A Secure, Peer-to-Peer File Locker System

Arpad Kovacs, Alex Grover

December 4, 2014

Abstract

We have built a secure, distributed, scalable, peer-to-peer file-locker system which enables users to mirror files of their choice in a decentralized "cloud" of individually untrusted commodity consumer computers, while still ensuring the integrity of replicated files by utilizing keyed-hash message authentication codes. Our system is cross-platform (it can be ported to any device which is capable of running the Java virtual machine) and utilizes standard TCP/IP networking to promote adoption and usage across the wider internet. Participants in the system are registered in a distributed hash table backed by the Chord algorithm, therefore saves and lookups in the system scale logarithmically in proportion to the number of participating clients.

1 Introduction

File, photo, and video-sharing sites constitute an overwhelming majority of traffic on the modern Internet. The most popular such services are centrally-hosted (eg: Google, Youtube, Facebook, Flickr, Dropbox, RapidShare) and are therefore susceptible to censorship and service outages while also forcing users to give up control of their content to commercial entities whose financial incentives might not be aligned with the user's interests. In addition, the centralized model is also costly and inefficient, since each website owner must construct/colocate multiple large datacenters or purchase hosting and content-distribution-network services to maintain availability and ensure acceptable latency for the end-user.

An alternative distribution model has emerged in the form of peer-to-peer technologies such as BitTorrent, which have managed to attract millions of users worldwide[1]. P2P systems like BitTorrent are advantageous because they allow content to be replicated on end users' machines rather than at a central host, reducing the cost and barrier to entry of sharing large files. However, there are still numerous design flaws in this system:

- BitTorrent depends on a central tracker server to initiate client downloads. This introduces a single point of failure into the system.
- The system design is optimized for distribution of sets of large files, rather than many smaller files. Peers must rediscover each other through the tracker after each fileset download, which means it can take some time after a user initiates the download before the optimal download/upload rate is achieved.
- Users generally only seed files that they are personally interested in downloading or sharing. Consequently, unpopular or infrequently-accessed files may no longer be seeded and therefore become essentially unavailable in the system.

We set out to improve upon this model by creating a distributed, secure peer-to-peer file-locker system that is designed with some of these shortcomings in mind. As such, the fundamental goals of our system design include the following:

- Completely decentralized architecture
- High availability of files, including tolerance of the failure or loss of any node
- Maintain file integrity even in the presence of malicious peers
- Low overhead to find and retrieve remote files

We also build off of strengths of the BitTorrent model, especially in terms of security and content integrity. The system uses cryptographic checksums to ensure content integrity for end users. The security model is discussed in detail in section 2.

2 Security Threat Model and Distributed System Failure Modes

We assume that the computer belonging to a legitimate user of the system (the "user") is secure and therefore serves as a trusted computing base. The uploaded file is replicated to r distinct replica nodes (of which f may be untrusted), and the original file may subsequently be deleted from the initial trusted node (aka the user might accidentally delete the file on their computer). Provided that the uploading user can recall the hash/signature of the file and secret key, the goal of the system is to enable the user to recover the file from the r replicas and ensure that it has not been tampered with. Our threat model consists of an adversary who can select any f peer nodes in the Chord ring and cause them to fail in a Byzantine manner. In particular, the adversary may perform arbitrary operations on the local filesystem of those f Chord nodes (where the replicated data is stored), and can send arbitrary requests over the network originating from the f selected Chord nodes. In our model, the adversary cannot impersonate an existing functioning or failed Chord node other than the f nodes in its possession via IP address spoofing or similar techniques. We believe this assumption is permissible due to the existence of protocols such as IPSec which can be implemented to ensure data origin authentication and protection against replay attacks. We also presume that it is possible to create a secure channel over the network using protocols such as SSL/TLS, therefore we do not deal with man-in-the-middle attacks due to the complexity of implementing secure key exchange and HTTPS which is outside the scope of our project.

The earlier Practical Byzantine Fault Tolerance paper [2] deals with byzantine failure models in asynchronous distributed systems. It implements a fault-tolerant NFS system via deterministic, replicated state machines. Clients request operations through the primary, which performs a 3 phase commit protocol. However, its usage of multicast to contact all replicas is not suitable for peer-to-peer systems. The paper proves that we require at least $r \geq 3f + 1$ replicas in an asynchronous system to guarantee safety and liveness when f of the replicas may be faulty. This is because f of the replicas might not respond, and of the remaining responding replicas, the correctly-functioning ones must outnumber the f faulty ones, eg: $n - 2f > f$.

We selected Chord [3] as the distributed hash table algorithm due to its symmetric and cross-cutting finger tables which have the potential to route around local disruption. Specifically, at equilibrium in a large Chord ring with evenly-distributed keyspace, node n's finger table can contain references to m other distinct nodes in the ring which are at locations $n + 2^{i-1} \pmod{2^m}$ indices away for $1 \leq i \leq m$. We were inspired by the partitioning and replication scheme described in the Dynamo system [4] to implement a successor list for each node, onto which we place r replicas for all objects whose ids hash into that node's keyspace.

There is a shortcoming in this scheme in that if nodes repeatedly join and leave the Chord ring, then the replicas will be moved and could accumulate on adversarial nodes. To solve this problem would require tracking the lineage information of the replicas to ensure that at most f of the replicas could have been tampered with by malicious nodes. We leave this problem as an area of future research; for now we assume that the adversary will have difficulty the Chord ring will be relatively static, with few nodes joining or leaving that would cause the replica to be moved for the duration of a replica's lifetime.

We also considered the Kademlia DHT algorithm [5], however we were initially concerned that a clever adversary could select node identifiers that are "near" a targeted node in terms of XOR distance. By surrounding the target node and intercepting any messages to it, (eg: filling up the K-tables pointing towards that node), we feared that adversarial nodes could cut off the node from the rest of the Kademlia tree. Later, while performing additional research on how to secure Chord RPC lookups, we stumbled upon S/Kademlia [6] which adds a secure key-based routing protocol that can mitigate this issue; however by that time we had already implemented a Chord-derived protocol.

What are the ways in which an adversary can disrupt the Chord ring? In the unlucky case when the first node contacted by the uploader user's trusted during the join operation is owned by the adversary, the adversary may construct its own Chord ring consisting exclusively (or a majority) of the f malicious nodes and induce the trusted node to join this fake ring, rather than the full legitimate Chord ring. If this attack were to succeed, then the $r \geq 3f + 1$ condition would not

hold, and safety/security/liveness are no longer guaranteed. To mitigate this attack, we require the uploader to initiate the join at a "trusted" node (eg: a friend verified through offline means), and contact at least $3f + 1$ distinct nodes during the join operation, otherwise we abort the join. After the join operation is completed, the trusted uploader's node should also query each finger's predecessor for the successor nodes to sanity-check the finger table (note: implementation of this feature is a work in progress). The $3f+1$ threshold presents a problem during bootstrapping of the initial chord ring, but we assume that for small rings the operators of the chord nodes can verify each other via offline channels and establish a web-of-trust which overrides the safeguard; only at larger scales does the threat of unknown infiltrators/adversaries become a realistic problem.

In the more general case, malicious nodes may lie when asked to execute the `find_successor` remote procedure call, by claiming to own a key which is outside of their jurisdiction, or pointing to an invalid successor. This poses a problem during failure recovery, since according to the Chord paper, in the worst case `find_predecessor` only makes progress by traversing the successor pointers. To mitigate this problem, we have implemented a list of successors rather than a single successor. We set the length of the successor list to $r = 3f + 1$, therefore we are assured that the valid successors will outnumber malicious and failed successors.

For our implementation, we have focused primarily on solving the replica placement and file integrity issues which are discussed in additional detail in the following section. We have made some progress in securing the Chord ring, but the challenge of implementing a secure version of RMI has been daunting. Therefore we relax the requirement that the adversary can send arbitrary messages over the network for now, and only guarantee that any local filesystem changes that malicious nodes make will not affect the integrity of the replicated files.

3 Design and Implementation

The system can be downloaded by participants as a standalone executable (a Java jar file which includes all of its dependencies), a single instance of which should be run on each participant's computer (hereafter referred to as a "node"). The system consists of two core components, the trusted "Shard" and the untrusted "ChordNode". This dichotomy provides a logical separation between local requests which can be fulfilled by the user's computer and remote interactions which require querying or modifying the distributed hash table and may be tampered with by malicious peers.

The Shard is the user's interface to his/her local compute and storage resources. It performs all local filesystem and cryptographic operations, and also embeds a self-contained Jetty webserver which exposes HTTP/REST endpoints for uploading and retrieving files from the local machine. The user can specify the algorithm used to generate identifiers for objects added to the Chord ring (currently defaults to secure HMAC-SHA256, but insecure/unkeyed SHA256 identifier is also supported to provide a performance baseline), as well as the trusted Chord ring node to initiate the join from at the time of boot via command line parameters. For ease-of-use, the user can open a simple HTML page using any standards-compliant web browser to upload and access his/her files.

As discussed in the Security Threat Model section, the user's computer is presumed to be part of the trusted compute base (the adversary cannot tamper with the local filesystem or compute capabilities of the Shard). To ensure the integrity of files, we load a 256-bit secret key in hexadecimal form from a user-provided file, or if it does not exist then we generate a new secret key (a cryptographically-random 256-bit value) at the time the Shard is first started, and write it locally to a private persistent file so it is available in the future. This key is subsequently used to generate HMAC-SHA256 hashes for all uploaded files; these hashes verify the data integrity and authenticity of the message on retrieval.

When a user uploads a file, the Shard begins computing the HMAC-SHA256 (the user's checksum) and SHA256 (the Chord identifier key) of the file as the data is streaming in. It writes the inputStream to a temporary file in the temp/ directory which is given a randomly-generated UUID filename to avoid collisions when multiple files are uploaded simultaneously. Once the file upload has completed successfully, the shard performs an atomic move to the data/ directory and renames the file to a hexadecimal representation of its computed HMAC-SHA256. This is

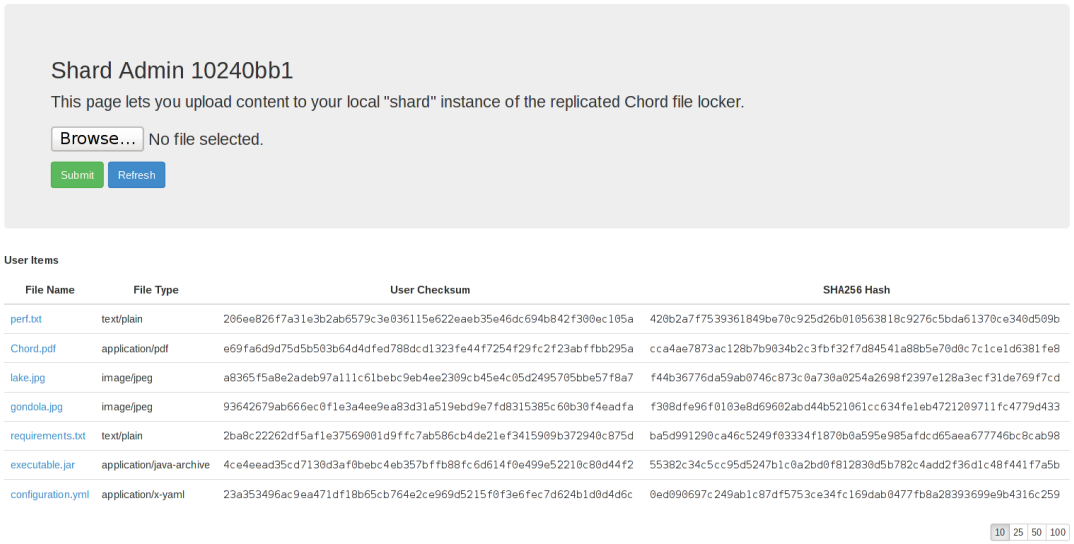

Figure 1: Screenshot of Shard administration web interface

done to ensure that only valid files are registered in the system; if the upload was interrupted or unsuccessful, then the thread will abort and the truncated or incomplete file will remain in the temp/ directory until it is garbage-collected in the future.

The user must use the full HMAC-SHA256 hash to lookup the file in the system, since that is the unique identifier (and in the case of encrypted files, the signature) of the file in the system. This is a deviation from the original Chord paper - which utilized SHA-1 as the hash for both node IP addresses and object identifiers - since SHA-1 has been cryptographically broken since 2005 [7] and is not recommended for usage in digital signatures due to the risk of adversaries engineering hash collisions [8].

We wish to avoid hash collisions between uploaded files, but since nodes are uniquely determined by their unspoofable IP addresses and cannot be impersonated/collided (see Security section), we optimize by only using the top 32 bits of the file's SHA-256 hash for the purposes of determining placement and replication of the file in the Chord ring. We assume that fewer than 4 billion nodes will participate in the ring; and therefore do not send the full hash when executing the Chord protocol to reduce network payload size and latency.

The ChordNode is our implementation of the Chord distributed hashtable protocol. The distributed hashtable essentially maps from each node's 32-bit integer identifier to the corresponding IPv4 address and port, and also serves to determine the interval of the uploaded object keys that each node is responsible for. We implement the ChordRing RPC calls using Java RMI. ChordNode is intended to be completely invisible to the end-user and is only used for determining the placement/location of replicated files (basically, mapping from 32-bit keyspace identifier to IP address/port) on other nodes and ensuring that the ring remains unbroken/balanced when nodes join or leave.

To ensure that the keyspace is relatively evenly distributed even when only a few sequentially-numbered nodes are participating in the system (eg: 192.168.1.1, 192.168.1.2, ...), we compute a hash of the IPv4 address using a modified version of Knuth's multiplicative method $inthash(input) = input \times 2654435761 \mod 2^{32}$ as described in TAOCP Section 6.4 [9] rather than using the raw 32-bit IPv4 address value. I selected this hash function because it performs one-to-one remapping from the set of positive 32-bit integers to positive 32-bit integers ($inthash : \mathbb{Z}^+_{\text{32-bit}} \rightarrow \mathbb{Z}^+_{\text{32-bit}}$) since 2654435761 and 2^{32} have no common factors. Therefore we are guaranteed that there will not be any collisions. Note that since Java does not support an unsigned integer type, we had to convert the input into long values, add Integer.MIN_VALUE as an offset, perform the multiplicative factors operation, and then subtract Integer.MIN_VALUE to compute the signed integer inthash identifier.

Most of our implementation follows the description in the original Chord paper: we maintain

references to the predecessor node, as well as a finger table to ensure $\mathcal{O}(\log n)$ lookups for any key in the system. Each node in the system is identified by its 32-bit inthash (as discussed above), and is responsible for all objects in its key range, as well as objects belonging to the nodes whose fingers point at it if that node is one of the r longest fingers and is therefore assigned a replica.

We implemented the self-stabilizing version of the Chord algorithm described in the paper. To optimize the system for many concurrent joins and leaves, this method foregoes the mandatory messages to notify the rest of the ring upon nodes leaving. Instead, it only updates its predecessors successor pointer and updates a random member of each node's finger table at a set interval. When a node leaves the ring, all RMI calls to its location will fail and the node tries the lookup on the previous node instead. In the worst case, this will mean a linear walk of the ring to find the file specified but still guarantees correctness for key lookups. However, finger table entries are updated soon after nodes leaving and therefore lookups will still take $\mathcal{O}(\log n)$ time on average. This optimization made sense given the nature of our application and the fact that nodes can fail or leave the ring at any point in time.

The Chord paper does not specify how to perform service discovery, therefore at the present the user must provide his/her local shard with the IP address (and optionally a non-default port) to an existing node in the Chord ring when starting their local instance. If no argument is specified, then a new Chord ring is created; otherwise if the user specifies an IP address then query the specified node and join the existing chord ring that it is a member of (eg: by performing ChordNode.join operation via RPC). Details on join security are provided in the Security section.

4 Evaluation

To test performance, we uploaded an assortment of different file types (including plaintext files, source code, documents, photos, and other media, ranging from 16 bytes to 1.5MB in size), and observed the performance overhead of the features we have implemented. Instrumentation was provided by the DropWizard metrics-core library which we incorporated into the Shard's embedded Jetty server. To ensure consistency, we executed the upload 100 times for each file, and measured total latency from the time the request was received by the Shard until the response was completed by the server. We have plotted the 50th (median) percentile latencies in seconds, as a function of the size of the file and replication factor below. We observe that the overhead of symmetric-key HMAC encryption is surprisingly negligible, incurring only about 10% overhead.

We performed evaluation of the final system on one physical machine running 12 instances of the system within virtual machines, for a total of a 12-node cluster. While we realize that this benchmark is not reflective of the expected usage patterns of our system, we found it instrumental in comparing the latencies of different replication factors relative to our baseline of reading/writing only to the local machine. We observe that local reads and writes appear to be very fast, while network operations appear to take longer time. We hypothesize that serializing byte-arrays over RMI may be a bottleneck, and also believe that the operating system might be performing some form of file caching/paging. This could also explain why reads appear to be faster than writes in general.

The write performance scales nearly linearly with the number of replicas, since the file byte array must be copied to each replica and written to disk. Read performance is dominated by the number of file transfers that must be made back to the trusted user's node. When the file is found on the first replica ("Replica1 Read"), there is a single transfer. If the file is not available on the first replica ("Replica2 Read"), then a we must lookup the successor and read from that replica. When the read from the first replica is corrupted ("Replica1 Corrupt"), then we are required to follow the successor and perform a second read from the second replica. The roughly doubled access time compared to "Replica1 Read" is in line with out expectations.

References

[1] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis," in *Peer-to-Peer Systems IV*, pp. 205–216, Springer, 2005.

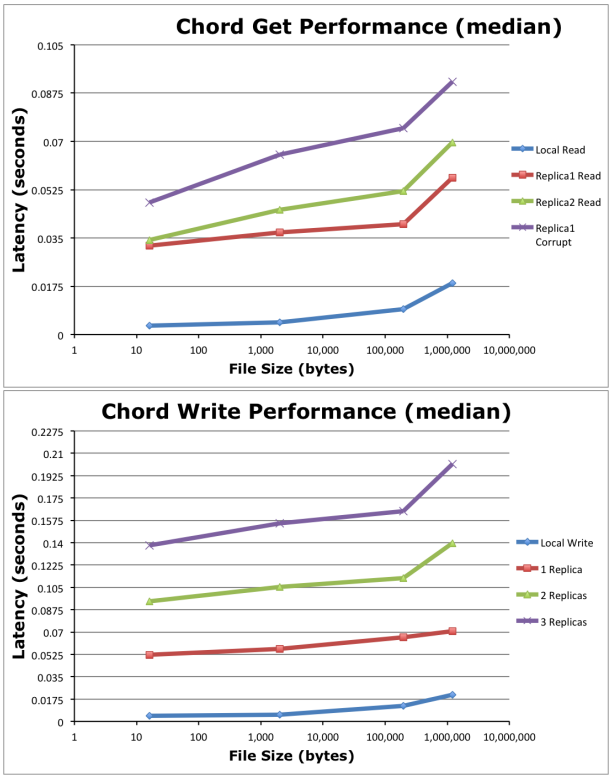

Figure 2: Median read and insert latencies as a function of filesize

[2] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 173–186, USENIX Association, 1999.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[5] 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, (London, UK, UK), pp. 53–65, Springer-Verlag, 2002.

[6] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing.," in *ICPADS*, pp. 1–8, IEEE Computer Society, 2007.

[7] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *In Proceedings of Crypto*, pp. 17–36, Springer, 2005.

[8] B. Schneier, "Cryptanalysis of sha-1." `https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html`, 2005.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.