GitSync

Xinyu Ma Zhaoning Kong

I. INTRODUCTION

A. Introduction

Named Data Networking (NDN) is a new internet architecture, where name is given to each piece of data, instead of data containers. Application can fetch data from anywhere in the network, without requiring a server to be always online. By naming and securing data directly, applications can also reason about the integrity and authenticity of the data, regardless of its provenance. These features enable some application to be implemented in an entirely different approach.

A well known application is Git, a distributed version control system. To synchronize data between different machines, Git currently uses SSH or HTTPS, which requires a logically centralized server and stable network connection. This project dedicates to suggest a new approach of running Git over NDN, replacing a centralized server with multiple nodes, which sync with each other using an NDN Sync protocol. Such an approach allows application users to collaborate in Git, as long as one of the nodes is available.

The rest of this project report is organized as follows. The rest part of Section I gives our motivation for designing GitSync. Section II gives the background of Git and NDN Sync. Section III describes the GitSync design in detail. Section IV lists current issues, and the problems to be solved in the future. Section V concludes the report.

B. Motivation

Current common practice of using Git is a centralized system, which makes the server a single point of failure. The protocol Git uses to communicate with the server is https or ssh, which does not work under an unstable network connection. Though Git itself uses a decentralized data-centric model, current workflows do not exploit this feature due to the absence of data-centric transport. Even everyone can maintain a different fork, their repositories have to be always available to the Internet so the official repository can pull commits from them. This leads to the reliance upon an online Git platform such as GitHub, which goes back to a centralized system.

NDN is a data centric protocol which supports decentralized data sharing and synchronization. Here, we develop a Git system over NDN to achieve the following goals:

1) **No single point failure**. Our system should survive any nodes being down. After a failure, our system should restart working after making a few changes to the configuration.
2) **Supporting unstable network connections**. Nodes without a continuous network connectivity should be able to fetch commits from and push local changes to any branch. The system should perform a best-effort delivery.
3) **Supporting network partition**. When the network is partitioned, nodes should be able to share local changes with other nodes in the same subnet. Those changes should propagate to the whole network after the two subnets merge.

II. BACKGROUND

A. Git

Git is an open source distributed version control system, and stores a repository as a series of "snapshots". In contrast to centralized version control systems (e.g. SVN), Git not only replicates the latest snapshot of the repository, but also the full snapshot history. To support efficient storage of all history snapshots, Git adopts a data model as shown in Fig. 1.

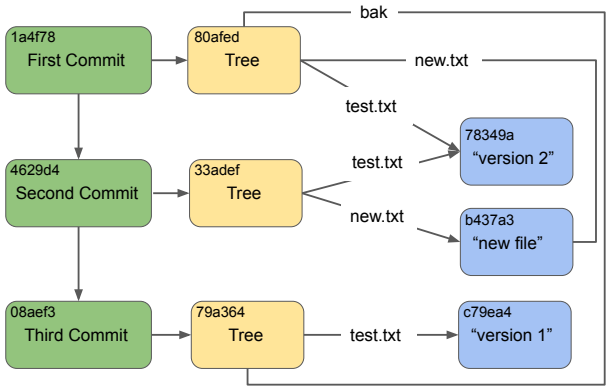

Fig. 1: Git Overview

The repository is simply stored as a set of immutable objects, in a key-value fashion in the local file system. Each filename (key) is the SHA-1 hash of the Object content (value). There are three types of objects in Git: Commit, Tree, and Blob. Every time a user performs a commit operation, a commit object is created to store information related to this commit. Each Commit object then contains a hash to a Tree object, which corresponds to a directory entry: a Tree object contains one or more entries, each of which is the hash to another sub-tree, or a Blob object. A Blob object contains the content of a file in a compressed form. These three types of objects form a directed acyclic graph (DAG), connected by their pointers to other objects.

In addition to objects, a repository contains a set of object pointers called refs. Each Git branch would maintain a ref pointer to a Commit object, which represents the latest commit on this branch. This means that whenever a user commits, the ref pointer of the corresponding branch will advance to the newly created commit object. To fetch from a remote branch, Git first obtains the ref pointer from the remote repository. Missing objects can then be fetched by traversing the DAG, starting from the Commit object referenced by the ref pointer.

B. Sync

While NDN already provides the Interest-Data exchange primitive at the network layer, it is still cumbersome to build applications directly over it. As distributed applications often involve data or state sharing between distributed parties, Sync protocols have been proposed to serve as the transport layer abstraction on top of the Interest-Data exchange primitives. Applications running over Sync protocols can simply publish data in a shared dataset, without worrying about how others would discover the new data. Meanwhile, Sync provides eventual consistency of the dataset, and is responsible for reliable transmission, resolving simultaneous updates, network partitioning recovery, etc.

Sync achieves distributed dataset synchronization by synchronizing the namespace of a dataset between multiple nodes. If a producer application wants to publish data into a shared dataset, it can inject the data name into the Sync protocol. After the state has been synchronized across nodes, the Sync protocol at the consumer side would notify the application about newly discovered names. After learning about the change in dataset state, the application would decide whether to fetch the corresponding data based on its own needs.

Multiple sync protocols over NDN have been proposed, including CCNx 0.8 Sync, iSync, CCNx 1.0 Sync, ChronoSync, RoundSync, PSync, VectorSync, etc. These protocols adopt different approaches in data naming, namespace representation and state sync mechanisms. Based on GitSyncs design, we used VectorSync for synchronizing the state between distributed nodes.

III. DESIGN

A. Overview

Fig. 2 and 3 show the overall architecture of the system. Each node runs a server daemon *gitsync-daemon*. The server daemon communicates with other nodes via NDN. It has 3 functions. First, it runs a synchronization protocol to share updates of branches over the network (Section III-B). Second, it stores all Git objects in a key-value database and makes them available to the network (Section III-C). Third, it processes push requests to the branches it manages and publishes branch updates (Section III-D).

Git uses a helper *git-remote-ndn* to communicates with the server daemon (Section III-E). The helper program works as a temporary server working on local Git folders, processes list, fetch and push commands from Git program and transfers objects between local folders and daemons.

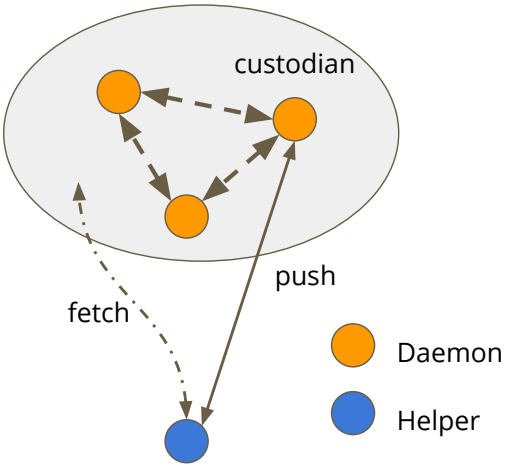

Fig. 2: System Architecture

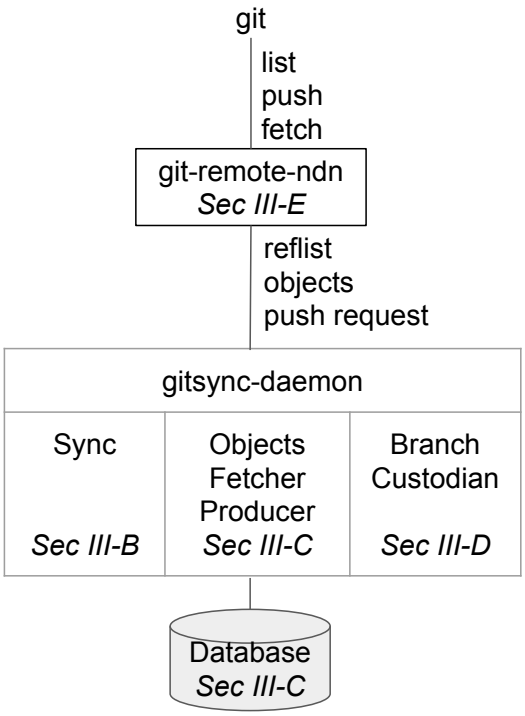

Fig. 3: A Node's Structure

Our design uses two namespaces, one for fetching and the other for commands. All objects and refs under a specific repo share a same repo prefix (Section III-C). In our implementation, we use *"/git"*. For specific commands like push, which needs to be processed on a specific site, we use the site prefix given by the site manager (Section III-D). The site prefix can be got during the site setup, e.g. receiving from AP. The list of names used in GitSync is illustrated in Table 1.

Names	
objects	/git/<repo_prefix>/objects/<hash>
sync	/git/<repo_prefix>/sync
refs	/git/<repo_prefix>/refs/<ref_name>/<timestamp>
ref-list	/git/<repo_prefix>/ref-list/
push	/<custodian_prefix>/push/<repo_name>/<branch_name>/ -<ParameterSha256Digest>

Table. 1. Naming

B. Sync

To efficiently represent the data names in the shared dataset, VectorSync names data sequentially: besides the dataset prefix, each data packet has a producer prefix and a sequence number attached to its name (e.g. *"/<dataset_perfix >/<producer_prefix>/<data_seq>"*). Producers publish data under its own prefix with monotonically increasing sequence numbers, so that the entire dataset can be represented by: (1) the group prefix, (2) the producer prefixes, and (3) the sequence number of each produce prefix. These information are encoded into a state vector that contains a list of *<producer_prefix, data_seq>* pairs.

The dataset state synchronization process consists of exchanging two types of packets: Sync Interests and Sync Replies. The name of the Sync Interest first contains a dataset prefix for demultiplexing. Sync Interests contain a state vector in the interest parameter, representing the latest dataset state known by the sender. To synchronize, each node sends Sync Interests periodically. Nodes receiving a Sync Interest would first merge the state vector with that of their own, and then send a Sync Reply containing the merged state vector. Sync will notify the application of any new states discovered during the state vector exchange through a callback.

Given that Git objects form a DAG, and each branch is represented by a ref pointer to the DAG, its sufficient to synchronize just the ref pointers. After Sync notifies the GitSync application regarding the update of a ref pointer, the application (GitSync) can first fetch the value of the ref pointer. After obtaining the ref pointers value, missing Git objects can be fetched by traversing the DAG, starting from the object pointed to by the ref pointer.

However, ref pointers are not immutable, since its value would change after each commit. Achieving immutability requires assigning ref pointers a different name whenever it is updated. To name ref pointers directly, we append the commits timestamp after the refs name. Given that time increments monotonically, we use timestamps as sequence numbers in VectorSyncs state vectors directly, meaning that the state vector contains a list of *<ref_name, timestamp>* pairs.

C. Object Storage and Fetch

The Git repositories are stored in a key-value fashion. Clients typically store repositories in the local file system, so that local Git tools can access it to perform various operations. Each object is stored as a file, the path to the file serves as the key, and the objects value is the file content. On the server daemon side, the storage implementation is transparent to the Git client, as long as the server daemon can respond with git objects upon request. Server daemon can either follow the same approach as the Git client to store the repositories in the local file system (i.e. bare repositories), or it can run over a key-value store database.

GitSync nodes uses MongoDB for storage, maintaining a series of MongoDB collections under the same MongoDB database. We use gitsync as the name for the database, but this name can be configured by the operator. Under this database, we maintain an object collection for storing Git objects, a repos collection for storing the names of tracked repositories by this node. In addition, we maintain a per-repository collection, which stores the metadata for every branch in a repository.

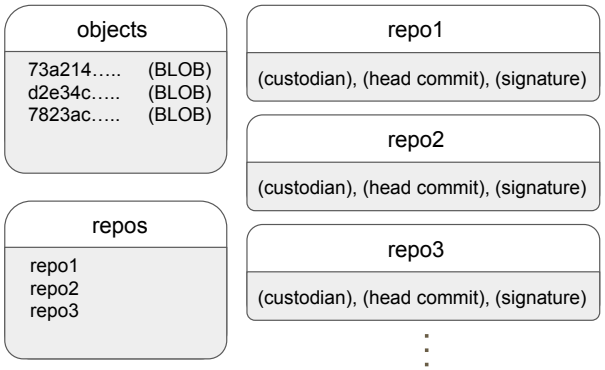

Fig. 4: Database Schema

Given that the possibility of SHA-1 collisions is minimal, we maintain an "objects" collection for storing all objects across all repositories hosted on this node. Under this collection, each object is stored as a MongoDB document with a "key" field and a "value" field. An index is maintained over the "key" field to improve lookup performance. An example of the objects repository is shown in Fig. 4.

The other MongoDB collections mentioned above are used for providing persistent storage: after a node crashes or got shut down by the operator, it can restart and bootstrap by recovering its states from the database. First, the database maintains a collection named "repos", storing the repository names that the node is currently tracking. Then, the database maintains a per-repository collection named after the repository, each containing a list of branch names which the node tracks. This per-repository collection also records metadata for each branch, including the corresponding custodians name, and a custodians signature for the latest commit. Fig. 4 shows examples of these collections.

Nodes need to fetch objects from each other in order to keep their data consistent. As mentioned in Section III-B, when Sync notices that a ref has been updated, it would notify the server daemon of the latest timestamp of this ref through a callback. The server daemon then sends an Interest to fetch the refs value from other nodes. The name of this Interest is in the form of *"/git/<repo_prefix>/refs/<ref_name>/<timestamp>"*. The

timestamp here is necessary to distinguish between refs pointing to different commits, and ensure that only the latest refs are fetched. Server daemons are also responsible for replying to ref Interests carrying the latest timestamp.

After obtaining the refs value, the server daemon sends Interests for the individual objects, traversing the DAG until all missing objects have been fetched. When the Git client sends a fetch command to the remote helper, the remote helper would fetch the missing objects in a similar fashion.

D. Object Push

We assign a custodian for each branch to manage the reference of that branch. Only the custodian of a branch can publish updates to its ref. Therefore, every push request must go to the custodian. When receiving a push request, the custodian fetches all missing files and then decide whether to accept or reject.

1) Custodian: Custodians are designed to solve the consistency problem. People may submit conflicting commits to a branch. These conflicts have to be solved by humans, but our system must get a consensus on the head of a branch. Existing consensus algorithms, like Raft, need to elect a leader or coordinator, which leads to complexity in implementation when nodes can join and leave the sync group freely. Because if multiple trustworthy nodes can publish to the same branch, we need to maintain a list of keys or prefixes for those nodes so that we can verify them. There has to be another consensus on this list. Nevertheless, none of these algorithms solve the problem of how to merge two different pushes. Therefore, it is not a good trade-off to use those algorithms.

Having a custodian in charge of each branch does not hurt the functionality. Since Git uses hash values to refer to objects, there is no need to strictly verify the producer of objects. A node can create a new branch for itself and advertise it by Sync without any further restriction. Thus, one can easily advertise its own commit without pushing to an existing branch charged by others.

A branch is managed by one party in software development. In current practice, code contributed by external parties need to be reviewed first before merged into the master branch. This makes source code more secure and robust.

A custodian can be a single node or a set of nodes controlled by the same party and sharing the same prefix. In our implementation, we assume a custodian is always a single node.

2) Push Request: To push a commit to a branch, a push request needs to be sent to the custodian of the branch. A push request is a signed Interest containing the commit to push. Its name is in the form of *"/<custodian_prefix>/push/<repo_name>/<branch_name>/<ParametersSha256Digest>"*. Its ApplicationParameters field carries the SHA-1 name of the commit. The InterestLifetime field is used to indicate how long the requester is going to wait, i.e. the custodian needs to respond with a Data to this Interest no matter whether it has finished the request. The requester needs to ensure all objects in the pushed commit is available during the InterestLifetime.

When a custodian receives a push Interest, it verifies the signature, fetches the missing objects, validates the commit and modifies the ref of the branch. Authorized push requesters are configured locally by the custodian. Existing NDN authentication method, such as schematizing trust, can be used here. After signature is verified, the custodian traverses the reference tree from the commit to fetch all missing objects. An object can be fetched from not only the requester but also other nodes who have it, especially when the commit is already pushed to another branch. After fetching all objects needed, the custodian knows if the pushed commit is consistent with current ref of the branch, so it can decide to accept or reject the push request. If the push request is accepted, the custodian publishes a new branch data and notifies other nodes by sync.

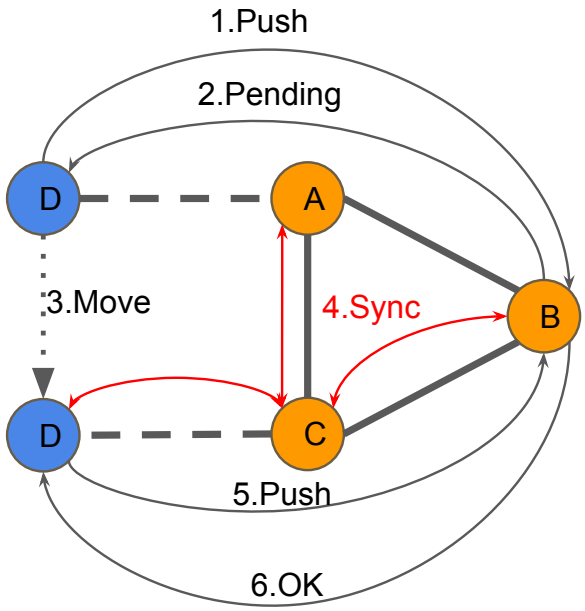

Fig. 5: Push From A Mobile Node

A custodian must respond to a push request with a Data packet within its InterestLifetime. If the custodian cannot fetch all missing objects, it replies with a Pending flag; otherwise it replies with the results, which is either a Success or a Failure. This is because there are two cases. One case is that the requester can establish a stable connection to the custodian and the objects to be pushed are small, so the push can finish in a short time. In this case, the requester can set the InterestLifetime to be long enough and expect the Data is a result. The other case is that the push cannot finish in a short time (Fig. 5). In this case, the requester can first push the commit to a personal branch whose custodian is local machine. After that, sync tries to propagate this commit to every node. The requester can then push to the target branch with a short InterestLifetime. After getting the Pending reply, the requester has no more responsibility to

make the objects available, so it can move or disconnect. The sync protocol continues carrying the objects in a best-effort way. The requester can send the same push Interest again to confirm the result.

E. Helper

The helper program git-remote-ndn acts as a temporary server to communicates with local Git program and server daemon. The helper offers 3 functions to Git: list the refs of remote branches, fetch remote commits and push local commits. Here, we continue using the word remote to describe branches stored in the server daemon, which generally runs on the local machine.

To simplify the implementation of the helper, the server daemon produces list Data and branch info Data. List Data is a collection of all branches and their refs according to the sync state the daemon has, using the name *"/<repo_prefix>/ref-list"*. Branch info Data provides details of a branch, such as custodian prefix, key name, etc, using the name *"/<repo_prefix>/branch-info/<branch-name>"*.

For a fetch command, the helper fetches the commit object and traverses the tree, fetching all missing objects. For a push command, the helper first produces all files on the local repo. And then it fetches the corresponding branch info to know the custodian prefix and sends a push request to that custodian. If the result is Pending, The helper will not retry because the time for the custodian to fetch missing objects is unknown. The procedure of push is shown in Fig. 6.

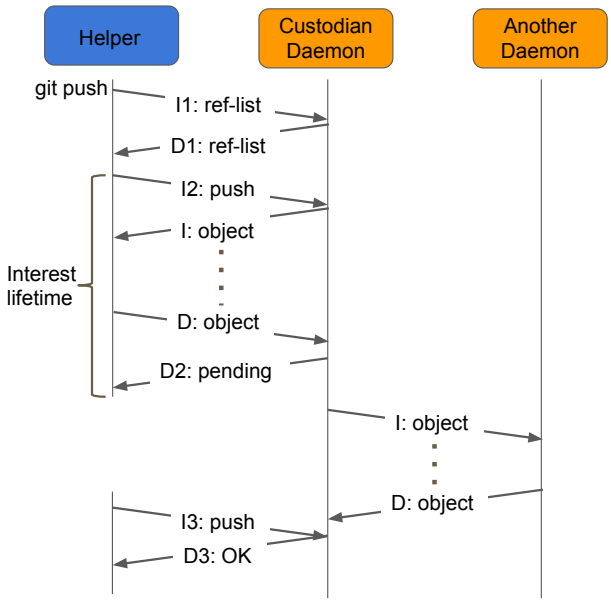

Fig. 6: The Sequence Diagram for Push

IV. DISCUSSION

A. Security

Concerns for security in our system are about the management of branches. Since all objects are named by its SHA-1, objects are less possibly to be malformed. The two issues are how to allow nodes creating branches with least restriction, and how to securely share branch meta-info, which contains the custodian and key of a branch.

In current implementation, a node tracks every branch it meets. When a branch is created, it is added into the sync. A node will fetch the branch meta-into when it recognizes a new branch. This may lead to spam attack where a compromised node creates infinite new branches, or a man-in-the-middle attack where a compromised node sends malformed meta-info.

Given that GitSync relies on VectorSync for synchronization, the sync process also has to be secured. For example, an external node may send fabricated Sync Interests to tamper with the synchronization process. We are still discussing how to solve this problem.

B. Code Review and Continuous Integration

Code review and continuous integration (CI) should work in a distributed system.

There are existing code review systems, such as Gerrit, which uses a decentralized workflow but depend on a central server to store data. This can be integrated into GitSync by storing the data in a separate repository. Similar to code commits, Gerrit allows people make comments and votes to a commit. This can be considered as a repo with a single master branch that everyone can push to. We can specify no custodian for this specific repo, because comments are generally not recalled and tempered comments are less harmful.

CI procedure takes a commit as input and generate a build and test results as output. If we take other variables that affects the result also as input, such as platform and compiler, the output will be uniquely decided by the input set. Therefore, CI procedure can be run at any node. The results can be collected into another Git repo.

C. Relationship with Repo

NDN repo is a general network storage whose data can be fetched by NDN Interests. GitSync is a special form of NDN repo, because Git adds 3 features to data:

- Historical versions are stored.
- Any specific version of Data is immutable.
- Multiple users can publish to the same name.

These features can be applied to not only source code but also a wide range of data. Nevertheless, the way code review and CI interact with GitSync discussed in Section IV-B can also represent a wide range of applications that subscribe to, consume, process and produce data without human interaction. Thus, it is highly possible that GitSync will eventually converge with NDN repo and form a general system together with self configuration and application interface such as CNL. That system decouples data storage and applications that processes data, providing a unified interface for applications to fetch and publish data, as shown in Fig. 7.

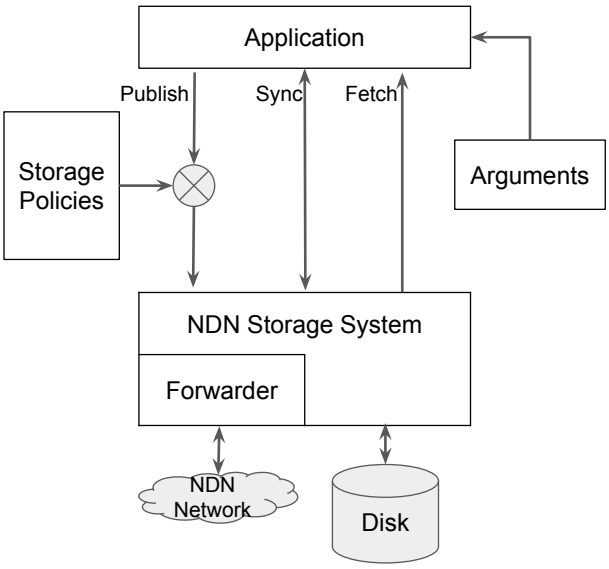

Fig. 7: Unified NDN Storage System

V. CONCLUSIONS

In this term project, we implemented GitSync, a distributed Git system over NDN. Our system has no single point of failure and can survive unstable connection and network partition. A node can publish new branches or push commits to existing branches. The change of a branch reference will be propagated by synchronization protocol. Nodes will continuously fetch missing objects.

Our work is still in its early stage. We have discussed several open issues, including the security, integration with code review and CI, and the convergence with NDN repo. Our future work is to solve these issues and make our software more convenient and robust.

REFERENCES

[1] Zhang, Lixia, et al. "Named data networking." ACM SIGCOMM Computer Communication Review 44.3 (2014): 66-73.

[2] Zhu, Zhenkai, and Alexander Afanasyev. "Let's chronosync: Decentralized dataset state synchronization in named data networking." 2013 21st IEEE International Conference on Network Protocols (ICNP). IEEE, 2013.

[3] Fu, Wenliang, Hila Ben Abraham, and Patrick Crowley. "Synchronizing namespaces with invertible bloom filters." 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). IEEE, 2015.

[4] Zhang, Minsheng, Vince Lehman, and Lan Wang. "Scalable name-based data synchronization for named data networking." IEEE INFOCOM 2017-IEEE Conference on Computer Communications. IEEE, 2017.

[5] Janak, Jan, Jae Woo Lee, and Henning G. Schulzrinne. "GRAND: Git Revisions As Named Data." (2011).