Source Code Management Using Version Control System

Rana Majumdar[1], Rachna Jain[2], Shivam Barthwal[3], Chetna Choudhary[4]

[1,3,4]*Amity School of Engineering and Technology, Amity University Uttar Pradesh*
[2]*Amity Institute of Information Technology, Amity University Uttar Pradesh*
[1]*rmajumdar@amity.edu,* [2]*rjain1@amity.edu,* [3]*shivambarthwal@live.com,* [4]*cchoudhary@amity.edu*

***Abstract*— Software is the most vigorous product of Information Technology. The face of software development has changed unprecedentedly and become more custom over a period of time. If we go back 15 years, one can see software were developed for supporting workstations in organizations with centralized database over multiple locations, but rise in international work standard in turn has made software development more competitive and challenging. As a result, software development happens to take place in a collaborative platform – programmers collaborate their code to a central point from multiple locations. How can programmers collaborate? Well, Source code management tools (Version Control System) tackles every single barrier associated with managing source code such as: Integrity, robustness, synchronization, linearity, and revision control. Version control adhere unique functionalities: commit, push/pull to and from code base, snapshots that makes managing source code fairly simple. However conventional version control system can't manage models and there are problems in syncing main repository and local working copy. This work draws shortcoming of 'Git' as a version control system in managing source code and gives insight into Git's association with source code taking line based approach that makes difficult to version control model diagrams with software development. The limitations are identified and the model is proposed to facilitate enhancement in future.**

***Keywords* — Commit, Git, Repository, Version Control System**

I. INTRODUCTION

Git as a Version Control tool is a vital asset of software developers that enables management of source code in the most efficient, and flexible manner that follows principle of version control. Git records every single form of a file stored at repository along with switching back and forth to any of these versions maintained in the database. Version control tools enable people to simultaneously work on a project. A developer edits his/her own copy of files and choses when to share with others, or rest of the team. Partial or temporary edition of files by a programmer will not interfere with rest of the team. SCM tool facilitates collaboration – The developers can collaborate from multiple locations, collaborating to the main code base. There are numerous version control tools to collaborate such as: Git [1], ClearCase, Subversion (SVN), Perforce, and many others, but in this paper we thoroughly focused on Git since it is the choice of developer in software industry. In software development processes, documentation and source codes written in the process are frequently changed for various reasons, e.g. customers' requirements changes, during development. Developers should have various versions of a product and manage them in their project. In this situation, the techniques for version control are significant to support their tasks by using computerized tools. Now the source code files are stored in the main repository where the software project or the source code repository is hosted. The code is generally found to be hosted at GitHub [3]. A software developer mirrors the code from the repository and can access it by having the local copy. Every time developer working in Git Environment commit changes and pushes/pull back and forth form main repository, the snapshot of a file being version controlled is taken to record and manage differences with the file.

II. LITERATURE REVIEW

This section gives insight into evolution of various phases of version management tools: Local Version Control Systems, Centralized Version Control Systems, and Distributed Version Control Systems. With rising complexity in managing source code, the need to enhance version controlling was realized which unleashed new ways of version management as a solution to underlying issues with source code control one after the other.

A. Local Version Control Systems

Initially, people preferred copying files from one directory to other by maintaining set of all updates made to a file; however, this way of managing files was soon seen as error prone technique as the files might get copied in wrong directory and it is not easy to remember in which directory a file is residing in. As a solution to this problem, local version control system [1] came into the picture having a database keeping record of files that were under revision control. RCS is such tool which is still used in many computers mainly in MAC operating systems. In other words client repository is a backup or clone to the server.

978-1-5090-3012-5/17/$31.00 ©2017 IEEE

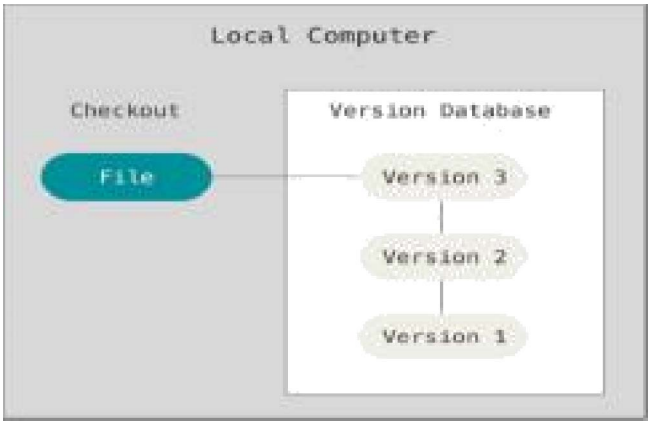

Fig. 1 Local Version Management

B. Centralized Version Control System

The local technique of version management didn't have the ability to allow developers to collaborate from other systems which brought Centralized Version Control System into picture. There are systems such as Subversion, Perforce, CVCS came into existence with just one server from where, all users/clients could checkout source code files. For over years this way of version management remained as the standard technique of source code control. The drawback of this technique was that it saw a single point failure where if the server goes down for certain amount of time, none of the client could facilitate contribution.

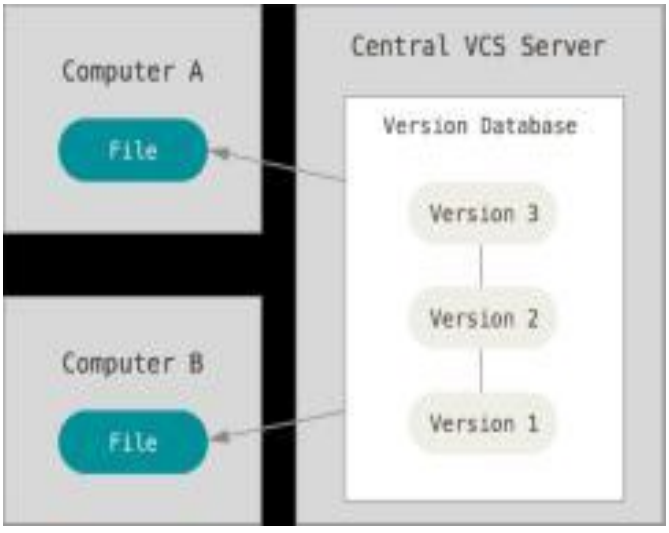

Fig. 2 Centralized Version Control Systems

C. Distributed Version Control System

Distributed Version Control tools such as Git, Bazaar, and Mercurial came into existence to tackle the single point failure issue, that saw entire repository being fully mirrored when clients checkout file.

If the server fails, either of the client repositories is available to copy the files to server.

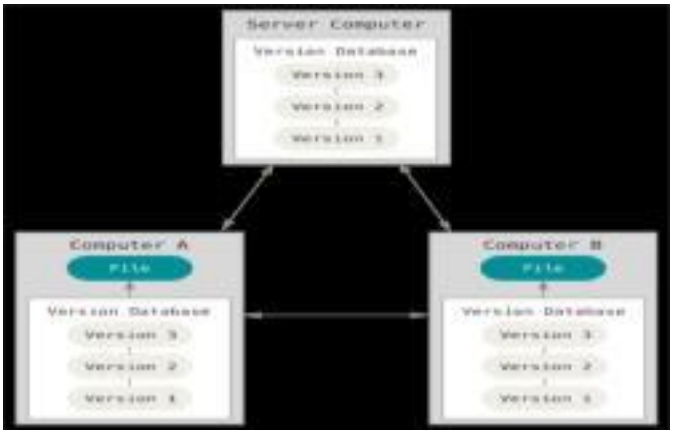

Fig. 3 Distributed Version Control System

D. Evolution of Git

There [1] is a controversial story behind birth of Git. The Linux kernel is known to be an open source project of fairly large scope. For most of the timeline in maintenance of Linux kernel (1991-2002), modifications to the software were passed as patches and archived files. In 2002, Linux kernel project started using DVCS called as Bitkeeper. In 2005, the relationship between community that developed Linux kernel and company developing Bitkeeper broke down; consequently, tool's free-of-charge status was revoked. This prompted Linux community to develop their own tool based on some of the lessons learned by using Bitkeeper. Some of the goals were: Speed, Simple design, Support non-linear development, fully distributed, capable of handling large projects like Linux kernel. Since 2005, Git came into existence.

In the effort of suggesting enhancement to the model of Git, the researchers have taken numerous approaches [5] all aiming to make Version Control systems more efficient and introducing powerful features. The researchers [2] took version controlling to a higher grade when they proposed suggestions for making tools manage diagrammatic models other than source code files. In consistent effort to advocate simpler and efficient way of version control, the researchers [3] performed detailed study of Distributed Version Control tools over Centralized ones with an empirical viewpoint by collecting results from developers from various organization.

The history is evident that Distributed Version Control Systems [1] such as Git are the choice of industry over centralized ones. The researches done in the past haven't given light to improvement possible in the architecture and methodology of Git. In this paper we have identified the shortcomings with Git, and proposed a model that could be implemented to fix this underlying problem.

III. PROBLEM CONCEPTUALIZATION

When developers in a team contribute to a chunk of code, or the contribution is made to a particular branch of development, they are free commit changes and update the history. Many a

times, the access is granted to cross-functional teams on need basis. Version Management follows practice of tracking changes on a regular interval aiming not to deviate from the actual line of development, which is done by retrieving commit history in numerous ways - searching by: author, date and time, and commit ID. The commit history is very often retrieved by searching through author name. "git log – author= author name" generates the list of commits made by the author ever since he initiated contributing to a project. Now if for some reason there arises a need to search for unknown author whether accidentally or incidentally, git terminal must respond with the message that author has not contributed to the project rather than generating no result. Figure 4 depicts under Section 3 depicts this shortcoming. The other problem that has been observed on version controlling with Git is that the commit ID in a main repository [9] doesn't corresponds with local copy, if the changes are made directly at the main repository, the local repository gets fairly outdated as compared with the main repository. Figure 5, and Figure 6 under Section 3 depicts this limitation.

IV. METHODOLOGY ADOPTED

The shortcoming identified with Git's way of version controlling is no response to unknown author commit history. This limitation currently prevails while using Git that can create biasness among the development team, perhaps causing security concern, whether or not the author ever committed changes to the project. We have identified this limitation causing security concern and proposed a model to suggest improvement in Git's architecture (Section 4 – Proposed Methodology). To illustrate this shortcoming, we monitored set of changes and updates that were made to a source code file. Every time the changes were committed, they were stored with the name of author who made the commit; however when we tried to search for commit history by unknown author not in the league of our coding project by using command git log– author = author name, the terminal returned with no response. If this scenario [8] is viewed at real time situation, this could mislead the line of actual development. If a cross functional team is working in the project, and have been assigned same rights with those in the main development team, it would be difficult to keep note of all developers involved in committing changes, if the team is not introduced to each other, or working from overseas location. So for security related reasons, the terminal must written with a message that the developer is not committing in the particular branch.

The other limitation of Git identified is that the changes when directly committed to main repository do not appear in commit history unless mirrored from repository. This can again obstruct the main line of development, mislead and create miscue working in a team when it is unknown that the changes have been made remotely. When commit history [8] is looked for in the local working end, the commit made directly on main repository of GitHub is not reflected with its Commit ID on client working end. The developer gets isolated from the changes which were introduced to the source code directly at the remote end which can bring in deviation from the main line of development. Figure 5 depicts this scenario: We hosted our main repository or remote repository on GitHub where the changes were introduced directly on the GitHub. The commit ID for the changes were not reflected when we input git log on git terminal locally. Figure 6 reflects this scenario.

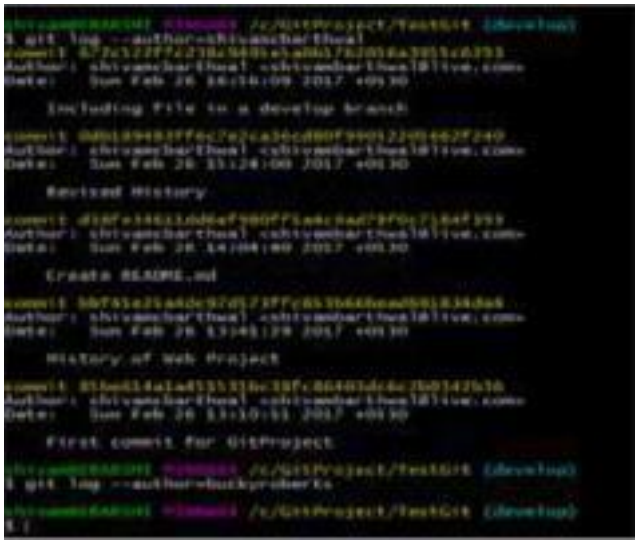

Fig. 4. Git Terminal No Response to Commit History

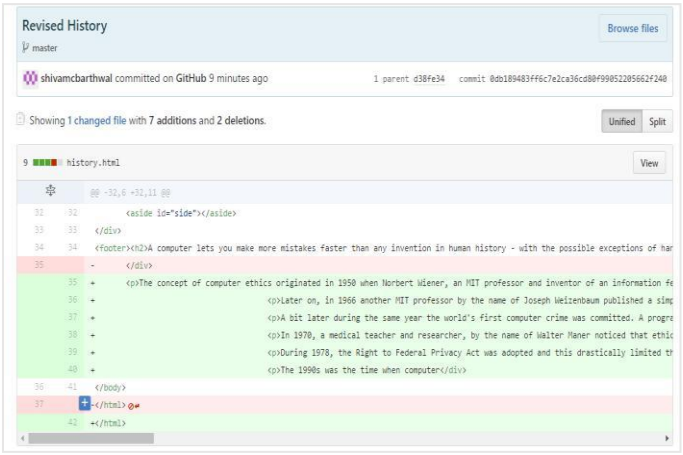

Fig. 5. Commit done on Main Repository of GitHub

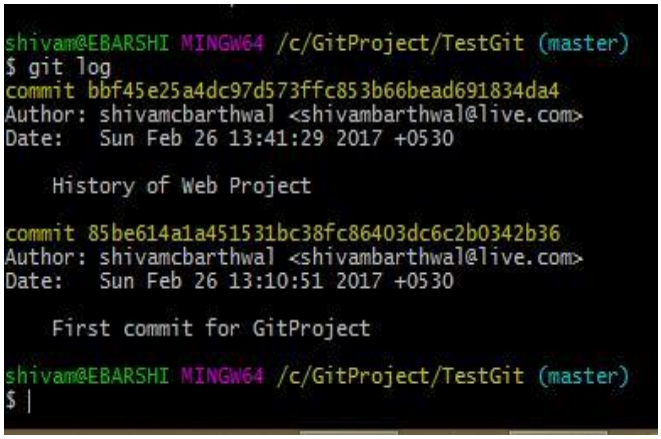

Fig. 6. Commit ID not reflected in local Git Terminal

V. PROPOSED APPROACH

We propose a tree type structure (Fig. 4) to suggest commit structure of Git, now every time a commit is made it points to successive commit and stores the information about all commits. The tree has a main object that contains list of all commits made which in succession points to next commit, thereby keeping the reference to all commits. The objects of the tree will be stored in Git repository. The commit object [1] is then created by Git along with metadata and pointer to the root object. Whenever a new commit is made, the Git repository will get updated. According to our proposition, whenever commit history is accessed by the client or developer, the changes which were introduced on remote repository will have reflection at client working end as reference to the commits are kept by the pointer object. This will eliminate the risk of isolation from updated version of source code files.

VI. OUTCOMES AND RESULTS

We have identified limitations with version controlling in Git. Our implications are: The Git terminal does not respond for unknown author history, and the commit history along with commit ID is not reflected on developer end if the changes are made directly at remote repository. The basic Git workflow goes something like this: File must be modified in working directory, then File gets staged and added to staging area. One performs a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory. So, by this work we suggest modifications in Git's architecture. A commit tree keeps details of history. This can sync the commit history between main repository and local repository. The tree will have the information about the list of authors contributing in a project with commit ID'S. So we hope our proposed suggestion will be incorporated.

VII. CONCLUSION

In this paper the version controlling shortcomings are identified and a logical tree structure is proposed to improve the working with Git. The tree contains all the information along with metadata and series of pointer pointing to successive commits.

We have given key focus to Git as a Distributed Version Control tool since the distributed version management is the recent standard for version control in software industry. We list our future work based on our findings as follow.

Improvement in Git Terminal Response: As discussed in Sect.3, the Git response to unknown commit history must be improved.

Correspondence with Remote Repository: Our findings are that commit ID generated to the commits made at the remote end must have reflection on client end whenever the commit history is looked in to remain constantly updated with changes introduced to the code in process of development, and also collaborate on updated version of source code.

REFERENCES

[1] Scott Chacon, Ben Straub " Pro Git ", 2nd edition , New York: Apress, 2014.

[2] Ben Collins-Sussman, Brain W. Fitzpatrick, C. Michael Pilato, "Version Control with Subversion", 2nd ed. California: O'Reilly Media, 2008, pp. 55-70.

[3] Mark Warren , "The Future of Git in the Enterprise ",,2015.

[4] Shaumik Dayitari, " Version Control Software ", 2014 .

[5] Alex Zhitntsky. " GitHub vs. Bitbucket: It's More Than Just Features", 2014.

[6] Github. http://www.github.com/. 2017.

[7] I. Jacobson, G. Booch,J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley, 1999.

[8] Concurrent Versions System. http://www.cvshome.org/. 2017.

[9] RevisioncontrolSystem. http://www.cs.purdue.edu/homes/trinkle/RCS/. 2016.