Replication, History, and Grafting in the Ori Distributed File System

CMPE-273
FALL-2018

Team Name: Techno Shine
Team Members: Megha Nair (011551293)
Premal Dattatray Samale (012566333)
Vidhya Vijayakumar (011529284)

ABSTRACT:

In today's world users have tera bytes of storage on laptops, mobile phones and tablets with low cost. Yet user still pay massive premium to online cloud storage providers to do the Data Management (Availability, Accessibility, Durability, Usability). The goal of ORI is to get these benefits of the managed storage and implement it with the hardware we already own. The problems with managed storage are limited Bandwidth, Privacy, High storage cost than local storage and poor integration of Replication, Versioning, Sharing. Ori leverage trends such as Big disk, Fast local area network, mobile storage to do better. Ori is a new secure distributed file system. Ori is fully peer-to-peer, offering opportunistic synchronization between user devices in close proximity and ensuring that the file system is usable so long as a single replica remains.

ORI FEATURES:

- Replication
- History
- Snapshot
- Grafting
- ORI as a Version Control

REPLICATION:

ORI replication provides unidirectional synchronization between multiple repositories. When a repository is replicated initially an empty repository is created and then the data is mirrored. Replication fetches only the missing data which is considered equivalent to a git fetch rather than a git pull.

Replication Operations:

Several operations takes place while replicating a source directory to destination,

1. **Metadata** is maintained, which includes the hash of latest commit from the source and latest commit object.
2. **Object Reference Counts**, the computation of reference count is done incrementally as and when the objects are received.
3. And the **Index** is updated as and when the objects are stored in the destination.
4. Unidirectional synchronization happens with the orisync

Fetching Operations:

1. **Distributed Fetch:** Ori transfers data by avoiding low bandwidth and high latency links and uses nearby hosts as a cache for requested data. During replication operations Ori identifies the nearby instances using mDNS/Zeroconf and statically configured hosts. Ori contacts hosts in order of latency as an approximation of network quality. Before transferring objects from the original source, the algorithm attempts to use nearby hosts by asking for objects by hash.
2. **Background Fetch:** This enables operations such as replicate, pull, and graft, to complete the bulk data transfer in the background while making the file system immediately available. If a process accesses files or directories that are not yet locally replicated, it blocks and orifs moves the needed contents to the head of queue, ahead of other less important remaining background operations

SNAPSHOTS:

In Ori commit objects represent the Snapshot. Snapshots are needed so we can use Ori as a traditional file system, without having to run extra commands or explicitly take snapshots, but still get the benefits of versioning and being able to look at old files. Implicit snapshots are time-based and change-based, and are part of the permanent history. These snapshots may be reclaimed based on a policy, e.g., keep daily snapshots for a month and weekly snapshots for a year.

User can access the file system with the snapshot name from the. snapshots directory. This is considered as a version history in Ori and can user can recover a deleted file from. snapshots directory,

HISTORY:

Ori's history mechanism improves durability by facilitating replication, repair, and recovery. It improves availability because all data transmission happens through pairwise history synchronization; hence, no distinction exists between online and offline access—any replica can be accessed regardless of the others' reachability. History improves WAN latency in several ways: by restricting data transfers to actual file system changes, by facilitating opportunistic data transfers from nearby peers (such as physically-transported mobile devices), and through a novel background fetch technique that makes large synchronizations appear to complete instantly. Finally, Ori improves sharing with its graft mechanism

Ori provides file history by three way merges and borrows Git's model for storing history. Object in Ori are Immutable and Immutable objects make it easier to record history and create fast snapshots. This history is then used to detect and resolve update conflicts. Because of three-way merges, Ori can resolve many update conflicts automatically.

GRAFTING

In today's world, users collaborate by different means, likes the cloud where the users put their files and work on them. The cloud will provide with two files whenever there is a conflict.
Another way by which users collaborate is through emails, emails maintain a version history by which the users can know the time when the documents were shared with them. Another way to collaborate is through version control like the GIT and mercurial. The version control systems have their own mechanism to avoid conflicts by maintaining different versions of the files.
The problem with all the above mechanisms is that the users have to maintain a version control system. i.e. create a GIT repo. Also, the users have to go through insane naming of the files in order to avoid conflicts.

A solution to handle all the above problem is through file sharing with versioning.

File sharing with versioning

Following are the advantages of file sharing with versioning

- The file system can automatically manage the versioning and sharing without the user's manual input.

- There is no need for the user to set up a version control system i.e. to create a git or mercurial repository.
- No file naming is required to avoid conflicts. The file histories across different file systems will enable the users to keep track of all the changes made in the past.

Grafting enables users to achieve file sharing through versioning. Grafting is the mechanism by which users collaborate and share their files with each other while maintaining the cross-file system history. Grafting is a way to share files with versioning. Data sharing with history across different file system is provided by a new mechanism called grafting.
Unlike copying, with grafting users can check whether a copy of a file includes all the changes in another. The users can check what changes have been made recently or determine what is the latest version.

The above examples shows the version history of two file systems. Alice has three file versions Bob has two versions in his repository. Bob wants to access Alice's data and Alice shares those data with Bob. Bob transforms the data provided by Alice and grafts those data in his repository. Now bob can access the version history and see the changes made. Also, **a cross repository link** is maintained between the repository which determines that the changes are coming from Alice's' repository.

In the next step, when Bob makes some changes, Alice will receive those changes and look into the history to view the changes. Similar approach can be used in merges and detecting conflicts in the file system.

When a file is grafted from source(S) to destination(D) i.e from one file system to another file system, it creates a graft commit record which consists of the following fields:

- graft-id: Unique id of the record
- graft-path: Pathname of the source
- graft-commit: Hash of the commit record
- graft-target: Pathname of destination

There is a **graft algorithm** which takes a set of these graft commit commands and forms it a part of the history. A full version of history is imported from source to the destination. Even though a full version history is imported from source to destination. The entire content of the snapshot is not needed to be stored in the history. Only the changes made in the graft file or directory are needed to be stored in the history.

ORI AS VERSION CONTROL SYSTEM

Ori's mechanism of maintaining the history acts a version control system, which can serve as an alternative to Git provides differential compression, complete file deduplication and compression whereas Ori simply provides compression and sub file deduplication.
There are few difference between GIT and ORI

ORI	**GIT**
Since ORI stores both the older version as well as the newer version, it requires more storage space	Git uses a mechanism of differential compression. (**Differential Compression** is the mechanism by which GIT takes the older version and the newer version and merges them and gives the difference between the two versions). Since GIT stores only the difference, it uses less storage space compared to ORI
According to the statistics published in the research paper, ORI has 64 % faster adds and commits and 70% faster cloning compared to GIT.	Git has slower adds, commits and cloning compared to ORI
ORI has less latency as compared to GIT	Since GIT uses the differential compression to calculate the difference between the two versions, GIT has more latency compared to ORI.
Ori has the mechanism to delete snapshots. Since ORI does not support differential compression, ORI avoids the additional steps of decompression and recompression steps which are necessary for deleting snapshots.	GIT does not have the mechanism to delete the snapshots

CONTRIBUTIONS:

Megha, Premal and Vidhya all three of us did a research on the paper individually and worked on the implementation together.

REFERENCES

1. Association for Computing Machinery (ACM)(2014, February 27). Replication, history, and grafting in the Ori file system[video file]. Video posted to : https://www.youtube.com/watch?v=yCedFCG3Wb0&t=846s%5C
   (Association for Computing Machinery (ACM), 2014)
2. Mastizadeh, A.J., Bittau, A.,Huang, F.Y., & Mazieres, D., (2013, November 03 - 06). Replication, history, and grafting in the Ori file system.
   Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles,151-166. doi: 10.1145/2517349.2522721
   (Mastizadeh, Bittau, Huang & Mazieres, 2013)
3. https://bitbucket.org/orifs/ori/issues/26/could-not-do-data-mirroring-files-inside-a.