GitSync: Distributed Version Control System over NDN

Xinyu Ma
UCLA
Los Angeles, California, USA
xinyu.ma@cs.ucla.edu

Lixia Zhang
UCLA
Los Angeles, California, USA
lixia@cs.ucla.edu

ABSTRACT

Git is designed as a distributed version control system and has been widely used. However, most of the existing projects use a workflow where developers have to synchronize with a central server. This poster explores the possibility of a truly distributed git platform, dubbed GitSync, by making git run over Named-data Networking (NDN). GitSync uses a peer-to-peer protocol to remove the need for central servers, therefore enabling higher availability when not all users are connected to the cloud all the time. GitSync eliminates single point of failure and can continue operation over unstable connectivity and network partition.

CCS CONCEPTS

• **Information systems** → **Distributed storage**.

KEYWORDS

Named Data Networking, Version Control System

ACM Reference Format:
Xinyu Ma and Lixia Zhang. 2021. GitSync: Distributed Version Control System over NDN. In *8th ACM Conference on Information-Centric Networking (ICN '21), September 22–24, 2021, Paris, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3460417.3483372

1 INTRODUCTION

Git is a distributed version control system widely used by developers. Git is designed for peer-to-peer collaboration. However, nowadays, most developers rely on a git server, such as GitHub, GitLab, etc., to host their codebase for synchronization. At the time of this writing, GitHub has over 200 million projects [3]. These git servers may become a single point of failure in a collaborative development project. The Named Data Networking (NDN) architecture [1, 8] allows applications to use data names directly to fetch data from anywhere in the network. This inspires us to explore the possibility of implementing a git platform over NDN that removes the reliance on any single server.

In this poster, we present the design of GitSync, a distributed git platform over NDN. Instead of using a centralized node to serve as the rendezvous point for all the collaborating developers to synchronize with, GitSync enables direct peer-to-peer Git synchronization. Such an approach allows developers to collaborate in Git wherever and whenever they are connected, improving Git's availability. In this poster we first give a brief description of Git together with the limitation in its usage, then describe GitSync and a new issue in conflict resolution GitSync encountered, followed by a discussion on the root cause of the problem and potential resolutions.

ICN '21, September 22–24, 2021, Paris, France

ACM ISBN 978-1-4503-8460-5/21/09.
https://doi.org/10.1145/3460417.3483372

2 BACKGROUND

2.1 How Git Works

To enable distributed collaboration, git stores the full history of a repository on every peer, so that each peer can synchronize with another peer whenever they get connected. Different from centralized systems such as SVN, git uses connectivity between peers for codebase synchronization only; editing can be done offline.

Git uses a DAG to store invariant objects, each is referred to by its SHA-1 hash, as shown in Figure 1. Git branches and tags with human-readable names are called refs which point to git commits uniquely identified by the hashes. Every file is pointed to directly or indirectly by some refs. The path to a file can be resolved by a traversal starting from its ref. For example, the file `README` in `master` branch can be resolved to `51db3`. Therefore, when two peers synchronize, they can start with synchronizing refs, and then fetch all the missing objects referred to by the refs.

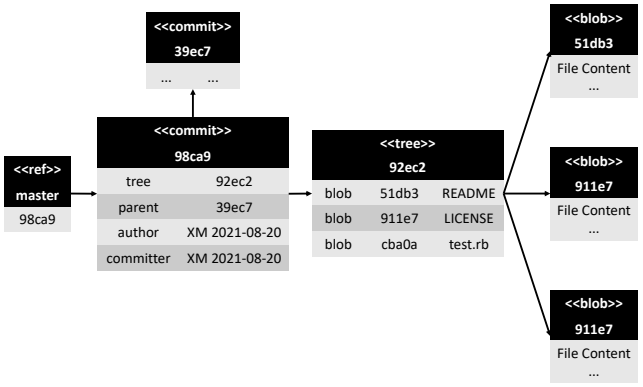

Figure 1: Git object storage

When two peers synchronize their local changes, it is possible that both of them have modified the codebase. Since commits have pointers towards their parent commits, git can discover the temporal relations among commits. If neither version is an ancestor of the other, there is a conflict. Conflicts generally need to be resolved manually. In the default setting, git can merge two changed codebase only if there is no single file that has been modified by both peers.

2.2 Centralized Git Server

Although git is designed to run as a distributed system, in its actual deployment, developers collaborating in the same project rely on

a centralized git server to synchronize, because the following two factors in today's Internet make direct peer-to-peer communication infeasible.

Connectivity Most personal computers do not have fixed IP addresses. Many of them are behind routers which reject external connections.

Security There is no widely available, usable peer-to-peer public key infrastructure for peer authentication.

Therefore, it is difficult to establish an SSH or HTTPS connection to another peer, with its identity verified.

3 DESIGN

3.1 GitSync Overview

GitSync benefits from three basic features of NDN. First, NDN communicates by names, and each peer obtains its security credentials in bootstrapping stage, which enables secure peer-to-peer synchronization via any available connectivity between them. Second, NDN lets each Git process secure its data directly, thus peers only need to verify the authenticity of received data, not the data containers connected to.Also, secured data can be stored online, without requiring specific peer to be online.

The design of GitSync has two parts[1]: one runs as a daemon on every peer, which acts as a local git server; the other `git-remote-helper` is a plugin to Git which implements NDN on the client side, as shown in Figure 2.

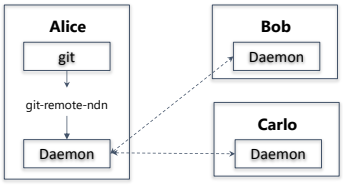

Figure 2: GitSync overview

The local client pushes to, and pulls from, the local GitSync daemon in the same way as if it connected to the existing git server. GitSync daemon takes the responsibility for git object storage synchronization with any connected peers. It keeps a local copy of the repository, and runs a sync protocol that resembles the State Vector Sync (SVS)in the background [4][2]. The Sync protocol broadcasts a synchronization Interest both periodically and after a change is made, carrying the root hash of the local storage which enables connected peers to detect changes and synchronize up as needed.

3.2 Conflict Resolving

Conflicts in git often needs to be resolved manually. This is not an issue today, because git only pushes or pulls from the central server upon a user's request, and the user is responsible for resolve any conflicts with the central server. In contrast, GitSync daemon runs *in the background*, the user may not be present when a conflict is detected with another peer; furthermore, there is no clear rule which peer should be responsible for the conflict resolution.

[1]Proof-of-concept GitSync code is available at https://github.com/JonnyKong/GitSync.
[2]GitSync implementation uses a slightly modified sync protocol to exploit the hash names and parent relations of git commits.

In our current design, GitSync simply marks the merge conflict and waits for users' to resolve. Formally, a GitSync daemon tries the following steps in order when it identifies a conflict:

(1) If one commit is a dependent of the other, the dependent is always taken as the new head.
(2) If there does not exist a single file that both commits modified, GitSync creates an automatic merge commit.
(3) Otherwise, GitSync daemon marks this conflict and requires the user to solve it when next time the user fetches from, or pushes to, the GitSync daemon.

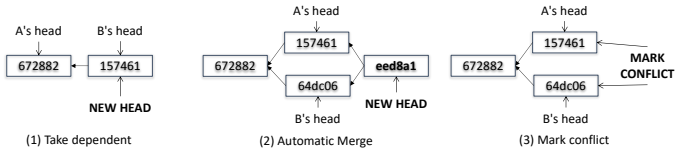

Figure 3: Conflict in sync

4 DISCUSSION

One most asked question about GitSync is why conflicts cannot be resolved automatically, given multiple successful collaborative editing algorithms exist, such as *Conflict-free replicated data types (CRDT)* [6, 7], *operational transformation (OT)* [2], and Diff-Match-Patch [5]. Unfortunately, GitSync cannot directly use them due to two reasons:

- Git has no knowledge on what type of data is stored in a file. Therefore, we cannot decide the correct strategy to merge when a file is modified by both parties.
- Though it is not required by git, in practice, a commit is supposed to be meaningful. For example, the code of a commit is expected to compile. However, a code file merged by a collaborative editing tool may not compile because the tool is unlikely to understand the code semantics.

In a different scenario where operations are applied to *specific* data types, automatic merging is possible. For example,

- Collaborative editing of a shared text file, where users make text change asynchronously. In this case, any aforementioned algorithm can be used.
- A key-value database, where named objects can be added and removed. Here, operations are commutative.

The usage of hash names in git is also worth mentioning. Since they do not contain semantics, every name resolution must start from some ref, which is a pointer with a human-readable, semantically meaningful name towards a hash-identified commit. Git's design of making every machine store everything allows git to recognize the relationship among hash-named commits by walking down the tree structure and inspecting the reference of each object. However there is no fast and easy way to identify specific files, that some application may be interested in, without walking down the tree.

ACKNOWLEDGEMENT

This work is partially supported by the National Science Foundation under award 1719403.

REFERENCES

[1] Alexander Afanasyev, Tamer Refaei, Lan Wang, and Lixia Zhang. 2018. A Brief Introduction to Named Data Networking. In *Proc. of MILCOM*.

[2] Daniel Berlin and Joe Gregorio. 2009. Google Wave. In *23rd Large Installation System Administration Conference (LISA 09)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/lisa-09/google-wave

[3] GitHub, Inc. 2021. *The world's leading software development platform · GitHub*. https://github.com/

[4] Tianxiang Li, Zhaoning Kong, Spyridon Mastorakis, and Lixia Zhang. 2019. Distributed Dataset Synchronization in Disruptive Networks. In *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. 428–437. https://doi.org/10.1109/MASS.2019.00057

[5] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.

[6] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. 39–49. https://doi.org/10.1145/2957276.2957310

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[8] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73. https://doi.org/10.1145/2656877.2656887