6.S974 Project Report - Ouranos

Nicolaas Kaashoek

October 17, 2020

1 Introduction

Ouranos is a decentralized file-sharing service similar to existing systems like Keybase and IPFS. Ouranos attempts to address the issue of freshness within such a system, and the different approaches one might take to guarantee it. Ouranos uses the concept of a "freshness ticket" to decouple the freshness check and the file-sharing completely, making the check more reliable. The current implementation of Ouranos is written entirely in Go, but could be easily extended to support other languages and integration with other services such as blockstack.

When designing Ouranos, the question of freshness as well as the bootstrapping protocol ended up being the most challenging portions of the implementation, and revealed much about the difficulty in designing decentralized systems.

2 Overview

Many of the systems discussed during the course, such as IPFS[3], fail to address file freshness. Without any kind of freshness guarantees, users can't be sure what versions of a file they are actually reading. Ouranos is an attempt to create a system that provides both a high degree of freshness and functions when some subset of the users' machines are offline. Ouranos ensures freshness on the order of seconds, rather than hours (the guarantee achieved by a system using the blockchain).

Allowing for file sharing even when the party sharing the file is offline greatly improves the usability of Ouranos. Having to have both users active at the same time almost defeats the purpose of having a system for file sharing, as the users might as well exchange the file out of band, so Ouranos ensures that as long as a subset of machines in the system remain on, it can still function as normal.

Another major concern in decentralized systems is security, and how the system behaves in the presence of malicious users. While Ouranos doesn't focus on security, it signs and encrypts all sensitive data. While this lowers the potential impact of Sibyl attacks, Ouranos is still vulnerable to Denial of Service attacks.

```
type OuranosFile struct {
    Name    string
    Version int
    Creator string
    Content []byte
}
```

Figure 1: Ouranos Files and their associated metadata

3 System Design

This section describes the design of Ouranos and some of the common operations users perform within the system.

3.1 Names, Keys and Files

Users within Ouranos are machines. Ouranos does not allow users to migrate from one machine to another, so each machine within the system is a separate user. All of these users are associated with some name, as well as with a private key. This key is either generated by Ouranos or supplied by the user. These keys are standard RSA private keys.

Each user maintains some set of files, which are either private or shared with a subset of the other users. Each of these files is an array of bytes, so any format should be share-able. Ouranos was evaluated on simple text files and images, but it should be able to handle videos as well, albeit with a significant performance hit.

Each file is also associated with a file encryption key used to encrypt it when the file is distributed to other users. These file keys also facilitate file sharing. Files are also associated with a version number and a creator. Important metadata such as the version number and creator is stored within the file itself, as seen in Figure 1.

3.2 Bootstrapping

In order for Ouranos to work, each user must be able to contact some other users within the system. Many production systems rely on DHTs such as S-Kademlia[2] to do so, and Ouranos can easily be modified to use these as well. Currently, it builds up the equivalent of a routing table for each node so that they know about some subset of the network. The size of this subset is configurable, with larger sizes making Ouranos more tolerant to faulty machines.

When a user first accesses Ouranos, they must provide the address of some peer that is already within the system. Users would find out about Ouranos through other users anyways, so this should be natural. Were Ouranos to grow large enough, a tracker file as used in bittorrent could be used instead. The user begins by contacting this peer, who shares with it information about its peers. The user can then contact each of these peers in turn

```
type BootstrapMessage struct {
    Location  string
    Name      string
    PublicKey string
}
```

```
type FileMessage struct {
    Name    string
    Version int
    Content []byte
}
```

Figure 2: Message formats used by Ouranos

to build up some idea of who exists in the network.

When exchanging information with a peer, the user and the peer each provide the other party with their location on the network, their public key, and some human readable name. The exact message format can be found in figure 2. Configuration information is saved to prevent Ouranos from repeating this process every time it starts.

3.3 Contacting Other Users

Most of the operations in Ouranos require the user to make contact with some other user in the system. Because the bootstrapping process doesn't give the user information about all peers, she may not know about the peer she wishes to contact. In a situation where User A wants to contact User B, A will reach out to all her peers asking about B's location. If none of these peers know about B either, they will reach out to their peers and so on and so forth until the information about B's location is received. This is then propagated back to A, who can then reach out to B.

3.4 Creating a file

A user begins the process of adding a file to Ouranos by first creating a new file locally. After the file has been created, Ouranos generates an AES encryption key for this file, and stores that key in the users local storage as well. Ouranos then encrypts the file the generated key, and then contacts the peers of the user, letting them know that the user has added a file to the system. The file is initialized with a version number of 1. This encrypted file is then signed with the users public key, preventing malicious users from faking the file.

At this point the user sends the file to the peers she knows about. This leaves the user with an un-encrypted version of the file in local storage, while her peers have an encrypted version of the same file in their local storage. Keeping the un-encrypted file around isn't necessary, but it makes the process of editing the file much less painful for the user. The number of peers who receive the file can be manually configured.

These steps, as well as those described in 3.5 and 3.6 are summarized in Figure 3.

3.5 Sharing a file

In Ouranos the idea of sharing a file has little to do with the file itself. Instead, users share the keys used to encrypt files already present in the system. This way a user doesn't need

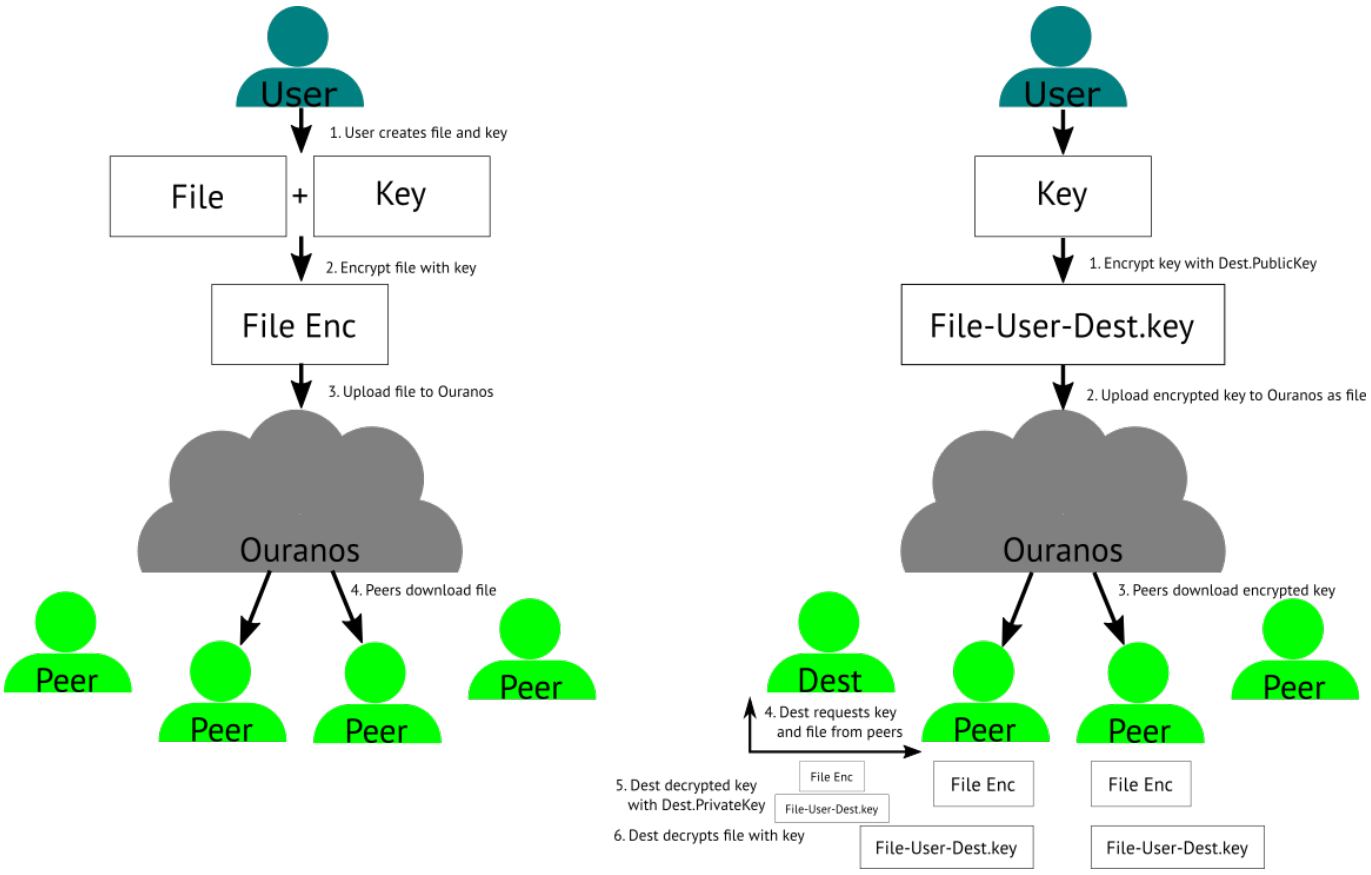

Figure 3: File Creation and Sharing

to continuously re-encrypt a file when they want to share it with a new peer. Instead, the creator of a file takes the file key generated when they initially created the file and encrypts it with the public key of the peer they want to share it with. If the user's instance of Ouranos doesn't already know about the public key in question, she can simply follow the process described in 3.3 to find the key.

This key is then encrypted with the peer's public key using 256 bit RSA-OAEP encryption. Ouranos assigns the encrypted key a name with the format: `FileName-Creator-Peer.key`. This file is then placed into Ouranos as any other file would be, ensuring that even if the peer is offline, she will be able to find the file when she comes online again.

If the user needs to share the same file again, she can re-encrypt the file key with a different public key and put that key file into the system for the new peer to access. This way, adding another person who can read the file doesn't require any overhead for the users the file is already shared with.

3.6 Reading a file

For a user to read one of its shared files, she needs to collect the encrypted file and the encrypted file key. To do so, the user asks Ouranos for each of these files. When the files are returned to her, she first decrypts the file key using her own private key. Then, she takes the key and uses it to decrypt the file, at which point she can read the file.

3.7 Editing a file

Users can edit a file using whatever programs they want. Once they are done editing the file, they notify Ouranos, which increases the version number of the file and uploads it to a subset of the other peers in the system and distributes a new freshness ticket (see 3.9).

3.8 Signatures and Integrity

Security wasn't a primary concern when building Ouranos. However, it is important to have some minimal guarantees, so Ouranos allows clients to sign the files and keys they create. This ensures that machines can at least check whether they are getting the files they expect out of Ouranos. Ouranos doesn't need a signature key per file as each file is only ever written to by a single user.

3.9 Freshness

Each file in Ouranos has an associated version number contained within the file itself. In addition to this version number, Ouranos maintains a separate "freshness ticket" for that file (figure 4). When a user wants to read a file shared with her, she retrieves both the file and the freshness ticket from Ouranos. The freshness ticket is signed with the key of the creator of the file, and is therefore self-verifying. Users can find the freshness tickets by looking for

```
type FreshTicket struct {
    Name      string
    Version   int
    Creator   string
    Timestamp time.Time
}
```

Figure 4: Freshness Ticket Format

them under the same name as the file they refer to. The tickets can be distributed in any way that allows all users to access them; Ouranos supports the following three schemes.

Gossip The simplest method of storing the freshness ticket is to put it in Ouranos. In this scheme, each user contacts peers in a widening net to collect tickets for the file. She then compares all these tickets to see what the latest known version of the file present in the system is. The more peers a user contacts, the more certain about the freshness of the file the user can be.

Centralization The second scheme uses a centralized storage provider to store the freshness tickets. This comes at the advantage of lower network traffic, lower latency, and better freshness guarantees. However, this scheme introduces a point of centralization to the system. Having the tickets be self-verifying greatly mitigates the risks this introduces.

Combination Combining both schemes enables peers to act as "auditors" for the centralized storage provider, similar to the way CONIKS[4] operates. This increases network load when acquiring or distributing tickets, but provides stronger trust guarantees than the centralization scheme on its own. In this case, every so often when a user fetches a file from Ouranos, it performs the same check as in the gossip scheme. It compares the result of this check to the version fetched from central storage, and if they don't match, checks with the central authority again, and if they still don't match, the user notifies all peers to exclusively use the gossip protocol.

3.10 Evaluating the Gossip Protocol

Consider an Ouranos deployment where users distribute freshness tickets to P peers in an N peer system, with some round trip time for communication between nodes. Assuming no malicious users, then for a user to guarantee that the user has talked to at least one peer with the latest freshness ticket, they must contact $N-(P+1)$ peers. At minimum, where the user already knows about $N-(P+1)$ peers, this would take 1 round trip to accomplish, as each request can be sent in parallel. However, given that Ouranos would benefit from having a small P and a large N, it is unreasonable to assume that users would know about that many peers.

Were Ouranos to use a DHT to back the routing, it would need to reach out to $\log n$ peers to find a specific peer. In the worst case where a user only knows about 1 other peer, it needs to find out about $N - (P + 1) - 1$ more peers to contact for the freshness ticket, and each of those lookups takes $\log n$ round trips. After all these peers have been found, the user needs to reach out to all of them to request the freshness ticket. This takes an additional round trip. Executing all these requests in parallel, it would take a total of $\log n + 1 = O(\log n)$ round trips for the user to be sure about the freshness of a file. With the combination scheme, this process doesn't need to happen on every file fetch, but every F fetches.

4 Implementation

Ouranos is implemented in approximately 4000 lines of Go. It consists of a single Daemon that spawns a client and server instance. Testing was done with 10 machines by having the servers each listen on different ports of localhost. Ouranos uses gRPC for communication between peers, and the standard go crypto libraries to sign and encrypt files. File metadata and contents are encoded to the file using the Go's gob encoder/decoder.

Users interface with Ouranos through a simple shell run by the Go Daemon, and execute commands like make, share, and update to control the system. Testing was performed in a variety of situations, with varying numbers of on/offline machines, and some simple malicious clients. Ouranos' codebase is modular, and could easily be integrated with existing technologies like Kademlia and BlockStack.

The centralized storage protocol was implemented using Microsoft's Azure Storage as a backend, but could be replaced with other commonly used cloud storage services such as Amazon S3.

5 Discussion

Ouranos is easy to use and intuitive, if unpolished. Its construction makes it straightforward to replace parts of the system with existing solutions. One particularly useful example would be using Blockstack to hande naming, as currently Ouranos has no way of ensuring that names are unique.

From a security standpoint, having users sign all their files and freshness tickets ensures that at the very least they can be confident about what Ouranos returns to them, even in the face of malicious clients. However, Ouranos is still vulnerable to Sibyl attacks, as they can easily perform denial of service on any peer in the network. Such risks could be reduced through the use of a system like Blockstack[1] that makes it harder to obtain names for many machines at a time.

One area where Ouranos needs further development is key management. Currently users can register with a single key, but have no way of revoking keys, recovering files in the case of a lost key, or migrating to a different machine. If keys were shared between machines, it would be straightforward to allow users to use the same key in multiple places. Systems like Keybase allow each user to have a key per machine, which is impossible in Ouranos' current

state. Key revocation and recovery are also impossible in the current system, and having hardware stolen compromises a users Ouranos instance. Adding a login service could take care of the latter issue, as keys could be encrypted with a user's password. Further thought is needed to deal with revocation/key change.

Another area for future work is revoking access to a file. Currently, a user needs to generate a new file key, re-encrypt the file, and then distribute both the new file and the new key to the appropriate peers. This process creates significant overhead if access control is frequently changing, and may need to be reconsidered.

Ouranos accomplishes the goals it set out to do. It functions well even if a good portion of the users in the system go offline, and is able to provide strong freshness properties using tickets, which are distributed in three possible schemes. Decoupling the ticket mechanism from the actual file makes it flexible, as the tickets remain small regardless of the size of the corresponding file and can therefore be easily distributed in different ways.

6 Acknowledgements

Thanks to Prof. Morris for providing feedback on Ouranos' design.

References

[1] Muneeb Ali et al. "Blockstack Technical Whitepaper". 2017. URL: https://blockstack.org/whitepaper.pdf.

[2] Ingmar Baumgart and Sebastian Mies. "S/kademlia: A practicable approach towards secure key-based routing". In: *In Procs of the Int'l Conference on Parallel and Distributed Systems*. 2007.

[3] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)". 2017. URL: https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf.

[4] Marcela S Melara et al. "{CONIKS}: Bringing Key Transparency to End Users". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. Washington, D.C.: {USENIX} Association, 2015, pp. 383–398. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.