Woland: storage is all you need

Abstract

This paper introduces a simple approach to distributed information across peers. Unlike a traditional peer-to-peer network, exchange is on public storage (servers or cloud) and not on the peers themselves. This approach is more scalable and more secure than traditional peer-to-peer networks. Data privacy is ensured by encrypting the data before it is stored on the public storage. The encryption key is shared between the peers using a key-value file on the storage. The key is the identifier of the peer and the value is the key encrypted with the public key of the peer.

Introduction

Modern cloud technology offers scalability and high availability without the complexity of a private setup and maintainance. However, the applications deployed on the cloud are totally transparent to the cloud vendor and the application's owner loses control over the data and the application itself. While peer-to-peer networks offer a solution to this problem, they are not scalable, they offer low availability and safeentially high latency.

Key concepts

The network is composed of users and storage. The peers are the nodes of the network and the storage is a service (servers or cloud) where the data is stored in a hierarchical structure that resambles a file system. The storage is named *safe* to recall the concept of a closed and protected container that holds information. Each item within the safe consists of a binary sequence of bytes, essentially forming a file.

Peers are qualified by an elliptic encryption key, that can be used both for encryption and signing.
A safe is protected by a symmetric encryption key named *primary key*. Files in a safe are encrypted with a key derived from the primary key and only peers that know the primary key can access the files in the safe. Peers possess varying degrees of control over the safe, establishing a hierarchy of permissions. These permissions range from mere readers who can access data to administrators with the authority to add or remove both peers and administrators. The permissions for a peer are defined with bitwise flags and they include the following access:

- 1: read when the peer can only read the safe
- 2: add when the peer can add content to the safe
- 16: admin when the peer can add or remove other peers
- 32: superadmin when the peer can add or remove administrators

Data can only be appended to the safe, and explicit removal and modification are prohibited. This design ensures data immutability. However, due to space constraints, the safe can be configured to perform periodic housekeeping operations that remove older data to manage storage limitations.

A *safe* has some support data structures to allow the peers to get access without previously owning the key. The first one is a key-value data structure that maps the peer's id to the master key. Each value is encrypted with the elliptic key of the peer so to prevent other peers to read the encryption key. The key-value file is named *keystore* to recall the concept of a key-value store. The map is signed by the peer making the changes, allowing other peers to verify that the modification was authorized for that specific peer.

The second one is a data structure that lists changes in membership to the safe. The data structure is named *changelog* to recall the concept of a log of changes. Only administrators can change the membership and the changes are signed by the administrator who made the change. A change signed by a peer without administration capabilities is ignored. Each change has a timestamp and changes are validated in chronological order. For instance a peer cannot add another peer if it is not granted earlier the required level of control. The first record in the *changelog* must be created by the safe creator, whose public key is provided to the peer by email or other means. While the *changelog* is used mostly for membership changes, it can be used for other changes like housekeeping operations.

The third one is metadata records, one for each file in the safe. Since files are organized in hierarchical structure, the metadata records are organized in a hierarchical structure as well. The metadata records are named *metadata* to recall the concept of a metadata store. Each record contains a secondary encryption key that is used to encrypt the file content. The record is instead encrypted with the primary key of the safe. The use of different keys for the metadata and the file content allows to change the primary key without re-encrypting the file content. The metadata records are signed by the peer who created the file.

The last supporting data structure is a configuration with information about the safe and like the administrators list, it is protected by a signature of the creator. The configuration is named *manifest* to recall the concept of a manifest file.

Change algorithm

When peers are added to or removed from the safe, the process involves updating the various aforementioned data structures. The sequence of steps for this process is as follows:

1. The peer initiating the change creates a new entry in the changelog containing the updated membership details and signs the entry. This change is permitted only if the peer holds the required level of control. Prior records in the changelog can be referenced to verify the change's effectiveness.
2. For an addition, the peer appends the new peer's identifier and encrypted key to the keystore, then signs the updated keystore. This completes the process.
3. If the change involves removal, the peer generates a fresh keystore with a new key. The peer includes the identifiers and encrypted keys of all peers in the safe within this new keystore, signing it before writing it to the safe.
4. As the key has changed, all metadata records must be re-encrypted using the new key. The peer reads each metadata record, applies the new encryption, and writes the

updated metadata records to the safe. The old metadata records are then deleted. This operation is recursive due to the hierarchical structure of metadata records.
5. After re-encrypting all metadata records, the peer can delete the old keystore.

When modifications are applied to the safe, it necessitates that other peers synchronize their local information. At periodic intervals, each peer scans the changelog for any updates. Upon detecting a change, the peer undertakes the subsequent actions:

1. The peer reads the complete changelog and reevaluates the present composition of peers within the safe, along with their corresponding levels of control. Should the peer no longer be a part of the safe, synchronization halts due to an authorization error.
2. If alterations have occurred in the keystore, the peer reads and decrypts the entry associated with its own identifier. If the peer's identifier cannot be located, synchronization ceases due to an authorization error.
3. The peer designates the new encryption key as the primary key, which is then employed for ensuing operations.

In essence, this process guarantees the accurate propagation of peer membership changes throughout the safe's data structures, all the while upholding security and consistency through encryption, signing, and re-encryption protocols. The practical implementation is slightly more intricate, devised to optimize input/output (I/O) operations—particularly during scenarios involving simultaneous removal of multiple peers or when peers have previously read the changelog.

Replica algorithm

The safe can be replicated on multiple storage services. The replicas are identified by a URL and the peers can access the safe from any of the replicas. The replicas are kept in sync by a background process that periodically checks the consistency of the replicas. The process follows these steps:

1. Given n replicas, the process reads the change logs from each replica and merges them in a single list.
2. The process compares each replica with the merged list. If the replica is not in sync, the process updates the replica with the missing changes.
3. The process compares the data and metadata of each replica across the hierarchical structure. For each node in the structure, the process copies the data files that may be missing. Metadata records instead are merged and a new file is created with them. Other metadata files are deleted.

Implementation

The concepts described above are implemented in a library called woland with reference to the famous novel "The master and behemoth. The library is written in Go but can be used from any language that supports dynamic libraries. The library is available on GitHub at http://github.com/stregato/master/woland.

At the same GitHub location there is a sample application that uses the library to implement a simple chat application. The application is called Behemoth and it is available at http://github.com/stregato/master/behemoth

Access

Woland supports multiple storage services, including S3, WebDAV. SFTP and local file system. The storage service is abstracted by an interface and can be easily extended to support other services. The storage service is identified by the URL and this is one of the required information for access to the safe. Besides the URL, the path location in the storage is required. Furthermore the peer needs the public key of the safe creator to verify the *manifest* and the *changelog*. For redundancy reasons, the safe may be mirrored on different storage services and the peer would need the URL of all the mirrors.

For convenience the URLs of the safe, the path and the creator's key are encoded in a single string called *access* in base 64. Optionally the access string may be encrypted with eccliptic key of the peer who should receive the access. This is useful when the access string is sent by email or other unsecure means.

As described above the safe contains data structure that allow authorized access. The main configuration is a file *manifest.json* with the following fields:

- description: a description of the safe
- creator: the public key of the safe creator
- relaxed: when true all peers can add or remove peers from the safe
- changeWatchInterval: the interval for checking changes in the safe
- replicaInterval: the interval for checking consistency of the replicas
- signature: the signature of the manifest file (non including the signature field)

The changelog is structured using multiple files to prevent conflicts when peers add or remove others concurrently. When a peer introduces one or more change records, these are saved to a file with the .change extension. Though named differently, this file is essentially in JSON format and includes the following fields:

- type: the type of change (e.g. permission, replica)
- modTime: the time of the change
- change: a base64 encoded string with the change
- by: the public key of the peer who made the change
- signature: the signature of the change (built with the previous fields)

The file is validated by checking the signature of each record in chronological order. Not only the signature must be valid but also the peer must have the required level of control to make the change. For instance a peer cannot add another peer if it is not granted earlier the required level of control.

The data structure that describes the change depends on the type of change. For instance a change of permission is described by the following fields:

- userId: the public key of the peer whose permission is changed

- permission: the new permission for the peer

It's worth mentioning that changelog files are preserved indefinitely, with no deletions.

The *keystore* is instead a file with extension *.key* and it is a JSON file with the following fields:

- keyId: identifier of the encryption key
- keys: a map of public keys to encrypted keys
- delta: true when the file contains only the changes since the last update
- signature: the signature of the keystore (non including the signature field)

Similar to the *changelog*, there can be multiple *keystore* files. However, there's a key distinction: when a peer is removed, files holding old keys are deleted following the re-encryption of metadata files with the updated key.

Data organization

Data is organized in a hierarchical structure so that peers retrieve only content for the path they are interested in. For instance, if a peer is interested in files created in the last days, paths can include the date of creation. The actual path parts are hashed before being written to the store, while the original path is saved as metadata of the file. The drawback is that unlike a file system, path components (aka folders) cannot be listed without reading at least the metadata of one file in the folder. This resambles some storage services like S3, where folders are not real folders but just a prefix of the file name. Like for S3 folders do not exact as independent entities, but only when at least one file has a matching prefix.

The caching covers both metadata and data. When a peer enquiries files for a specific path, it retrieves the list of files for the provided path. The list is cached in a SQL table and subsequent enquiries for the same path are served from the cache unless new files are added to the path. More in details each folder has a sentinel file named .touch. The file is updated when a file is added to the folder. By checking the modification time of the .touch file, the peer can check if the cache is still valid. If the cache is not valid, the peer retrieves the list of files from the safe and updates the cache.

Similarly file content is cached on the local file system for a configurable amount of time. When a peer wants to retrieve a file, it first checks if the file is already in the cache. If so, the file is served from the cache. If not, the file is retrieved from the safe and stored in the cache.

About metadata

Metadata is stored in files with extension .meta. The name of the file is a generated ID similar to Twitter's snowflake. One file may have multiple records for multiple files. Each record contains the following fields:

- path: the path of the file including the name
- size: the size of the file
- modified: the last modification time of the file
- creator: the public key of the peer who created the file
- content type: the content type of the file (IANA media type)

- tags: a list of tags associated to the file
- zip: true if the file is compressed
- preview: file thumbnail (only for images)
- AES key: encryption key for the file content
- downloads: locations where the file has been downloaded locally
- cache: locations where the file has been cached locally

When a peer adds a new file to the safe, it creates a corresponding file for metadata. The metadata file is encrypted with the primary key of the safe and stored in the safe.

When the primary key in the safe changes (due to user removal), all the metadata files in a folder are merged and re-encrypted with the new key. Like for the keystore, the metadata files are written with an optimistic locking mechanism.

About data

The data is stored in files with extension .data. Like for the metadata, the name of the file is a generated ID similar to Twitter's snowflake. Optionally the file can be zipped during the write operation. In any case the file is encrypted with the secondary key kept in the metadata information.

The API

The library provides a simple API to access the safe. The API is composed of the following functions:

```
EncodeAccess(urls []string, path string, creator string) (string, error)
DecodeAccess(access string) (urls []string, path string, creator string, err error)

Create   (access string, options CreateOptions) (safe, error)
Open     (access string) (safe, error)
ListFiles     (safe, bucket string, options ListOptions) ([]File, error)
ListDirs (safe Safe, bucket string, options ListDirOptions) ([]string, error)
Put      (safe Safe, bucket, name string, r io.reader, options PutOptions) error
Get      (safe Safe, bucket, name string, w io.writer, options GetOptions) ([]byte, error)
Remove   (safe Safe, bucket, name string) error
SetUsers (safe Safe, users [string]Level) error
GetUsers (safe Safe) (users [string]Level, err error)
```

Sample application

A demonstration application highlights effective patterns to use Woland. The application is called Behemoth and similar to Microsoft Teams, it includes chat rooms and file sharing. The application is written in flutter and runs on Android, iOS, Windows, Mac and Linux. Both the binary and the source code are available at http://github.com/stregato/master/behemoth For Linux a snap package is available at http://snapcraft.io/behemoth

The application is built on the concept of covens. Each coven has a main room called *lounge* and users can create new rooms as needed. Each room has a chat and a file sharing area. The chat allows text messages and images. The file sharing area is a hierarchical structure of folders and files.

In the design of the application, the first step is to define how covens and rooms are represented. Behemoth uses a safe for each room. Initially only a room *lounge* is available. As an example let's assume a coven for people living in New York whose storage is available at https://d93838592ed8198272ca9113568.r2.cloudflarestorage.com/behemoth The first space would be *lounge* and the related safe would be named new_york_city/lounge. The access token would be built from:

- URL: https://d93838592ed8198272ca9113568.r2.cloudflarestorage.com/behemoth
- name: new_york_city/lounge
- creator: the public key of the person that creates the community

A single room is not enough and users can create new rooms as needed. Each new room is indeed a new safe, created by the user who creates the room. And the user that creates the room decides which other users can access.

Each safe has a simple structure with two root folders:

- chat: contains the messages, one file for each message
- content: contains the files shared in the chat room and it can have subfolders

Conclusion

This example application demonstrates the efficacy of leveraging unique cloud storage infrastructure to construct peer-to-peer applications. This approach maintains data ownership and control, ensuring a robust and user-centric experience.