Rendezvous: toward a distributed secure storage model

Kelly Nicole Kaoudis
kaoudis@colorado.edu

ABSTRACT

We describe a distributed anonymous filesharing network based on insight from studying the efficacy and usability of anonymity-providing services. First we justify two related pillars of our approach: security built in from the start everywhere possible, and realizing the implications of usability (or lack thereof). We detail related literature and our takeaways from others' failures and successes.

Our work originates with prior study of the construction of the Tor network's performance, the onion protocol, and interaction between Tor clients and hidden services. We do not use Tor itself as the foundation for our system due to performance and protocol concerns. We also incorporate design lessons from currently widespread implementations of peer-to-peer filesharing systems, distributed databases such as Kademlia and Pastry, and key management systems such as public files and functionally trusted third parties.

Recently we have begun considering implications of Byzantine failure and other potential faults for our system. In light of these concerns, the third pillar of our system is simplicity. This means reducing the number of technologies and techniques used, without compromising our other priorities, characteristic security and usability. The fourth pillar of our completed system design will be modularity, as in the familiar Unix philosophy. Each element should work as a cog in the greater system without disrupting the user's experience of the system as a seamless entity. Components will be independently testable and secured, composable, and reusable as needed as stand-alone modules in future work. We detail the implementation to date and possible issues with the continued execution of our design.

1. INTRODUCTION

We first describe the basic principles our distributed temporary file rendezvous is based on.

Security

While the Internet is a multinational, borderless entity, its cultural heritage is based in the expectation of being able to speak freely and exchange ideas and information with anyone, at any time. Tor [14], Freenet [8], and I2P [53], which all claim to allow their users to transfer and/or receive information without needing to expose real-world credentials such as legal names and IPs, are utilized nearly worldwide. This speaks to a strong desire for uncensored communication.

The United Nations' Universal Declaration of Human Rights details certain freedoms to which all human beings are entitled. Its twelfth and nineteenth articles in particular, similarly to the First Amendment to the United States Constitution and Section 2b of the Canadian Charter of Rights and Freedoms, state that all humans should have the right to freedom of expression and opinion and should not be subject to arbitrary interference with their assembly and privacy.

If free speech in particular is to be effectively realized, security and user privacy must be top considerations at all levels of application design. When security is not a bolt-on afterthought but rather is included in the specification from the beginning [47], it is much more difficult to compromise. In the interest of the continuation of a free and open web [25], we consider this the most important principle our design is based on.

Usability

Our second concern is the ability of our users to easily understand and employ our service [3] [56] without accidentally revealing personally identifiable information or other sensitive data. Our system should offer the fewest (unpleasant) surprises possible to the user both in terms of how the interface works and how the underlying system operates [44].

The general public expects the convenience of being able to launch any sort of application with little fuss. Additionally, popular virtual private storage / storage as a service providers such as Dropbox and Apple's iCloud do nothing to dissuade the general belief that properly securing data is trivial, or that one's data will be secure once entrusted to the service provider.

We argue that neither of these suppositions are currently true, especially in light of the high-profile breaches [30] [36] of such services that have occurred in the last few months, spreading the 'private' data of public figures across the internet. However, if a service requires extra effort and time compared to 'the usual', it is unlikely that most human beings will put in that effort even if it is understood that they

should (e.g. avoiding dentist appointments, and similar sociological phenomena).

While Tor and similar services were designed with some measure of usability in mind [12], we argue that it is possible to do better [23], especially considering recent events: hidden service operators and certain users likely lacking the technical knowledge and/or desire to read and understand parts of the Tor specification relevant to configuring their torrc file and software properly have had their anonymity and privacy compromised.

Simplicity

Snowden argues that some possible adversaries are capable of a trillion hashes per second [22], so it is logical to use as much layered, strong encryption as we can manage without overcomplicating the user experience or the back end code. File storage and retrieval is entirely based on whether one has the right PGP keypair [33], and all information entrusted to the system is encrypted and hashed with source and final destination in mind before it is temporarily stored in the cloud. Encrypted transit protocols are also employed both to/from the system and between storage nodes within the system itself.

During the beginning of the implementation process, we realized our design at the time used a lot of moving parts simply because the technology was interesting, new, or partially did one thing we needed. Byzantine faults are naturally more likely to occur in a more complicated system [12]: there are more parts that can fail to communicate properly with each other or operate in the desired fashion.

Failure due to Byzantine fault(s), a concept stemming from the Byzantine Generals problem, is a common affliction of distributed systems in particular, and a potential avenue for atackers to exploit (if Byzantine faults are likely). The less complicated and more streamlined a system is, the easier it is to implement redundancy and guard both against faults and malicious exploits of this nature. It is thought Byzantine failures cannot occur if there is no need for system-level consensus (even if Byzantine faults may happen) [16]. Yet data and hardware redundancy alone cannot protect fully against faults that cause the simultaneous failure of multiple types of component. The probability of simultaneous faults is small, but still something that should be taken into account from the start. Preventative actions such as component self-checks are thought to help reduce such occurrences. Future work should include incorporating thorough Byzantine fault *detection* in addition to tolerance through some level of redundancy and simplistic self-checking mechanisms.

Furthermore, the simpler and more sturdy a system and its parts are [48], the easier it is to extend that system and reuse its components for other projects. The simpler, more flexible, and more extensible a system is, the more likely we are to convince others to hack on and improve it.

Modularity

After simplifying as much as we could at the time, we realized we had considered the current wants and needs of the user and the author, but had failed to account for future individuals who might want to pick up this project or some part of it and make use of it in other ways. Therefore, we include the concept of modularity [44] as our final underlying design concern. Each component should do just one thing [2], should be composable, and should be designed and tested with limited reliance on elements belonging to other components. These modular units should compose to create the foundation of a system that fits the following problem statement and is addtionally extendable.

Design Overview

Our ideal is a simple, accessible, extendable temporary storage service that allows anyone to share files once (or multiple times) that are protected by encryption and randomization. That is, Alice should be able to easily exchange important information with Bob without allowing anyone else knowledge of what is being transferred or where specific data are located at any point during the process.

Neither Alice nor Bob should need extensive low-level technical knowledge to properly configure and use such a service in a secure, correct fashion. Neither the adversary nor Bob should know Alice's geolocation or IP. Neither Alice nor the adversary should know Bob's geolocation or IP. The service should keep the adversary from having knowledge of the mapping between data and users' personal information such as location and real name, while protecting data integrity.

We believe further exploration in this space is important. Anonymity is not a solved problem as can be be seen from the number of attacks publicized recently on popular privacy and anonymity systems such as Tor [27] [31] [5] [10] [28]. We propose a secure service that fulfils the pre-existing general expectation that data in the cloud is safe. We wish to facilitate anonymous, distributed information sharing without bias or penalty.

2. RELATED WORK

We present relevant parts of a few examples of work in the security and privacy space, and explore differences and similarities with our own architecture.

Tor and the onion protocol

Tor [14] is a widely used, well documented project claiming to provide more security than its users would have openly conducting business. The Tor FAQ [54] states that filesharing is not desired or to be the main focus of operation within the Tor network, especially using P2P protocols, which are not anonymized by Tor and SOCKS proxies in general due to a protocol mismatch.

In direct contradiction to methods employed by Onionshare [32] and similar services deployed on top of the ex-

isting Tor system, we chose to roll our own underlying system, reimagining and integrating only the features we really needed from Tor, mainly the hidden service/introduction point model and the anonymity given by the latest iteration of onion routing. Due to the nature of our project, it seemed most sensible to recycle concepts we found applicable to anonymous filesharing without including extraneous moving parts.

We argue that Tor compared to the open web [15] is slow primarily due to an imbalance [21] between users and certain kinds of contributors. Further, there are load balancing issues to do with circuit creation [1] and something of a bottleneck due to an imbalance in *kinds* of volunteer contribution exists presently. Due (we hypothesize) to the Tor system's notoriety in mainstream media and additionally the general public's lack of understanding of Tor's inner workings, the number of people who contribute back [42] by running a relay or an exit node or improving the codebase is much smaller than the number of people only using the service to protect themselves. The number of volunteers running an exit node, a critical part of the infrastructure, is currently smallest of all [43], due to a history of abuse by malicious users and backlash from ISPs. Systems such as TorWard [34] are being developed to assess and reduce the risk of ill-meaning traffic to exit node operators and other Tor volunteers, but work in this space is just beginning. Our future work may include contributions to Tor in this direction.

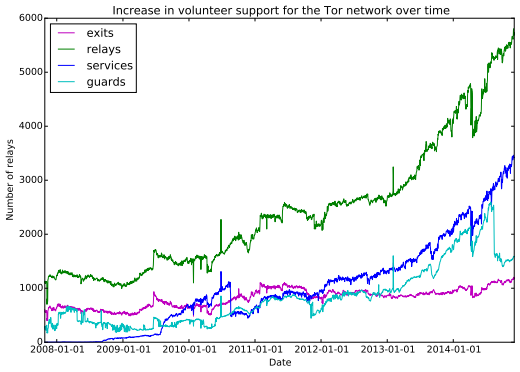


Figure 1: From an afternoon's work classifying and visualizing the current full span of official Tor Metrics server-side data, it is evident that relays and bridges with the 'exit' flag declared comprise the smallest number of active nodes at this time.

Furthermore it has been shown [7] using the Cisco traffic analysis tool Netflow that it is possible to uncloak around 81% of Tor users, especially in cases where the users did not have enough deep knowledge of the inner workings of the service. While we strongly advocate for an improved Tor, since it is already in widespread use, we propose a single-purpose partial alternative that addresses some of Tor's limitations.

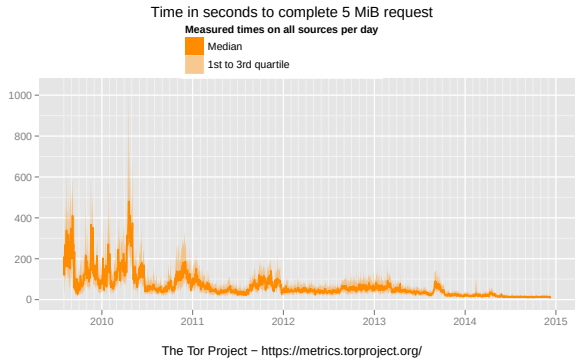
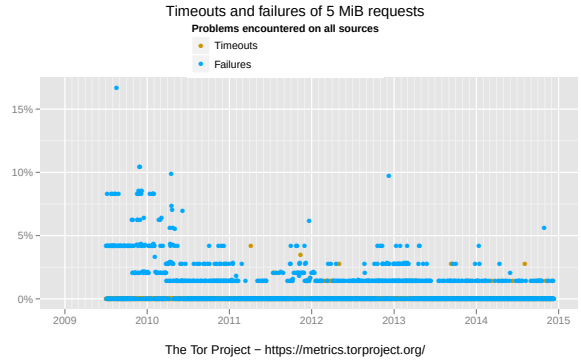


Figure 2: Performance of the Tor network has improved at an impressive rate as more volunteers have added bridge and regular relays. However, DNS resolution of open web resources accessed over Tor is still much slower than is evident from these official charts (which only include data collected when requests for hidden services or other data available only through Tor are registered), as is download or transfer of items greater than 5 MiB.



Not implementing our service over Tor gives us the freedom to use UDP at the transport layer where desired (SOCKS proxies only protect TCP traffic) and also to employ a specially tailored peer-to-peer protocol for uploading and downloading files from the network.

Tor offers users the ability to set up rendezvous points known as "hidden services", or alternatives to sites on the open web offering anything servable over a SOCKS proxy. While the idea is that operating such a service would be anonymous in that the operator's IP address and physical location should never be revealed, it has been proven that this assumption is flawed in some cases [10]. Hidden services, by way of a "hidden service descriptor", which consists of the service's public key and a list of its acknowledged introduction points, can advertise themselves in a globally available distributed hash table directory. The descriptor is what's first found when a user types the service's DESC.onion address into their Tor browser's URL bar. The "DESC" is a base-32 encoding of a 10-octet hash of the service's public key.

Tor currently uses an incremental path-building design: each successive hop in a circuit (a path between a user's Tor client's entry point into the network and her destination) has to negotiate its own session keys. We include this kind of modularity in our initial design; transport of a given chunk of data within the database is roughly modelled after Tor circuits. Tor multiplexes many TCP streams on a single circuit to allow for a noisier background and also provide more efficiency than just one stream on a circuit would have. Therefore the next step from the client point of view in navigation to DESC.onion once the service descriptor has been fetched from a randomly chosen hidden service directory is as follows. The client picks one of the service's introduction (rendezvous) points, and establishes a rendezvous circuit which will facilitate connection to that introduction point. Once the introduction point has been established by the client, the *service* builds itself a new Tor circuit ending at that point as well. The rendezvous point authenticates and relays cells between the client circuit and the server circuit. We also incorporate the idea of rendezvous points into our design in two places.

I2P

Somewhat like Tor, I2P [29] is an anonymous network based around the *I2P router*. I2P is designed to safeguard typical internet functions such as email, file sharing, messaging, and so on, mainly within a separate network called the *darknet* which is built atop the bespoke UDP and TCP-like protocols SSU and NTCP. These protocols run on top of the peer-to-peer network, which itself runs on top of IP.

Unlike Tor and similar architectures, the peer-to-peer network that is I2P's ground-floor layer is completely decentralized. Decentralization implies better scalability, and of course no central trusted authority which may be potentially compromised. All peer and service metadata is instead stored in a distributed hash table called *netDB*. Network database records are stored in another Kademlia-like DHT [37] *floodfill*. Since netDB is run alongside regular I2P nodes, it may be considered less a trusted *group* of authorities than functionally interwoven in the network.

All connections inside the darknet are end-to-end encrypted; the router multiplexes these connections over paired, single-direction *I2P tunnels*, which are (when taken as a paired connection from server to client) analogous to Tor's circuits. These tunnels are constructed with the onion protocol. All participants are meant to be anonymous.

However, in practice this does not always appear to be the case. I2P is a much smaller project than Tor; this means fewer users providing white noise around one's traffic, fewer developers, and fewer volunteers running services.

Additionally, the port of Tahoe-LAFS [58] allowing decentralized storage similar to Freenet within I2P is not maintained, and is not up to date with the current version of the filesystem. The website for the maintainer listed by I2P was down at time of writing. Actually, the only version the author could find that wasn't available from the Tahoe website was available as a package in the Arch Linux User Repository, and was not the port to I2P but rather the original version.

The largest difference between Tahoe and Freenet, according to Tahoe's creator [57], is Freenet's file distribution is entirely random and proper usage of Tahoe-LAFS depends on having a large network of friends and family one can cajole into also using the system, since Tahoe assumes discovery of storage happens out-of-band. Further, Tahoe makes no attempt to obscure the source or destination of requests. While it is exceedingly difficult to guarantee a high probability of remaining anonymous against sophisticated attackers (see the Tor section above) that doesn't mean that trying (and explicitly stating that users employ the service at their own risk) is not worthwhile.

Freenet

Freenet [8] is a location-independent peer-to-peer distributed filesystem of nodes that collaborate to store and retrieve encrypted files, which are named by 160-bit SHA-1 hashes that do not depend on the files' location. Each user maintains their own local datastore as a portion of their hard drive which might otherwise go under-utilized. Freenet maintains a *hops-to-live* limit on active requests for files, which is decremented at each node the request reaches, to prevent infinite cycling through the system.

We repurpose the *time-to-live* concept detailed by the IP specification and employed by Freenet to limit the number of downloads that can take place before the space dedicated to a file's chunks and identifier will be repurposed to hold other information (Freenet uses an LRU caching policy to determine what gets deleted from an individual node when there is not enough available space). Further, we do not store information locally on users' machines– our system requires obtaining a virtual private storage space to contribute instead.

Peer-to-peer transactions

Other cooperative peer-to-peer services like BitTorrent [9], Gnutella [45], uTorrent [55], and more experimental distributed architectures attempt to meet the goal of providing file transfer and/or temporary storage in a relatively private way. However, in the case of services based on the BitTorrent protocol and similar, a UDP-based tracking method for keeping a list of users' IPs and ports and passing it between peers is typically necessary. Every user whose torrent client requests the list of peers from which to obtain pieces of the file in question is not anonymous to those peers; nor are those peers anonymous to the user in question.

Distributed information stores related to the peer-to-peer model such as Pastry [46], Chord [49], Kademlia [37], Comet [20] and so on have advantages over widely used peer-to-peer services including greater consistency and fault tolerance, but none quite provide the kind of intrinsic, random-

ized, modular security we envision.

In our system, each file chunk gets its own peer-to-peer connection to a randomly selected node of the datastore, and then a further random transfer is made. Each of the initial peer-to-peer connections is untracked, and the destination node is not known to the client until transfer of a given data chunk is initiated.

Why Johnny can't stay anonymous

Johnny can't encrypt if the available encryption is non-intuitive to use. Whitten et al discovered by way of lab trials that PGP's user interface [56] was difficult to properly employ even for people with some technical background, at time of publication.

While the field of user experience has made huge progress since then and is one of the foremost considerations in many sorts of application and tools design, usability is still less frequently considered when security applications, tools, and frameworks are developed. If Johnny has some information he needs to get to Alice without others virtually reading over his shoulder, he shouldn't have to become an expert in low-level topics in order to understand and properly configure such services as will let him encrypt, sign, and send his documents with little fuss and without trusting a third party.

Let's say our hypothetical good guy Johnny is a journalist, activist, or just someone in a non-technical role at a corporation or other entity with shady regular practices which are actively hurting people or will in the long run be highly detrimental to a significant sector of the population. If services such as Tor are hard to set up or require too much knowledge Johnny doesn't already have or can't acquire quickly [40], he won't be able to get the word out. He might even be arrested if perhaps the use of such services is against local law. We propose creating a simple client-facing architecture and testing it on as many actual humans as we can convince to sit still long enough to try out Rendezvous before making it publicly available.

3. METHODS AND ARCHITECTURE

Our system has two main parts. The first is a clientside API which allows the user to prepare files for peer-to-peer upload, keep track of current in-progress and completed uploads, and orchestrate downloads. The second is a secure distributed database [13] [41] called a *rendezvous*, which (taken as a black box) behaves similarly to Tor's hidden service introduction points. The rendezvous consists of a layer of encryption and randomization [38] for data over storage space located on VPS platforms and contributed to the database as a requirement for uploading one's own data, pay-to-play style. These two parts will both operate on the assumption that *data is only sent when it is asked for by the receiver* [4].

Identifier storage and discovery

The storage of clients' onion addresses mapped to hashes of their available files and the files' destinations is handled by a much smaller (additional) distributed hash table that sits alongside the rendezvous. This *metadata store* is a functionally trusted entity which is readable by any who have uploaded their own credentials. Those who maintain their credentials within the metadata store may GET any data matching their public key, so long as they also have the file source's full public key.

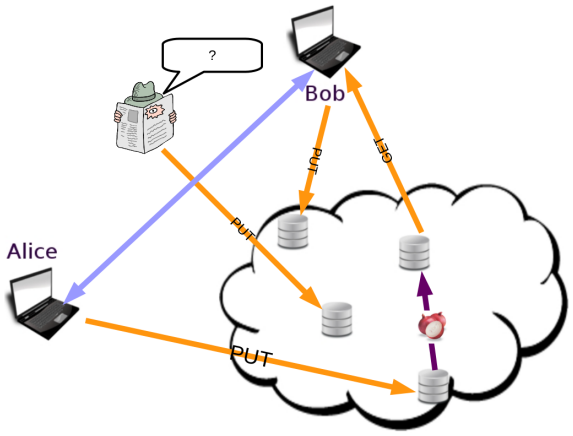


Figure 3: We present a simplistic overview of our system. Ideally, Alice and Bob first exchange public keys out-of-band. Then, Alice (chunk by chunk) uploads the data she wishes to share with Bob by making peer-to-peer connections to randomly selected datastore nodes. Finally, Bob acquires the file hash, queries for the file, and acquires it chunk by chunk, checking against the file hash to determine when he has the entire file. If all file chunks are not recovered within a certain time limit, Bob asks Alice to delete the file from the datastore and send it again.

Suppose Alice wants to send a file to Bob. She first registers an identifier [24] which, for our purposes, consists of the *onion id* taken from a hash of a PGP public key either previously in use and added to Alice's client application, or generated for her clientside. An onion id, similarly to Tor's "onion" URI [51], is the base-32 encoding of a 10-octet selection from a SHA2-512 hash of Alice's public key. If she would also like to upload content, the hash of the (encrypted) file in question is also calculated.

Assuming Alice has obtained the public key(s) of her file's recipient(s) out-of-band, she can essentially concatenate each one at a time with her public key and the file and take the hash of that. Data given a destination in this fashion cannot be reassembled and pulled out of the store without a public key that, hashed with the file and the public key of its origin point, matches the hash value attached to the file (and that data, once recovered in full, additionally cannot be decrypted without the matching private key). Privately destined data cannot even be *found* without knowledge of the public key belonging to its origin and a public key of a specified recipient.

There is some possibility for malicious users who have their own data in the store to pull out random onion addresses from the directory and try to get those users' files, but as long as the malicious users' keys don't match any part of the destination hash list, the unsuspecting "good" users' data should remain safe. Data without destination should naturally not be as sensitive as data directed at a specific entity. In the client interface, the default option will be to add a destination (or multiple destinations – which turn the single hash into a list of possible hash matches); the option to upload data without a specific recipient will be available but slightly less visible.

Alice's .onion id and a list of hashes of her full public key, the hash of the file, and the public key(s) of any recipients are sent

```
PUT { onion : idA,
      dest: { h( h(f), pubkeyA, pubkeyB),
              h( h(f), pbukeyA, pubkeyC),
                   ... },
      TTL : x }
```

to the metadata store. TTL, which stands for time to live, is an eight-bit integer historically indicating the time an IP datagram has left to exist. For instance, when using the network utility *ping*, each ICMP packet sent out has a certain number of hops between routers it can make before it is discarded.

Our version of the TTL value accounts for the number of downloads that can take place before the shards of Alice's file will be unconditionally written over. The default TTL will be one. If the TTL is blank, the destination list must also be blank. If both of these are true, the file will be considered open for public download until its originator sends a DELETE command on the hash of the file to be deleted to the key,value store. The list of destination hashes, the time-to-live value, and Alice's address will also be attached to fragments of the file such that each can be retrieved from its rendezvous point.

Clientside control: GET, PUT, DELETE

A client has the ability to perform three operations on the data store. One PUTs one's onion id and the hashes of any files available, then one PUTs the file chunks themselves. The file receiver will perform a GET at their convenience. The file originator's public key as well as the receiver's own onion id and public key (not to be stored) are sent to the metadata store to be verified against the file in question. A DELETE can only be issued by the file originator, in much the same way GETs can only be issued by those specified in the file's list of destination hashes. All of these operations are signed by their originator in much the same way a given encrypted, whole file itself is.

We chose to use PGP clientside to encrypt and sign data and requests. It has existed since roughly 1996 [6] (with no publicized successful attacks found during the author's research into file encryption), so we infer the system is secure enough for now. At a later date, we may replace OpenPGP with a bespoke implementation of the user's choice of proven-secure, time-tested asymmetric algorithms. The concept of the Web of Trust and key signing/verification by other users (so far) are outside the scope of this work. Whether imported keys have been verified by other parties or a newly generated key has only been verified by the user, she is still able to encrypt and sign documents.

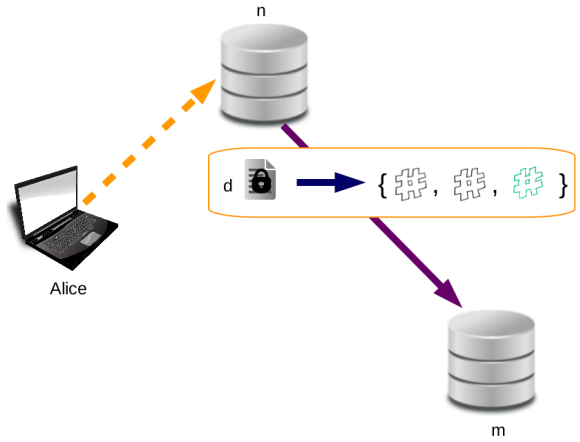


Figure 4: Operations performed clientside include generating or adding a PGP keypair; encrypting, hashing, and sharding the desired file; uploading credentials; upload and retrieval of file chunks via multiple TLS-encrypted peer-to-peer connections. When data upload takes place, the client first encrypts the file, then takes its hash. The file is sharded, and each shard is tagged with the whole-file hash as well as Alice's "onion" id. Each file chunk is individually uploaded to a different randomly selected node in the datastore.

The clientside API is currently written in node.js and uses the cryptography libraries crypto-js and openpgp.js. The only thing we are currently using from crypto-js is a SHA2-512 generation function. Since openpgp.js also includes SHA2, in the interest of keeping things as simple as possible it may make sense to discontinue use of crypto-js and substitute with the openpgp.js equivalent, or even roll our own SHA2 function.

Before any upload is allowed to take place, Alice's file(s) must be encrypted with her keypair. The encrypted files are then signed, and a destination list is added to the file encryption via OpenPGP. The file is then chunked. Each chunk is transmitted individually to a randomly selected datastore node over LEDBAT over TLS over UDP. What this means is that a TLS-encrypted channel for UDP datagrams is created between Alice and each datastore node, and peer-to-peer upload of each chunk then occurs.

Currently, we've facilitated this in sort of a hacky manner – a local Express (node's web server) instance is started

on a UNIX socket. As soon as file selection takes place, files are AJAX-ed over to the local "server", then encrypted and sharded before being sent onward to the database. A message stating that file upload is taking place is returned to the "client" as soon as the first P2P transaction is started, and it along with information received in exchange for the encrypted file shards from the datastore will be parsed and displayed to the user in responsive, simple form. We plan to use angular.js and d3.js to complete the clientside UI. We were planning to collaborate this semester with an interested computer science undergraduate on the UI design, but it appears we may have to include this as well within the work to be done in the next month on this project. At this time we have gotten the API working (totalling roughly 550 lines of javascript) and need only complete the "design" aspects of the user-facing work to have a finished, polished client.

Next steps after completion of the UI design (assuming enough of the network-level work as also been completed) will be to start convincing members of the general public to try out our UI and ensure we haven't run directly into the sort of pitfalls discovered by Whitten with PGP's original interface.

Efficient uploading to temporary data storage and crude randomized load balancing

Each person who is allowed to upload data is also running their own storage node. However, the choice of destination points for file chunks is as random as possible given certain factors like storage space available on the chosen nodes and number of nodes currently on the network. We select a simple random sample of *n* destinations, where *n* is the number of file chunks. We re-select from the list of nodes currently up in the case that a given possible destination's storage space is occupied up to the high-water mark. It is possible that a chunk of Alice's file could end up in Alice's storage node, though hopefully by using multiple random number generators and several selection steps, most of the file will be stored elsewhere.

Destination points are randomly assigned, using a different random number generator [39] from the one used for node selection, from the list to chunks of data. Ideally, a mixture of data chunks from many sources would be stored on Alice's node. Once upload is complete, the metadata store returns a confirmation to Alice as well as a reasonably current estimate of service-wide statistics.

Within the datastore itself, all transactions between nodes are automated and secured with session keys negotiated at transaction time. Each node has a persistent address, but the AES session keys for onion-style communication between nodes are merely temporary and last for exactly one connection between a given node *n* and a randomly selected endpoint *m*, over which multiple data chunk transfers might occur if the same destination is selected multiple times.

Each chunk of data makes at minimum one more randomized hop after being added to the database. The initial destination point *n* for a chunk of data *d* will randomly select another node *m* from the remaining available nodes (if the chunk comes from anywhere that isn't another node in the datastore), negotiate a set of session keys, and construct an onion-style circuit to *m*. The data chunk is sent over the circuit and the session keys are discarded. If *m* has reached its highwater mark, another secondary destination is chosen by *n* prior to key negotiation.

When Bob wants to retrieve the file Alice has left for him, he issues a GET including his public key, his onion id, and Alice's public key (obtained out-of-band) to the key,value store. The store returns the onion id where each file chunk is located, and the complete file's hash with the provided identifiers $h(h(f), idA, idB)$. Bob's client is now armed with the information needed to retrieve the full file. The client is able to resolve each onion address to the proper storage node by engaging the metadata store.

Bob makes a GET request to each onion id provided along with his hash, which starts a *trackerless* peer-to-peer download (in the style of the "trackerless" Kademlia-based distributed sloppy hash table method used by [35] [9] [55]) from each relevant node of the rendezvous directly to Bob of the chunks of the encrypted file. As Bob's storage node acquires each file chunk, he checks against his hash to ensure he's got it all correctly. Potentially the rendezvous point already checked the sum total of file chunks tagged with the hash of Alice's file against the hash itself when Alice was uploading the file, as a sort of "pre-flight" integrity check. This sort of checking would eliminate the possibility of Alice sending a file to the rendezvous point that does not match the hash she already registered with the network, but might make upload time prohibitive. We leave the benchmarking of these two options to future work.

Download procedure and probability

The download procedure is illustrated in Figure 5. We chose to use floodfill, a simple graph-searching algorithm often used as the basis for the paint bucket tool in drawing programs, as our search method. Floodfill is essentially an improvement to breadth-first search, though in this case selection of nodes to search next is completely random. Our search ordering depends (instead of going strictly by breadth) on the data structure containing nodes that have already been searched and should not be examined again, and on how good our random number generators are.

As floodfill is relatively simple and straightforward to implement and in the average case less than *O(n)* where *n* is the number of nodes that are up at a given time, we believe it is at the very least a decent starting point. In future, we will explore the literature further to determine if others have implemented successful/efficient randomized methods for graph searching (or more efficient graph-based algorithms that could be adapted to randomized next-item-to-search selection effectively), though we currently hold the (probably uninformed) opinion that this adaptation of flood-

fill is somewhat novel.

Retrieving a file from the database is a non-deterministic operation in that the time to collect all the chunks may vary because their placement is randomized, but so long as the number of file chunks and the number of nodes online where the chunks have not yet proven to *not* be located are finite, recorded, and available at search time of a given node, the file-collection operation is overall finite, though it will potentially increase as the number of available places for the file to be located increases (as volunteers join the network). It is possible to make an average-case estimate (essentially calculating the expected value) for how long retrieval of a given file would ideally take, treating the system as a hypergeometric probability distribution [39]:

$$P(x, N, n, K) = \frac{[\binom{K}{x}] * [\binom{(N-K)}{(n-x)}]}{\binom{N}{n}} \quad (1)$$

We have a set of n nodes randomly chosen without replacement (we colour all no's and end each search branch on a yes) from the list of N nodes currently up. K chunks of the file exist in the database. Each random sample n has x successful picks in it; the probability of getting a success changes each draw. Essentially, this breaks down to finding K (or more) successes. Given the addition of some level of data redundancy, the probability of success will increase, but drawing two identical chunks (just as a simple example) would also be possible.

If randomized file chunk retrieval becomes a problem in terms of how long the operation takes overall as the number of nodes in the store increases, we plan to try a nearest-neighbour scheme as our second choice. We believe that randomizing the search for file chunks and having it occur entirely automatically following discovery of the first node by the client will help in protecting their identity, so long as we can sufficiently secure the client's initial connection to the database and each connection transporting a file chunk back to Bob. Each connection between nodes, as with the movement of regular data, is secured by a set of throwaway AES session keys in a manner similar to a single hop within one of Tor's onion routing circuits.

Node entry, exit, and information loss

Rendezvous nodes will be mostly run by volunteers who use the service regularly. Whenever Alice has spun up her contribution to the network (her storage node), she is allowed to upload her onion address and list of files on offer. Thus, node entry can happen at any time, at the client's will. Node exit, similarly, is random, depending upon how long the user wants their node operational. Ideally, they would leave their storage node up all the time and ensure greater storage capacity as well as randomness for the network, but this might not always be the case.

Information loss could naturally occur when a node unexpectedly drops out of contact with the rest of the data store.

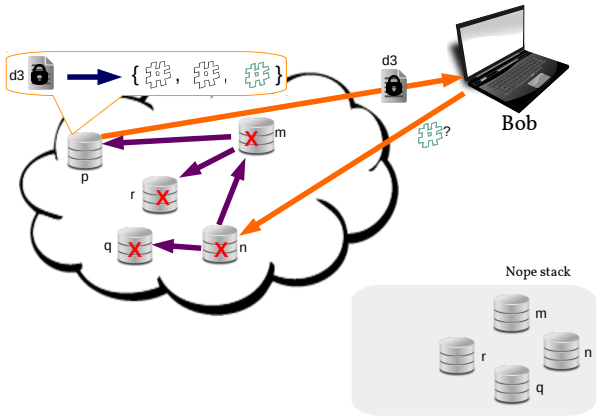


Figure 5: Bob is ready to download a file. The destination list of this file includes a hash based on Bob's public key. Bob's client discovers a Rendezvous node and makes a request to the metadata store with Bob's public key, Bob's onion id, and the file originator (Alice)'s public key. The node searches itself, but no chunk tagged with Alice's public key is found. The node (n) adds itself to the stack of "nopes" and asks a randomly selected set of neighbors, all of which (recursively) perform the same operation until a file chunk is found (then that particular branch of the lookup is finished, and peer-to-peer download of the chunk to Bob's onion id is initiated). Unless the file does not exist, the full set of nodes in the network should not be searched except in the worst case. This lookup procedure is a variant on the floodfill graph-coloring algorithm, commonly used in computer graphics to fill large spaces of pixels. No node that has already been coloured (added to the stack of no's) should be searched twice, and no node containing a piece of the file should be searched twice as that B branch of the search should terminate.

However, we know that having perfect replication of all data across all nodes is unsustainable. In the future we plan on benchmarking to determine the proper amount of data redundancy across nodes. Due to the nature of the network, the redundancy will also be randomized; but no two pieces of information that are the same should be stored on the same node.

A shared-nothing (no storage space of any kind is common to any two nodes) multiple-node model with limited data replication would provide both reasonable fault tolerance and additionally (hopefully) reduce the number of nodes examined to fully reconstruct a file.

Each node, upon selection by the metadata store as a potential initial location for a file chunk, is checked to see if the amount of storage space available is less than that node's declared highwater mark. Each highwater mark is unique to the amount of storage space initially obtained for a given

node by its owner.

Node architecture

Initially, we considered using sqlite as the foundation for each node instance, but this proved to be more functionality than needed per node. We require no set-theoretic (or MapReduce-style) operations whatsoever, given that no computation over the data is done at any point other than load balancing and basic existence checking in our current model. We avoid the relational model entirely by having a separate metadata store to field queries about identifiers and treating the actual datastore as a giant (semi-sloppy) distributed hash table. We explored the possibilities offered by the NoSQL model and have currently settled on an implementation akin to the Apache Foundation's CouchDB [19]. Users can perform the equivalent of a handful of HTTP-esque commands that merely depend on successful hash lookups of the data in question: GET, PUT, and DELETE as discussed above. Each node's datastore instance simply maps a list of keys (file hashes) to a list of one or more data chunks. If a file is not marked for any particular destination, it is downloadable by everybody who has the ability to get the file originator's public key. Within the data-store, an "undestined" data chunk's tag is simply the hash of the hash of the file with the source's public key $h(h(f), pubkeyS)$. Future work (as suggested during the presentation of this work) will include looking at HDFS and Google's File System to see if they can be modified to better suit our purposes.

4. CONCLUSION

We present some justification for and description of a service that allows anyone to protect the privacy of their data while sharing it as desired. This service is dependent on a volunteer-run network of nodes that together comprise a distributed hash table for files and a metadata store for keeping track of a limited amount of information about file originators, namely their onion id's and the hashes of what files they have on offer.

In order to develop this model, we synthesized concepts we found useful from diverse areas of computing, including collaborative computing, graphics, the design of OSes and other large software architectures, the design and usability of secure client interfaces, and randomized algorithms.

During the next month, we plan to do much of the coding, given that much of the design sketching is now complete, and hopefully to open source this project and accept contributions in terms of both code and architectural suggestions from any interested parties. In the next year, we hope to have this project at a state of completion suitable for alpha release to the general public. We'd like to thank the reader for their patience, and everyone who inadvertently supplied the author with ideas over the past year and a half (or just sat still long enough for explanation to take place).

5. REFERENCES

[1] M. Alsabah, K. Bauer, T. Elahi, and I. Goldberg. The path less travelled: Overcoming tor's bottlenecks with traffic splitting. In *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS 2013)*, July 2013.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[3] D. Balfanz, G. Durfee, D. Smetters, and R. Grinter. In search of usable security: five lessons from the field. *Security Privacy, IEEE*, 2(5):19–24, Sept 2004.

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2. In *HTTPbis Working Group*, Fremont, CA, USA, July 30, 2014. IETF (Internet Engineering Task Force).

[5] T. Brewster. Swedish researchers uncover dirty tor exit relays. `http://www.techweekeurope.co.uk/workspace/malicious-tor-exit-relays-136828`.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. Rfc 4880: Openpgp message format. `https://tools.ietf.org/html/rfc4880`.

[7] S. Chakravarty, M. V. Barbera, G. Portokalidis, M. Polychronakis, and A. D. Keromytis. On the effectiveness of traffic analysis against anonymity networks using flow records. In *Proceedings of the 15th Passive and Active Measurements Conference (PAM '14)*, March 2014.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[9] B. Cohen. The bittorrent protocol specification. `http://www.bittorrent.org/beps/bep_0003.html`.

[10] N. Cubrilovic. Analyzing the fbi's explanation of how they located silk road. `https://www.nikcub.com/posts/analyzing-fbi-explanation-silk-road/`.

[11] C. Dewey. This debunked kickstarter project may be the biggest crowdfunding fail to date. *The Washington Post*, October 16 2014.

[12] R. Dingledine, N. Mathewson, and P. Syverson. Deploying low latency anonymity: Design challenges and social factors. `http://www.dtic.mil/dtic/tr/fulltext/u2/a527761.pdf`.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Rendezvous points and hidden services. `https://svn.torproject.org/svn/projects/design-paper/tor-design.html#sec:rendezvous`.

[14] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[15] R. Dingledine and S. J. Murdoch. Why tor is slow and what we're going to do about it. https://svn.torproject.org/svn/projects/roadmaps/2009-03-11-performance.pdf.

[16] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance: From theory to reality. In *Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2003.

[17] C. Egger, J. Schlumberger, C. Kruegel, and G. Vigna. Practical attacks against the i2p network. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2013)*, October 2013.

[18] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[19] A. Foundation. Apache couchdb. http://couchdb.apache.org/.

[20] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key/value store. In *OSDI*, 2010.

[21] J. Geddes, R. Jansen, and N. Hopper. How low can you go: Balancing performance with anonymity in tor. In *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS 2013)*, July 2013.

[22] A. Greenberg and L. Poitras. These are the emails snowden sent to first introduce his epic nsa leaks. http://www.wired.com/2014/10/snowdens-first-emails-to-poitras/.

[23] P. Gutmann and I. Grigg. Security usability. *IEEE Security and Privacy*, 3:56–58, 2005.

[24] S. Han, V. Liu, Q. Pu, S. Peter, T. E. Anderson, A. Krishnamurthy, and D. Wetherall. Expressive privacy control with pseudonyms. In D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, *SIGCOMM*, pages 291–302. ACM, 2013.

[25] F. House. 2014: Freedom on the net report. https://freedomhouse.org/report/freedom-net/freedom-net-2014#.VIHcFOrOpp9.

[26] U. N. International Community. The universal declaration of human rights. 1948.

[27] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The sniper attack: Anonymously deanonymizing and disabling the Tor network. In *Proceedings of the Network and Distributed Security Symposium - NDSS '14*. IEEE, February 2014.

[28] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users get routed: Traffic correlation on tor by realistic adversaries. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 337–348, New York, NY, USA, 2013. ACM.

[29] jrandom (Pseudonym). Invisible internet project (i2p) project overview. Design document, August 2003.

[30] N. Kerris and T. Muller. Apple media advisory: update to celebrity photo investigation. http://www.apple.com/pr/library/2014/09/02Apple-Media-Advisory.html.

[31] E. Kovacs. Operation against tor dark markets raises security concernts. http://www.securityweek.com/operation-against-tor-dark-markets-raises-secu

[32] M. F. Lee. Onionshare.

[33] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[34] Z. Ling, L. Junzhou, K. Wu, W. Yu, and Z. Fu. Torward: Discovery of malicious traffic over tor. In *Proceedings of Infocom: IEEE Conference on Computer COmmunications*. IEEE, 2014.

[35] A. Loewenstern and A. Norberg. Bittorrent's distributed sloppy hash table specification. http://www.bittorrent.org/beps/bep_0005.html.

[36] D. Macmillan and D. Yadron. Dropbox blames security breach on password reuse. *Wall Street Journal: Digits Blog*.

[37] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[38] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[39] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[40] G. Norcie, J. Blythe, K. Caine, and L. J. Camp. Why johnny can't blow the whistle: Identifying and reducing usability issues in anonymity systems. In *Proceedings of the 2014 Workshop on Usable Security (USEC)*, February 2014.

[41] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New

York, NY, USA, 2011. ACM.

[42] T. T. Project. https://blog.torproject.org/blog/call-arms-helping-internet-services-accept-anonymous-users.

[43] T. T. Project. Tor metrics portal. metrics.torproject.org.

[44] E. S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[45] M. Ripeanu. A peer-to-peer architecture case study: The gnutella network. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 99–, Washington, DC, USA, 2001. IEEE Computer Society.

[46] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[47] B. Schneier. More crypto wars ii. https://www.schneier.com/blog/archives/2014/10/more_crypto_war.html.

[48] R. W. Stevens and S. A. Rago. *Advanced Programming in the UNIX(R) Environment (2Nd Edition)*. Addison-Wesley Professional, 2005.

[49] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, Aug. 2001.

[50] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with Project Voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association.

[51] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, Washington, DC, USA, 1997. IEEE Computer Society.

[52] Tails. Tails: the amnesiac incognito live system. https://tails.boum.org/.

[53] J. P. Timpanaro, I. Chrisment, and O. Festor. A bird's eye view on the i2p anonymous file-sharing environment. In *Proceedings of the 6th International Conference on Network and System Security*, Wu Yi Shan, China, November 2012.

[54] Tor. Tor project: Faq. https://www.torproject.org/docs/faq.html.en#FileSharing.

[55] uTorrent. Elegant, efficient torrent downloading. www.utorrent.com.

[56] A. Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[57] Z. Wilcox-O'hearn. Freenet compared to tahoe-lafs. https://emu.freenetproject.org/pipermail/tech/2012-March/003093.html.

[58] Z. Wilcox-O'Hearn and B. Warner. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 21–26, New York, NY, USA, 2008. ACM.