with limited memory and processing power.

¹⁶<http://mute-net.sourceforge.net/personInTheMiddle.shtml>

the server can serve as the repository for keys. This allows for the anonymous and reliable distribution of encryption keys to clients. It avoids the potential man-in-the-middle attacks, which had discouraged the earlier creation of similar networks. Thus, any two peers exchanging data have their identities concealed through a number of levels of rerouting through other random peers. Meanwhile, the encryption on the payload protects the information being transferred from being tracked or analyzed. This system also protects the user against perfect knowledge attacks. Even if it were conceivable that an adversary had a perfect set of data of all the connections on the network, it would only be possible to conclude that two peers transmitted some data, but it would be impossible for the adversary to determine exactly what the information was.

4 Operation

Client Initialization

When a client starts up it immediately begins to generate a store of encryption keys. First a set of RSA keys is generated, followed by a Twofish cipher. The number of keys generated will depend upon the settings that the user has specified in the preference file; the default is 5.

When a client is started up for the first time it will wait for further instructions from the user. There is no default server, so the client will not have anything to connect to unless it has previously been used. In order to connect to a server to share data or to obtain new data, the user must obtain a hash file for the file they wish to download. The hash files serve a similar purpose to a .torrent file: they contain the information necessary to verify portions of the file as it is shared, as well as the address of a server to connect to.

Connecting to a Server

When a client connects to a server, they exchange RSA keys, which are used immediately to encrypt and transmit a Twofish cipher generated by the client. The cipher is then used to encrypt all further communications between the client and server. After the Twofish encryption is established, the client will request from the server a random list of IPs and ports. This is a list of mini servers that other peers are running on their clients. The client will connect directly to the IPs specified in order to search the network and transfer data. These connections are encrypted in much the same way that the connection with the server is established.

Searching the Network

In order to search the network, the client must connect to a server and request a group of search peers (initialization). Having done that, whenever a new hash file is loaded the

client will send search requests to all of the peers to which it is directly connected.

If a client receiving a search request has the file in question, they will send back a message, along the route that the search request got to them, indicating that they have the file. They will also pass the search request on so that the searching peer can get the most peers possible available for data transfer.

Once informed that a peer has data available for transfer the client will choose a random chunk of the file to request from each of the peers which indicated that they have the file available (chosen from the chunks that that peer has available for sharing). As the chunks begin to transfer, the client will also indicate to searchers that it has pieces of the file available.

Limitations

In order to function, the client must be able to form direct connections to other peers. Unfortunately, a great many potential clients will be behind NATs or firewalls. NAT traversal is certainly possible but it is not addressed in this version of the application. Hopefully, with the increasing adoption of IPv6 this will no longer be an issue.¹⁷

Server Initialization

Once the server starts up, it begins to generate encryption keys which it uses as clients connect. It generates a unique, new encryption key for each client and replenishes the pool of encryption keys as it goes. As with the client, the number of keys kept in memory for use is determined by the preference file, but unlike the client, the server will keep many more keys available with the goal of always having at least 50 available for new clients.

No state is saved between server shutdowns, so the server starts up with no knowledge

¹⁷IPv6 does not implement NAT, instead assigning individual global IPs to every computer

of any of the clients who were previously connected to it. All client connections must be completely reinitialized from the public key stage. Connections established before the server went down can continue to be used, but no new connections can be established, and no new peers can join the network.

Functions

The server provides a very minimal set of functions, with no control over what materials are shared over the network, only over which clients are allowed to connect to the network. Effectively, the only role of the server is to provide each client with a list of peers with which it can connect. In addition, the server provides help in the key exchange portion of client communications, in order to help prevent man-in-the-middle attacks. For example, key substitution attacks, a primary concern for the developer of MUTE, are avoided using this method.

Luckily, such attacks are costly to mount. Because of their rarity it is unlikely that they will be a real threat to a network. Nonetheless, users who anticipate concentrated surveillance, such as live stream filtering of their internet connection, should consider using an additional layer of protection before connecting to the network. The author does not know of any applications that are immune to such an attack, however.

5 Specification

Preface

Description

GRRP is an application layer protocol for anonymous, encrypted file sharing over TCP/IP. The protocol has two broad categories of communications: peer-to-server, and peer-to-peer. Encryption is provided by pseudorandomly generated keypairs, while anonymity is achieved through the use of an arbitrary number of intermediate peers for every data connection. The protocol runs on TCP/IP.

Terms

Terms which may have ambiguous meanings in use are strictly defined below in alphabetical order.

Table 1: Terms

Term	Meaning
action	A specific request/response pair. Specific actions are defined in sections 3 and 4.
client	An application which provides the services described in section 4.
command	See action.
grrId	A unique intra-network identifier.
message	The basic unit of communication. Contains one or more actions, and the associated datum. Fully defined in section 2.
peer	See client.
search peer	Each client is directly connected to a number of other clients to create mesh used for searching. A search peer is one of these clients. Note that this term only has meaning in a relative context - all clients act as search peers.
server	An application which provides the services described in section 3.

Communication is defined in terms of actions which are contained in messages which may span multiple network packets. A message contains one action.

Overall Operation

GRRP is a file-sharing protocol that provides both anonymity and strong end-to-end encryption through the use of a central server acting as a key repository.

A client bootstraps onto the network by registering their IP address in a request for a

grrId. This bootstrap action is called registering. If no errors occur, and the IP is not banned from the network, the server will return a grrId to the client.

Once a client has bootstrapped onto the network, they request search peers from the server. This request action is called populating.

To search, a client will send a search request to their immediate search peers. Each search peer will forward the search request on to each of their search peers. A client receiving a search action for a file that they are sharing will validate that the search message is genuine via signature validation and if the message is genuine, the client will send an encrypted response back along the path on which they received the request. The client will also send a message with a request action to the server.

Encryption and File Validation

Handshaking is done using RSA public key encryption with a 2,048 bit key. Once RSA keys have been exchanged, a 256 bit Twofish cipher is generated, encrypted using the RSA keys and sent. This process is explained in full in the relevant section - Client or Server - later in the document.

Files are added for download/upload by the use of binary .grrp files. The .grrp file contains a list of all the files included in the download, in addition to the file hashes for each of those files. Lastly, .grrp files contain the default address for the server that they were initially distributed on, and that server's public key.

Example .grrp file and associated data file:

testfile.grrp:

```
208.113.212.122      public encryption key      testfile.txt
ed4edad3/42b53eb6582deec2dd290db7bc52bce15fb29fc2ed2
ef616358c8e2aff2a494562da732bb1dc3bbf9a433af01cbca60411cf4b812649
f282d55f550bc28b2a9a
```

testfile.txt:

```
Hi, I am a test file.
```

Each file in a .grrp file is hashed in 1MB chunks using the SHA512 hashing algorithm. Each 1MB chunk has a corresponding 64 byte hash.

Messages

Messages are the basic communication unit in GRRP. A message must consist of a public header, a private header, a body and a closing statement. Messages can span more than one network packet, depending on MTU size and the size of the message being transmitted. The public header is unencrypted so that intramash routing can take place. The private header, body and closing statement are strongly encrypted in all messages other than search. For a search action message, the private header, body and closing statement are unencrypted.

1. Public Header: The public header contains the following information: a grrId associated with the peer to which the message is addressed.
2. Private Header: The private header contains the following information: the grrId of the client which sent the message. (If it is not a message to or from the server or a search message).
3. Body: The body contains the action type byte and the data to be processed.

Message Types The three messages types are peer-to-server, peer-to-peer search, and peer-to-peer normal. They are described in more detail in sections 3-4.

Peer-to-Server Communication

Peer Actions

Peers initiate all communications, but the server can initiate communications with

other peers in response to a peer's request. There are 5 message types for peer-to-server communication. All message types must be supported as defined. The following list expresses actions from the server's perspective.

Inputs and outputs are specified as "Variable Name (Variable Type)". Optional variables are denoted with an asterisk (*), an array of arbitrary size is denoted by square brackets ([]). For this section, "Input" means that data is passed from the client to the server, "Output" shall means data is passed from the server to the client.

1. Initialize

Action Number: 0
Input: Public Encryption Key (bytes)
Response Number: 1
Output: Public Encryption Key (bytes)
Output Encryption: RSA

Description: The initialize command sends the public encryption key that the client wishes to use to the server. The server sends the public encryption key which it will use for the connection to the client in response.

2. Cipher Exchange

Action Number: 1
Input: Twofish sBox (bytes), Twofish sKey (bytes)
Input Encryption: RSA
Response Number: 2
Output: Port Number (int)
Output Encryption: Twofish

The Cipher Exchange command is used after the initialize command. The client sends

the server the Twofish cipher it wishes to use to create a connection. The server responds with a random port number on which the client opens up a socket to listen for other peers and to participate in intra-network routing and searching.

3. Populate

Action Number: 2

Input: None.

Response Number: 3

Output: Array of IP Address (4 bytes), Port (4 bytes)

The populate command allows a client to request a refresh of its directly-connected searching peers. The server will return a list of random peers selected from the network. This list may be empty if there are no other clients connected to the network. No error is given in this case.

4. Connect

Input: grrId (int)

Output: IP Address (String)

The connect command allows a peer to request a connection with another peer on the network using a grrId of that peer. The connection will then be established by the server sending commands to each of the peers involved in the connection with either one or two IPs to connect to. Middle-man peers will receive two IPs, and the communicating peers receive will one.

Server Actions

For some of the peer initiated commands, the server must contact another peer to

complete the command. For these commands, the Input comes from the server and the output from the second peer.

1. Middle

Input: IP Address [] (String)

Output: Confirmation (boolean)

The middle command instructs a peer to act as the middle-man between the two IP addresses given as arguments. The end peers will signal to the middle-man peer when they are finished with the connection. A middle-man may serve as the middle-man between two end peers, two middlemen, or one end peer and one middle-man. It is not indicated, by design, whether a middle-man knows what the nature of the nodes it is connected to is.

2. Connect

Input: IP Address (String), grrId (String)

Output: Confirmation (boolean)

Description: The connect command from the server tells a client that a connection is being established through the given IP address to the grrId that the client requested a connection to.

Peer - Peer Communication

Peer-to-peer communication is generally encrypted and anonymized using middle-man transfer tunnels. There are two kinds of peer-to-peer traffic: search and passthru. Search messages are unencrypted and evaluated by every peer receiving them, while passthru messages are encrypted, passed thru along anonymized connections, and evaluated only by the end peers.

Searching

Each peer has a number of directly connected search peers which it uses as a search mesh. A peer can either use these directly connected peers to search - a potential anonymity risk - or it can request an anonymized connection tunnel from the server to use another peer in the network as a proxy for the search.

1. Search

Contents: Hash (String), grrId (String)

Encrypted: Signed.

Anonymous: Optional.

The search action is multicast to every search peer to which the searching peer is directly connected. Each client receiving a search packet will compare the hash to the files they are sharing. If the hash matches, the client will send a connect request to the server with the grrId of the requesting peer. A confirm message is then sent, as an encrypted message, inside that anonymized connection.

Passthru Communications

All passthru messages consist of an encrypted payload: the grrId of the recipient and the grrId of the sender. Passthru messages are sent through anonymized tunnels constructed by the server.

1. Confirm

Contents: Hash (String), grrId (String)

Encrypted: Yes.

Anonymous: Yes.

A confirm message is sent by a peer who has received a search request for a specific file hash to the peer who requested the file with that hash.

2. Data

Contents: Payload (binary), grrId (String)

Encrypted: Yes.

Anonymous: Yes.

The data message is the sole method for transporting real data over the network. Data messages are sent through anonymized connections as passthru messages.

Transmission Speed and Timeouts

To ensure network usability, reasonable timeouts must be implemented. This is especially difficult in the GRRP network, where each anonymized connection link has an arbitrary number of middle-man peers. Moreover, to avoid security issues no peer should have tangible data relating to the number of middle-man peers. This section discusses the timeout considerations that have been added to GRRP.

Anonymized Links

To ensure data integrity, while simultaneously maintaining security in anonymized links, timeouts are tracked between directly connected-nodes. Data is preserved at each peer until its reception at the next node is confirmed. Thus, if peer A is transmitting data to peer B through peers C, D and E, a timeout could occur between peers A and C or between C and D, but not between A and D.

When a timeout occurs, the sending peer will attempt to retransmit the timed-out message and will increment the timeouts counter. When 3 consecutive timeouts occur or 5 messages timeout out of 10, the peer sending messages will wait a short amount of time and request a new connection from the server.

When a new connection is requested, the server adds a temporary row in a tracking database, which expires in 24 hours. This contains the IPs of all of the peers involved in the old connection to deter spoilers (false reports), which are a burden on the network, and 'bad' clients (intentional packet droppers).

Since both of these spoiling behaviors appear, upon logging, to be the the same action, it is impossible to tell which is occurring. Depending on the number of times a peer shows up in that database, it may be temporarily or permanently banned by the server administrator to keep poorly performing or corrupt peers out.

Client - Server Timeouts

The server purges all data, other than timeout information, relating to a client's IP and associated grIds. It also breaks down that client's connections, rerouting when requested.

If a client's 5 consecutive messages to the server have timed out, the client will assume that the server is down, disconnect, and attempt to reconnect. It may be the case that the client was banned silently, in which case the error message should display upon reconnecting to the server.

6 Code

The Twofish algorithm is implemented in `grrpFish.java`. The GNU Crypto Library is leveraged to provide the encryption.

```
package grrp.common;

import java.io.*;
import java.util.Iterator;

import java.security.SecureRandom;
import java.security.InvalidKeyException;

import gnu.crypto.cipher.*;

public class grrpFish{

    final int BLOCKSIZE=16;
    final int INTSIZE=4;//Size of int in bytes

    private Object sessionKey;
    private byte[] key;
    Twofish cryptor;
    SecureRandom rand;

    public grrpFish(byte[] b){
        cryptor = new Twofish();
        key=b;
        genKey(key);
    }

    public grrpFish(){
        rand=new SecureRandom();
        key=new byte[32];
        rand.nextBytes(key);
        sessionKey=null;
        genKey(key);
    }

    private void genKey(byte[] b){
        cryptor=new Twofish();

        try{
            sessionKey = cryptor.makeKey(key, BLOCKSIZE);
        }
        catch (InvalidKeyException e){
            e.printStackTrace();
        }
    }

    public Object makeMeAKey(byte[] b){
        Object sessionKey;
        int[] sBox = new int[1024];
        int[] sKey = new int[40];
        for(int i=0;i<b.length;i+=4){
            if(i<sBox.length*4){
                byte[] temp = new byte[4];
                temp[0]=b[i];
                temp[1]=b[i+1];
                temp[2]=b[i+2];
                temp[3]=b[i+3];
                sBox[i/4]=grrpLib.byteArrayToInt(temp);
            }
            else{
                byte[] temp = new byte[4];
                int j=i-(sBox.length*4);
                temp[0]=b[j];
                temp[1]=b[j+1];
                temp[2]=b[j+2];
                temp[3]=b[j+3];
                sKey[j/4]=grrpLib.byteArrayToInt(temp);
            }
        }
        Object[] ret = new Object[] {sBox,sKey};
        return ret;
    }

    public Object makeMeAKey(String input){
        Object sessionKey;
        int[] sBox;
        int[] sKey;
        String[] splitted = input.split("#");
        String[] sbox = (splitted[0]).split("@");
        String[] skey = (splitted[1]).split("@");
        sBox = new int[sbox.length];
        sKey = new int[skey.length];
        System.out.println(sbox.length);
        for (int i=0;i<sbox.length;i++){
            byte[] temp=sbox[i].getBytes();
            sBox[i]=grrpLib.byteArrayToInt(temp);
            System.out.println(i);
        }
        for (int i=0;i<skey.length;i++){
            byte[] temp=skey[i].getBytes();
            sKey[i]=grrpLib.byteArrayToInt(temp);
        }
        sessionKey = new Object[] {sBox,sKey};
        return sessionKey;
    }

    public byte[] getKey(){
        return key;
    }
}
```

```

public byte[] getSessionKey() {
    Object[] sk = (Object[]) sessionKey; // extract S-box and session key
    int[] sBox = (int[]) sk[0];
    int[] sKey = (int[]) sk[1];
    byte[] output = new byte[sBox.length*INTSIZE+sKey.length*INTSIZE];
    for (int i=0; i<sBox.length; i++) {
        byte[] temp = grpLib.intToByteArray(sBox[i]);
        for (int j=i*INTSIZE; j<((i+1)*INTSIZE); j++) {
            output[j] = temp[j%INTSIZE];
        }
    }
    for (int i=0; i<sKey.length; i++) {
        byte[] temp = grpLib.intToByteArray(sKey[i]);
        for (int j=(sBox.length+i)*INTSIZE; j<((sBox.length+i+1)*INTSIZE); j++) {
            output[j] = temp[j%INTSIZE];
        }
    }
    return output;
}

public byte[] encrypt(byte[] arr) {
    int arr3Size = (arr.length%16==0)?(arr.length):(arr.length+16-(arr.length%16)); //Figure out what size the array has to be
    byte[] arr3 = new byte[arr3Size]; //Make the full size array
    for (int i=0; i<arr.length; i++) //Put the data into the full size byte array
        arr3[i] = arr[i];
    for (int i=arr.length; i<arr3.length; i++) //Pad the data with 0s
        arr3[i] = 0;
    byte[] arr2 = new byte[arr3.length]; //Empty array for encrypted data
    for (int i=0; i<arr3.length; i+=16) {
        cryptor.encrypt(arr3, i, arr2, i, sessionKey, BLOCKSIZE);
    }
    return arr2;
}

public byte[] decrypt(byte[] arr) {
    int arr3Size = (arr.length%16==0)?(arr.length):(arr.length+16-(arr.length%16)); //Figure out what size the array has to be
    byte[] arr3 = new byte[arr3Size]; //Make the full size array
    for (int i=0; i<arr.length; i++) //Put the data into the full size byte array
        arr3[i] = arr[i];
    for (int i=arr.length; i<arr3.length; i++) //Pad the data with 0s
        arr3[i] = 0;
    byte[] arr2 = new byte[arr3.length]; //Empty array for decrypted data
    for (int i=0; i<arr3.length; i+=16) {
        cryptor.decrypt(arr3, i, arr2, i, sessionKey, BLOCKSIZE);
    }
    return arr2;
}

public static void main (String args[]) {
    new grpFish();
}
}

```

RSA Public Key encryption is implemented in grpCrypt.java. The GNU Crypto Library is leveraged to provide the encryption.

```

package grp.common;

import java.math.BigInteger;
import java.security.*;
import java.security.interfaces.*;
import java.util.BitSet;
import java.security.KeyPairGenerator;
import java.util.logging.Level;
import java.util.logging.Logger;

public class grpCrypt {
    private final static BigInteger one = new BigInteger("1");
    private final static SecureRandom random = new SecureRandom();

    public BigInteger privateKey;
    public BigInteger publicKey;
    public BigInteger modulus;

    private int bits=2048;
    private int msgSize=bits/8; //Absolute limit is bits/8. Any higher breaks RSA (msg > modulus).

    private byte[] bitVals = new byte[] { (byte)1, (byte)2, (byte)4, (byte)8, (byte)16, (byte)32, (byte)64, (byte)128 };

    public grpCrypt() {
        try {
            grpLib.dbLog("Generating encryption keys.");
            SecureRandom ran = new SecureRandom();
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            keyGen.initialize(bits, ran);
            KeyPair pair = keyGen.generateKeyPair();
            RSAPrivateKey priv = (RSAPrivateKey) pair.getPrivate();
            RSAPublicKey pub = (RSAPublicKey) pair.getPublic();
            modulus = priv.getModulus();
            privateKey = priv.getPrivateExponent();
            publicKey = pub.getPublicExponent();
            grpLib.dbLog("Done.");
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(grpCrypt.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

```

public grpCrypt(BigInteger publicKey, BigInteger modulus) {
    this.publicKey = publicKey;
    this.modulus = modulus;
}

public grpCrypt(byte[] modulus) {
    this.publicKey = new BigInteger("65537");
    this.modulus = new BigInteger(modulus);
}

BigInteger encrypt(BigInteger message) {
    return message.modPow(publicKey, modulus);
}

BigInteger decrypt(BigInteger encrypted) {
    return encrypted.modPow(privateKey, modulus);
}

public BigInteger getPublicKey() {
    return publicKey;
}

public BigInteger getModulus() {
    return modulus;
}

public String outputPubKey() {
    return new String(publicKey.toString()+">" + modulus.toString());
}

public byte[] getBytes() {
    return modulus.toByteArray();
}

/*
Encrypted Message Format:
xyz
x - 4 bytes - chunk count
y - n bytes - 1 bit per chunk. If the bit is set to 1, there are 257 bytes in that chunk. If 0, 256 bytes.
z - m bytes - chunks
*/
public byte[] encryptBytes(byte[] pText) {
    /*
    Each bit refers to a chunk.
    If set to 1, the chunk is 257 bytes.
    If set to 0, the chunk is 256 bytes.
    This is needed because the encrypt method produces a 257 byte cipher text ~20% of the time.
    */
    byte[] chunkBits = new byte[0];
    byte[] tempChunkBit = new byte[1];
    int chunkSet = 0;
    byte[] cText = new byte[0]; // The cipher text
    int lastI = 0; // Keeps track of the number of chunks in the message.
    int size = 0; // The size of this message
    for (int i = 0; i < pText.length; i += size) { // i keeps track of the position in the pText
        if ((pText.length - i) == msgSize + 1) {
            size = msgSize / 2;
        }
        else if ((pText.length - i) > msgSize) {
            size = msgSize; // If there is enough data left for two messages - set size = max msgSize
        }
        else {
            size = pText.length - i; // Otherwise, just the number of bytes left
        }

        byte[] temp = new byte[size];
        for (int j = 0; j < size; j++) { // Get the subsection of the input byte array.
            temp[j] = pText[i + j];
        }
        BigInteger temp2 = encrypt(new BigInteger(temp)); // Encrypt it
        byte[] temp3 = temp2.toByteArray(); // Store it in a byte array

        if (temp3.length == msgSize + 1) {
            chunkSet = 1;
        }
        else {
            chunkSet = 0;
        }
        if (lastI == 0) {
            tempChunkBit[0] = chunkSet * bitVals[lastI % 8];
        }
        else if (lastI % 8 == 0) {
            chunkBits = grpLib.concatBytes(chunkBits, tempChunkBit);
            tempChunkBit[0] = 0;
            tempChunkBit[0] = chunkSet * bitVals[lastI % 8];
        }
        else {
            tempChunkBit[0] = chunkSet * bitVals[lastI % 8];
        }

        cText = grpLib.concatBytes(cText, temp3); // Concatenate the new chunk onto the old byte array
        lastI++; // Increment the chunks number
    }
    chunkBits = grpLib.concatBytes(chunkBits, tempChunkBit);
    byte[] count = grpLib.intToByteArray(lastI);
    count = grpLib.concatBytes(count, chunkBits);
    cText = grpLib.concatBytes(count, cText);
    return cText;
}

public byte[] decryptBytes(byte[] b) {
    byte[] c = new byte[0];
    byte[] chunks = new byte[4];
    for (int i = 0; i < 4; i++) {

```

```

        chunks[i]=b[i];
    }
    int chunks2=grrpLib.byteArrayToInt(chunks);
    byte[] bits = new byte[(chunks2/8)+((chunks2%8==0)?0:1)];
    for(int i =4;i<4+bits.length;i++){
        bits[i-4]=b[i];
    }
    boolean[] bitSet = new boolean[chunks2];
    int bitItr=0;
    for(int i=0;i<chunks2;i++){
        if (i%8==0){
            if (i!=0){
                bitItr++;
            }
            if (bits[bitItr]<(byte)0){
                bits[bitItr]-=(byte)128;//Get rid of the integer overflow
                bitSet[i+7]=true;
            }
            if ((bits[bitItr]/bitVals[i%8])%2!=0){
                bitSet[i]=true;
            }
        }
        else if ((bits[bitItr]/bitVals[i%8])%2!=0){//Exploit integer division to set bits as appropriate
            bitSet[i]=true;
        }
    }
    byte[] data = new byte[b.length - bits.length - chunks.length];
    for(int i=0;i<data.length;i++){
        data[i]=b[(bits.length+chunks.length)+i];
    }

    int j,jholder=0;
    for(int i=0;i<chunks2;i++){
        int size=msgSize;
        if (bitSet[i]==true){
            size=msgSize+1;
        }
        byte[] temp=new byte[size];
        for(j=jholder;j<jholder+size && j<data.length;j++){
            temp[j-jholder]=data[j];
        }

        BigInteger tmp = new BigInteger(temp);
        BigInteger temp2 = decrypt(tmp);
        String temp3 = (new String(temp2.toByteArray()));

        c=grrpLib.concatBytes(c,temp2.toByteArray());
        jholder+=size;
    }
    return c;
}

}

```

The client is designed with most of its functionality included in grrpSocket.java. This file functions as the socket which the clients use to connect to both the server and other peers.

```

package grrp.client;

import java.net.*;
import java.io.*;
import java.math.BigInteger;
import java.util.ArrayList;
import grrp.common.*;

public class grrpSocket {

    private Socket socket;
    private BufferedInputStream in;
    private BufferedOutputStream out;

    private ServerSocket servSocket;

    private grrpCrypt keys;
    public grrpFish cipher;
    grrpCrypt serverKeys;

    private ArrayList<neighborSocket> neighbors=new ArrayList<neighborSocket>();

    private ArrayList<gFile> files = new ArrayList<gFile>();
    int port;

    grrp.client.clientThread outputThread;

    neighborThread nThread;
    int serverType;

    public grrpSocket(InetAddress addr, int port, int serverType) throws Exception{
        socket = new Socket(addr,port);//Connect to the server
        in = new BufferedInputStream(socket.getInputStream());//Buffer the output and input streams
        out = new BufferedOutputStream(socket.getOutputStream());
        keys = new grrpCrypt();//Create a set of RSA keys.
        byte temp=grrpLib.CINIT;

        this.serverType=serverType;

        out.write(temp);
        out.write(keys.getBytes());
        out.flush();
        new listenThread().start();
    }

    public grrpSocket(Socket socket, int serverType) throws Exception{
        in = new BufferedInputStream(socket.getInputStream());//Buffer the output and input streams

```

```

        out = new BufferedOutputStream(socket.getOutputStream());
        keys = new grgpCrypt();//Create a set of RSA keys.
        byte temp=grgpLib.CINIT;

        this.serverType=serverType;

        out.write(temp);
        out.write(keys.getBytes());
        out.flush();
        new listenThread().start();
    }

    public grgpSocket(InetAddress addr, int port) throws Exception{
        socket = new Socket(addr,port);//Connect to the server
        in = new BufferedInputStream(socket.getInputStream());//Buffer the output and input streams
        out = new BufferedOutputStream(socket.getOutputStream());
        keys = new grgpCrypt();//Create a set of RSA keys.
        byte temp=grgpLib.CINIT;

        this.serverType=0;

        out.write(temp);
        out.write(keys.getBytes());
        out.flush();
        new listenThread().start();
    }

    public void addFile(gFile fileIn){
        files.add(fileIn);
    }

    class listenThread extends Thread{
        public void run(){
            try{
                while(true){
                    byte[] b=new byte[1];
                    b[0]=-2;
                    int errVal = in.read(b,0,1);//Take one byte - the action definer.
                    if(errVal==-1){//EOF reached.
                        break;
                    }
                    else{//Take the action byte and the data associated and pass it on to the processing function
                        byte[] c = new byte[in.available()];
                        errVal=in.read(c,0,in.available());
                        process(b,c);
                    }
                }
            }
            catch(Exception e){}
        }
    }

    public void connect(byte[] id){
    }

    protected void process(byte[] b, byte[] c){
        try{
            int action = b[0];
            switch(action){
                case 1:
                    setKey(c);
                    break;
                case 2:
                    createServerSocket(cipher.decrypt(c));/*
                    if (serverType==grgpLib.SERVER){
                        nThread = new neighborThread(grgpLib.byteArrayToInt(cipher.decrypt(c)),files);
                        nThread.start();
                        nThread.setServer(this);
                        outputThread = new clientThread();//Start a clientThread - which handles manual input. for debugging.*/
                        outputThread.start();/*
                    try{
                        byte temp=2;
                        out.write(temp);
                        out.flush();
                    }
                    catch(Exception e){
                        e.printStackTrace();
                    }
                }
                else{
                    System.out.println("Client connection completed.");
                }
                break;
                case 3:
                    if (serverType==grgpLib.SERVER){
                        setNeighbors(c);
                    }
                    else{//Search message
                        byte[] hash = cipher.decrypt(c);
                        System.out.println("files.size() "+files.size());
                        for(int i=0;i<files.size();i++){
                            System.out.println("i="+i);
                            gFile tmp = files.get(i);
                            if(tmp.has(new BigInteger(hash))){
                                System.out.println("Have the file. Sending response.");
                                byte temp=3;
                                out.write(temp);
                                out.write(grgpLib.intToByteArray(port));//Using port as grgpId
                                out.flush();
                            }
                            else{
                                System.out.println("I dont have that file.");
                            }
                        }
                    }
                }
            }
            break;
        }
    }

```

```

        case 4:
            if (serverType==grprLib.SERVER){//Response to a request for a connection. Returns the encryption key from the requested client.
                byte[] endPeerKey = cipher.decrypt(c);
            }
        case 5:
            if (serverType==grprLib.SERVER){//Act as middleman
                byte[] ips = cipher.decrypt(c);
                byte[] ip1 = new byte[4];
                byte[] ip2 = new byte[4];
                for(int i=0;i<4;i++){
                    ip1[i]=ips[i];
                    ip2[i]=ips[i+4];
                }
            }
        default:
            System.out.println("Invalid action request. Action id="+b[0]);
    }
}
catch(Exception e){
    e.printStackTrace();
}
}

public void setPort(int port){
    this.port=port;
}

public void setFiles(ArrayList files){
    this.files=files;
}

protected void createServerSocket(byte[] c) throws Exception{
    int tmp;
    tmp = grprLib.byteArrayToInt(c);
    servSocket = new ServerSocket(tmp);
}

protected void setNeighbors(byte[] c){
    c = cipher.decrypt(c);
    int num = c[0];
    System.out.println("num="+num+" c.len="+c.length);/*
    if (num==0){
        grprLib.dbLog("No peers available as neighbors.");
    }
    for(int i=1;i<num*8+1;i+=8){
        byte[] A = new byte[4];
        byte[] B = new byte[4];
        for(int j=i;j<i+4;j++){
            A[j-i]=c[j];
        }
        for(int j=i+4;j<i+8;j++){
            B[j-i-4]=c[j];
        }
        try{
            int port=grprLib.byteArrayToInt(B);
            InetAddress addr = InetAddress.getByAddress(A);
            System.out.println("Connecting to peer at: "+addr.getHostAddress()+" on port="+port);
            neighborSocket tmp = new neighborSocket(addr,port);
            neighbors.add(tmp);
        }
        catch(Exception e){
            e.printStackTrace();/*
            System.out.println("Could not connect to client");
        }
    }
}

public void search(gFile file) throws Exception{
    System.out.println("Searching "+neighbors.size()+" neighbors.");
    for(int i=0;i<neighbors.size();i++){
        neighborSocket N = neighbors.get(i);
        N.search(file.hashArr.get(0).toByteArray());
    }
}

protected void setKey(byte[] c){
    this.serverKeys = new grprCrypt(c);

    cipher=new grprFish();
    byte temp = 1;
    byte[] b = cipher.getKey();
    byte[] g = new byte[1];
    g[0]=(byte)1;
    g=grprLib.concatBytes(g,b);
    byte[] d = serverKeys.encryptBytes(g);

    try{
        out.write(temp);
        out.write(d);
        out.flush();
    } catch (Exception e){
        e.printStackTrace();
    }
}

public grprSocket(Socket sock){
    try{
        socket = sock;
        in = new BufferedInputStream(socket.getInputStream());
        out = new BufferedOutputStream(socket.getOutputStream());
    }
    catch(Exception e){
        ;
    }
}

public void sendString(String s){

```



```

        byte[] b = s.getBytes();
        try{
            out.write(b,0,b.length);
            out.flush();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

    public void sendBytes(int actType,byte[] payload) throws Exception{
        byte[] tmp = grrpLib.intToByteArray(actType);
        out.write(tmp);
        out.write(payload);
        out.flush();
    }

    public byte[] getBytes(){
        byte[] b =new byte[1];
        try{
            b = new byte[in.available()];
            int val = in.read(b,0,b.length);
        }
        catch(Exception e){
            ;
        }
        return b;
    }
}

```

The neighborSocket provides peer-to-peer networking functionality.

```

package grrp.common;

import java.net.*;
import java.io.*;
import java.util.ArrayList;
import java.math.BigInteger;

import grrp.common.*;

public class neighborSocket {
    BufferedInputStream in;
    BufferedOutputStream out;
    grrpCrypt keys,clientKeys;
    grrpFish cipher;
    grrpSocket serv;

    public neighborSocket(InetAddress addr, int port) throws Exception{
        Socket socket = new Socket(addr,port);
        in = new BufferedInputStream(socket.getInputStream()); //Buffer the output and input streams
        out = new BufferedOutputStream(socket.getOutputStream());
        keys = new grrpCrypt(); //Create a set of RSA keys.
        byte temp=grrpLib.CINIT;
        new listenThread().start();
    }

    public neighborSocket(Socket inSock) throws Exception{
        in = new BufferedInputStream(inSock.getInputStream()); //Buffer the output and input streams
        out = new BufferedOutputStream(inSock.getOutputStream());
        keys = new grrpCrypt(); //Create a set of RSA keys.
        byte temp=grrpLib.CINIT;
        new listenThread().start();
    }

    public setServer(grrpSocket sock){
        this.serv=sock;
    }

    class listenThread extends Thread{
        public void run(){
            try{
                while(true){
                    byte[] b=new byte[1];
                    b[0]=-2;
                    int errVal = in.read(b,0,1); //Take one byte - the action definer.
                    if(errVal==-1) //EOF reached.
                        break;
                    else //Take the action byte and the data associated and pass it on to the processing function
                    {
                        byte[] c = new byte[in.available()];
                        errVal=in.read(c,0,in.available());
                        process(b,c);
                    }
                }
            }
            catch(Exception e){}
        }
    }

    public void search(byte[] hash) throws Exception{
        System.out.println("searching");

        byte temp = 3;
        byte[] cipherText = cipher.encrypt(hash);
        out.write(temp);
        out.write(cipherText);
        out.flush();
    }

    protected void process(byte[] b, byte[] c){
        int action = grrpFish.byteArrayToInt(b); /*
        int action = b[0];
        System.out.println("foo = "+b[0]); */
        switch(action){
            case 0:

```

```

        setKey(c);
        break;
    case 1:
        setCipher(c);
        break;
    case 2:
        System.out.println("connection connected");
        getPeers(c);/*
        break;
    case 3:
        //Connect to peer c
        serv.sendBytes(grpLib.CCON,serv.cipher.encrypt(c));
    default:
        System.out.println("c.len="+c.length);
        byte[] d=cipher.decrypt(c);
        byte[] a=new byte[c.length-1];
        for(int i=1;i<c.length;i++){
            a[i-1]=d[i];
        }
        System.out.println("Invalid action request. Message content is as follows:");
        System.out.println("As a string: "+new String(a));
        System.out.println("As a BigInteger: "+new BigInteger(a));
    }
}

protected void setKey(byte[] c){
    this.clientKeys = new grpCrypt(c);

    byte temp = 1;

    byte[] b = keys.modulus.toByteArray();

    try{
        out.write(temp);
        out.write(b);
        out.flush();
    } catch (Exception e){
        e.printStackTrace();
    }
}

protected void setCipher(byte[] c){
    /*
    System.out.println("keys.decryptBytes(c)="+new BigInteger(keys.decryptBytes(c));*/
    /*
    System.out.println("c="+new BigInteger(c));*/

    byte[] d=keys.decryptBytes(c);
    byte[] b=new byte[32];
    for(int i=1;i<33;i++){
        b[i-1]=d[i];
    }
    this.cipher=new grpFish(b);

    byte temp = 2;
    byte[] jiggy,jiggy2;/*
    String temp2=new String("Connection created .");*/
    /*
    port = (new Random()).nextInt(63000)+1000;*/
    /*
    jiggy2=grpLib.intToByteArray(port);*/
    /*
    jiggy=cipher.encrypt(jiggy2);*/
    /*
    System.out.println("jiggy2="+jiggy2.length+" jiggy="+jiggy.length+" port="+port+" unjig="+grpLib.byteArrayToInt(cipher.decrypt(jiggy))+

    try{
        out.write(temp);
        out.write(jiggy);*/
        out.flush();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

}
}

```

The `neighborThread` runs a server on each client, accepting connections from other clients to assist in forming the mesh network.

```

package grp.client;

import java.net.*;
import java.io.*;
import java.util.ArrayList;
import grp.common.*;

class neighborThread extends Thread{
    int port;
    ArrayList neighbors = new ArrayList();
    public ArrayList files;

    public neighborThread(int port,ArrayList files){
        this.port=port;
        this.files=files;
    }

    public void run(){
        int ID=0;
        try {
            ServerSocket s = new ServerSocket(port);//Start listening on grpLib.PORT
            System.out.println("Starting server on port:"+port);
            ID++;
            while (true) {
                Socket incoming = s.accept();//When something connects, accept the connection
                grpSocket in = new grpSocket(incoming,grpLib.NEIGHBOR);
                in.setFiles(files);
                in.setPort(port);
            }
        }
    }
}

```

```

        grpLib.dLog("Client connected - spawning thread.");
        grpNeighborThread tempThread = new grpNeighborThread(in, this, ID++); //Make a new thread for the new client*/
        tempThread.start();*/
        neighbors.add(tempThread); //Keep an active list of all connected neighbors.*/*
    }
}
catch (Exception e) {
    e.printStackTrace();
}
}
}

```

The gFile is a wrapper around a RandomAccessFile. The gFile class is used for generating, checking and loading .grp hash files, as well as transfer of datafiles.

```

package grgp.client;

import gnu.crypto.hash.*;
import java.math.BigInteger;
import java.io.File;
import java.io.RandomAccessFile;
import java.util.Hashtable;
import java.util.ArrayList;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import grgp.common.grrpLib;
import grgp.client.grrpSocket;
import java.net.InetAddress;

public class gFile {

    IMessageDigest md;
    RandomAccessFile rAF;

    /*
    Hashtable hashes = new Hashtable();*/
    public ArrayList<BigInteger> hashArr = new ArrayList<BigInteger>();
    public grrpSocket socket;
    /*
    InetAddress addr = InetAddress.getByName("localhost");*/
    /*
    sock = new grrpSocket(addr,grrpLib.PORT);*/

    public gFile(File file, String mode, int mode2){
        try{
            if (mode2==0){//Create Hash File
                this.rAF=new RandomAccessFile(file,mode);
                md = HashFactory.getInstance("SHA-512");
                generateHashes();
                System.out.println("Hashes generated.");
                outputHashes(new BufferedOutputStream(new FileOutputStream(new File(Integer.toString(rAF.hashCode())))));*/
            }
            else if (mode2==1){//Import Hash File
                md = HashFactory.getInstance("SHA-512");
                importHashes(file,mode);
                String fname = file.getName();
                String rAFname = fname.substring(0,fname.length()-5)+".download";
                this.rAF = new RandomAccessFile(rAFname,"rw");
            }
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

    public void generateHashes() throws Exception{
        long len = rAF.length();
        byte[] b = new byte[1024*1024];
        for(long i=0;i<len;i+=b.length){
            rAF.read(b);
            md.reset();
            md.update(b, 0, b.length);
            byte[] digest = md.digest();
            hashArr.add(new BigInteger(digest));
        }
    }

    public void outputHashes(BufferedOutputStream out) throws Exception{
        System.out.println(hashArr.size());
        for(int i=0;i<hashArr.size();i++){
            byte[] A = hashArr.get(i).toByteArray();//64 bytes per
            out.write(A,0,A.length);
            out.flush();
        }
    }

    public void importHashes(File file, String mode) throws Exception{
        RandomAccessFile temp = new RandomAccessFile(file,mode);
        byte[] addr = new byte[4];//Get the IP address
        temp.read(addr);
        InetAddress serv = InetAddress.getByAddress(addr);
        System.out.println("Connecting to server at: "+serv.getHostAddress());

        this.socket=new grrpSocket(serv,grrpLib.PORT);
        this.socket.addFile(this);
        byte[] hashes = new byte[(int)temp.length()-7];
        temp.read(hashes);
        for(int i=0;i<(hashes.length/64);i++){
            byte[] hash = new byte[64];
            for(int j=i;j<i+64;j++){
                hash[j-i]=hashes[j];
            }
            hashArr.add(new BigInteger(hash));
        }
    }

    public boolean has(BigInteger digest){
        for(int i=0;i<hashArr.size();i++){
            System.out.println("i="+i);

```

```

        if(hashArr.get(i).compareTo(digest)==0){
            return true;
        }
    }
    return false;
}
/* public boolean has(BigInteger digest){
    return hashes.containsKey(digest);
}*/
}

```

The `grpClient` is the user facing application for clients. Most of the work is offloaded to the `grpSocket` and `clientThread` classes.

```

// grpClient.java
package grp.client;

import grp.common.*;
import java.net.*;
import java.util.*;
import java.io.*;
import java.math.BigInteger;

public class grpClient {

    grpSocket sock;

    public grpClient(){
        try{
            System.out.println("grp prototype client v.1");
            InetAddress addr = InetAddress.getByName("localhost");
            sock = new grpSocket(addr,grpLib.PORT);
            new clientThread().start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void setSocket(grpSocket sock){
        this.sock = sock;
    }

    public static void main(String args[]){
        new grpClient();
    }
}

```

The `clientThread` provides the user interface through which users specify what actions they want their client to take.

```

package grp.client;

import java.io.*;
import java.lang.*;
import grp.common.*;

public class clientThread extends Thread{

    gFile loadedFile;

    public void clientThread(){
    }

    public void run(){
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        String userInput=null;
        System.out.println("What do you want to do?:");
        try{
            while ((userInput = stdIn.readLine()) != null) {
                System.out.println("stuff.");

                if (userInput.startsWith("hash")){
                    System.out.println("Hashing "+userInput.substring(5));
                    File tempFile = new File(userInput.substring(5));
                    gFile tempGfile = new gFile(tempFile,"r",0);
                    String fileName = userInput.substring(5)+".grp";
                    File output = new File(fileName);
                    BufferedOutputStream oot = new BufferedOutputStream(new FileOutputStream(output));
                    userInput=null;
                    System.out.println("Done hashing. Enter Server IPv4 address, with periods:");
                    while ((userInput = stdIn.readLine()) != null) {
                        try{
                            byte[] temp = stringToIPBytes(userInput);
                            System.out.println("temp.len="+temp.length);

                            oot.write(temp,0,temp.length);
                            oot.flush();
                            break;
                        }
                        catch(Exception e){
                            e.printStackTrace();
                        }
                    }
                    tempGfile.outputHashes(oot);
                    System.out.println("Done. Hash output as "+fileName);
                }
                else if(userInput.startsWith("load")){
                    System.out.println("Loading");
                    File tempFile = new File(userInput.substring(5));
                    String fileName = userInput.substring(5,userInput.length());
                    fileName += ".download";
                    loadedFile = new gFile(tempFile,"r",1);
                }
                else if(userInput.startsWith("share")){

```

```

        System.out.println("Sharing");
        File tempFile = new File(userInput.substring(6));
    }
    else if(userInput.startsWith("search")){
        System.out.println("Searching network for the loaded file.");
        loadedFile.socket.search(loadedFile);
    }
    else{
        System.out.println("Sorry, I don't understand that command. Try hash filename or load filename.");
    }
}
}
catch(Exception e){
    System.out.println(":"+userInput);
    e.printStackTrace();
}
}

public byte[] stringToIPBytes(String strIn){
    String[] strings = strIn.split("[.]*");
    byte[] out = new byte[4];
    if(strings.length==4){
        for(int i=0;i<4;i++){
            out[i] = new Integer(strings[i]).byteValue();
        }
    }
    else{
        System.out.println("Not a valid IP.");
    }
    return out;
}
}
}

```

The `grpServ` class spawns `grpServThreads` which each handle an individual client.

```

// grpServ.java

package grp.server;

import grp.common.*;
import java.net.*;
import java.util.*;
import grp.common.grpLib;
import java.io.*;

public class grpServ {

    protected List clients;
    protected Hashtable grpIds;

    public grpServ() {
        clients = Collections.synchronizedList(new ArrayList());
        grpIds = new Hashtable();
        grpLib.dbLog("Server started on port "+grpLib.PORT+");
        try {
            ServerSocket s = new ServerSocket(grpLib.PORT); //Start listening on grpLib.PORT
            int ID=0;
            while (true) {
                Socket incoming = s.accept(); //When something connects, accept the connection
                grpLib.dbLog("Client connected - spawning thread.");
                grpServThread tempThread = new grpServThread(incoming, this, ID++); //Make a new thread for the new client
                tempThread.start(); //Start communication with the client
                clients.add(tempThread); //Keep an active list of all connected clients.
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        new grpServ();
    }
}

```

The `grpServThread` provides the server services required by the `grp` network to clients. A new thread is spawned for each connecting client.

```

// grpServThread.java

package grp.server;

import grp.common.*;
import java.net.*;
import java.io.*;
import java.util.*;
import java.math.BigInteger;

public class grpServThread extends Thread{ //Handles one client.

    protected Socket incoming; //The client

    protected grpServ owner; //The server which spawned the thread

    private grpCrypt clientKeys; //Client's public encryption key and Modulus (RSA)
    private grpFish cipher; //Client's encryption cipher (Twofish)
    private grpCrypt keys; //Server's encryption keypair

    private int bits = 2048;

    protected int ID;

    public int port;
}

```

```

BufferedInputStream in=null;
BufferedOutputStream out=null;

public grpServThread(Socket incoming, grpServ owner, int ID) {
    this.incoming = incoming;
    this.owner = owner;
    this.ID=ID;
}

public byte[] modulus(){
    return this.clientKeys.getBytes();
}

/*
protected String connect(int pid){
    int connectee=-1;
    int connector=-1;
    if (this.owner.clients.size()>2){
        for(int i=0;i<this.owner.clients.size();i++){
            if(((grpServThread)this.owner.clients.get(i)).ID==pid){//found connectee
                connectee=i;
            }
            else if(this.owner.clients.get(i)==this){
                continue;
            }
            else{
                connector=i;
            }
        }
    }
    grpServThread mid = ((grpServThread)this.owner.clients.get(connector));
    grpServThread end = ((grpServThread)this.owner.clients.get(connectee));
    return "3: connectillated";
}*/

protected void connect(byte[] c){
    byte[] d = cipher.decrypt(c);
    int connectee=-1;
    int connector=-1;
    if (this.owner.clients.size()>2){
        for(int i=0;i<this.owner.clients.size();i++){
            if(((grpServThread)this.owner.clients.get(i)).ID==pid){//found connectee
                connectee=i;
            }
            else if(this.owner.clients.get(i)==this){
                continue;
            }
            else{
                connector=i;
            }
        }
    }
    grpServThread mid = ((grpServThread)this.owner.clients.get(connector));
    grpServThread end = ((grpServThread)this.owner.clients.get(connectee));
    mid.middle(this,end);//Set the mid to be the middle peer
    out.write(4);
    out.write(end.modulus());
    out.flush();
}

public void middle(grpServThread start, grpServThread end){
}

protected void setKey(byte[] c){
    this.clientKeys = new grpCrypt(c);

    byte temp = 1;

    byte[] b = keys.modulus.toByteArray();

    try{
        out.write(temp);
        out.write(b);
        out.flush();
    } catch (Exception e){
        e.printStackTrace();
    }
}

/*
Takes an action type number and a payload.
*/
protected void process(byte[] b, byte[] c){
    int action = grpFish.byteArrayToInt(b);/*
    int action = b[0];

    switch(action){
        case 0:
            setKey(c);
            break;
        case 1:
            setCipher(c);
            break;
        case 2:
            getPeers(c);
            break;
        case 3:
            connect(c);
            break;
        default:
            System.out.println("c.len="+c.length);
            byte[] d=cipher.decrypt(c);
            byte[] a=new byte[c.length-1];
            for(int i=1;i<c.length;i++){
                a[i-1]=d[i];
            }
            System.out.println("Invalid action request. Message content is as follows:");

```

```

        System.out.println("As a string: "+new String(a));
        System.out.println("As a BigInteger: "+new BigInteger(a));
    }

}

protected void getPeers(byte[] c){
    List nList = this.owner.clients;
    Random rand = new Random();
    int desiredSize = (nList.size()>5?5:nList.size()-1);
    byte[] subList = new byte[1];
    int tmp =0;
    while(tmp<desiredSize){
        grpServThread temp = (grpServThread)nList.get(rand.nextInt(nList.size()));
        if (temp!=this){
            subList = grpLib.concatBytes(subList, temp.incoming.getInetAddress().getAddress());
            subList = grpLib.concatBytes(subList, grpLib.intToByteArray(temp.port));
            tmp++;
        }
    }
    System.out.println("Peers requested. Assigning: "+tmp+" out of: "+nList.size()+".");

    subList[0]=(byte)tmp;
    byte temp = 3;
    subList = this.cipher.encrypt(subList);
    try{
        out.write(temp);
        out.write(subList);
        out.flush();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public boolean equals(grpServThread obj){
    if(obj.getID()==this.getID()){
        return true;
    }
    else{
        return false;
    }
}

public int getID(){
    return this.ID;
}

protected void setCipher(byte[] c){
    System.out.println("keys.decryptBytes(c)="+new BigInteger(keys.decryptBytes(c)));/*
    System.out.println("c="+new BigInteger(c));*/

    byte[] d=keys.decryptBytes(c);
    byte[] b=new byte[32];
    for(int i=1;i<33;i++){
        b[i-1]=d[i];
    }
    this.cipher=new grpFish(b);

    byte temp = 2;
    byte[] jiggy,jiggy2;
    String temp2=new String("Connection created .");/*
    port = (new Random()).nextInt(63000)+1000;
    jiggy2=grpLib.intToByteArray(port);
    jiggy=cipher.encrypt(jiggy2);
    System.out.println("jiggy2="+jiggy2.length+" jiggy="+jiggy.length+" port="+port+" unjig="+grpLib.byteArrayToInt(cipher.decrypt(jiggy))+
    " unjig.len="+cip

    try{
        out.write(temp);
        out.write(jiggy);
        out.flush();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void run(){
    String line;
    keys = new grpCrypt();
    clientKeys=null;//To appease java.
    int clientState =0;

    try{
        in = new BufferedInputStream(incoming.getInputStream());
        out = new BufferedOutputStream(incoming.getOutputStream());
        while(true){
            byte[] b=new byte[1];
            b[0]=-2;
            int errVal = in.read(b,0,1);//Take one byte - the action definer, read into b
            if(errVal==-1){//EOF reached.
                break;
            }
            else{//Take the action byte and the data associated and pass it on to the processing function
                byte[] c = new byte[in.available()];
                errVal=in.read(c,0,in.available());//Read the rest of the data into c
                process(b,c);
            }
        }
    }
    catch (IOException e) {
        System.out.println("Read failed");
        System.exit(-1);
    }
    catch (NumberFormatException e) {
        e.printStackTrace();
        System.out.println("Poorly formatted input.");
    }
    catch (Exception e){
        e.printStackTrace();
    }
}

```

```
        System.out.println("Dumping client.");
        this.owner.clients.remove(this.owner.clients.indexOf(this));
    }
}
```


7 Conclusion

Unresolved Issues

Efforts have been made to inoculate GRRP against attacks which can harm other file sharing networks. First, GRRP does not use any one port, foiling attacks which attempt to profile traffic by port number. There are still a number of weaknesses however, including, but not limited to DNS poisoning, man-in-the-middle and peer poisoning.

DNS poisoning is unlikely to be a major issue, as the default is to use hardcoded IP addresses in the .grp files. This eliminates a possible source of attack, which FTP and HTTP services are very susceptible to. Although not an easy tradeoff to make when the user remember an address to get to the content, like with HTTP, it is an acceptable tradeoff for GRRP. With the hash files distributed through third party networks instead of entirely inside the network, the user does not need very intimate knowledge of what servers they are connecting to, similar to how Bittorrent trackers function. Like Bittorrent, this approach ensures that the server itself is not susceptible to such a simple attack at a single point of failure. Instead, every server hosting the .grp hash file is a redundant point of failure, greatly scaling the cost and reducing the feasibility of launching an attack aimed at disturbing availability of a specific file.

Man-in-the-middle attacks are characterized by an adversary with access to some piece of network, be it a router or a wire, between two clients on a network. By intercepting all the packets sent and replacing the public encryption keys with custom crafted ones, the attacker in the middle is able to observe and interfere with the traffic between the two clients. Peer poisoning uses a number of peers in the network that act in a coordinated manner to attempt to determine the identity of users sharing files. Assuming that the server is compromised as well, this attack may conceivably be successful in revealing the identity of a peer, who shares data with one of the poisoned peers.

Both of these types of attack can be prevented by having the server hold the public encryption keys for all the clients, distributing them as appropriate. But this has the drawback of leaving the server operator with a lot of information.

Areas for Continued Work

An optional server configuration could allow clients to submit transmission speed information. The server can use this information to pair up IPs with similar transmission speeds if the clients have also indicated that they wish for that option. This is a potential security risk, as it removes randomness from the setup of anonymized links in exchange for transfer speed. Allowing the same peers to connect to each other within a smaller group will then increase the potential risk of a key or payload being compromised. but this is not considered a major security issue for most applications.

8 Bibliography

References

- [1] Schneier, Bruce. Applied Cryptography, Second Edition. John Wiley & Sons, Inc., New York, NY. 1996.
- [2] Kurose, James F. Computer Networking, Third Edition. Pearson. Boston, MA. 2005.