SOFTWARE VERSIONING IN THE CLOUD
Towards Automatic Source Code Management

Filippo Gioachin*, Qianhui Liang*, Yuxia Yao* and Bu-Sung Lee*†
*Hewlett-Packard Laboratories Singapore, Singapore, Republic of Singapore
†Nanyang Technological University, Singapore, Republic of Singapore

Keywords: Software development, Cloud computing, Version control system, Revisions, Collaboration.

Abstract: With the introduction of cloud computing and Web 2.0, many applications are moving to the cloud environment. Version control systems have also taken a first step towards this direction. Nevertheless, existing systems are either client-server oriented or completely distributed, and they don't match exactly the nature of the cloud. In this paper we propose a new cloud version control system focusing on the requirements imposed by cloud computing, that we identified as: concurrent editing, history rewrite, accountability, scalability, security, and fault tolerance. Our plan is to tackle these issues in a systematic way, and we present in this paper an overview of the solutions organized in three separate layers: access API, logical structure, and physical storage.

1 INTRODUCTION

With the advent of cloud computing, and the ubiquitous integration of network-based devices, online collaboration has become a key component in many aspects of people's life and their work experience. Users expect their data to be accessible from everywhere, and proximity to other users in the real world is no longer essential for collaboration. In fact, many projects and industries are set up such that different people are not geospatially co-located, but rather distributed around the globe. Projects range from simple asynchronous discussions on certain topics of interest, to the creation of books or articles, to the development of multipurpose software.

Software development, in particular, is a complex process composed of many phases including analysis, coding, integration, and testing. A large number of programmers usually collaborate to the development of a product, thus having efficient and effective collaboration tools is a key to improve productivity and enable higher quality of the software developed and better time-to-market. During the coding phase, programmers normally use version control systems (VCS) in order to record the changes made to the code. This is useful for multiple reasons. First of all, it allows different programmers to better coordinate independent changes to the code and simplify the task to integrate these changes together, in particular if the integration is performed or augmented by code reviewers. Finally, during the testing phase, it helps to detect more quickly why or how certain bugs have been introduced, and therefore to correct them more easily.

In order to be effective for the tasks described, the revision history needs to be clean and clear. If too few revisions are present, each exposing a very large set of modifications, the reader may have a hard time to understand exactly which changes produce certain results. On the other hand, when committing changes very often and in small amounts, the history may be cluttered with irrelevant changes that have already been reverted or further modified. Although the revision history is believed by many to be an immutable entity, allowing the freedom to refine it can be beneficial to better understand the code at later stages of the software development process, thus increasing productivity. These refinements to the history can be suggested either by the user explicitly, or by automatic tools that inspect the existing history and detect better sets of changes.

Clearly history refinement needs to be organized and regulated appropriately, and we shall describe some principles in the later sections. For the moment, we would like to highlight the fact that changes to the history ought not to affect users during their routine tasks of developing the software. This entails that the tools, automatic or otherwise, ought to have a complete view of all the revisions existing in the system. A cloud environment is the most favorable for such tools since all the data is known and always accessible, and users are not storing part of it in local disks.

Gioachin F., Liang Q., Yao Y. and Lee B..
SOFTWARE VERSIONING IN THE CLOUD - Towards Automatic Source Code Management.
DOI: 10.5220/0003609001600165
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 160-165
ISBN: 978-989-8425-76-8

The storage and management of many project repositories in the cloud also requires special attention. This management does not affect the user directly, but can help to significantly improve system performance and reduce the resource allocation of cloud providers. For example, if a repository and all its copies in use by different users can share the same underlying resource, the space requirement can be reduced. Ideally, the cloud should minimize the resource allocation to the minimum necessary to guarantee to their users the correct retention of data in case of failures.

This paper continues by describing related work in Section 2, and how out tool positions itself among them. Subsequently, Section 3 will focus on the requirements we deem important for our cloud versioning system, and Section 4 will give an overview of the system design. Final remarks will conclude.

2 RELATED WORK

Version control systems are heavily used in software development, and many solutions are available to this extent, starting from the oldest Source Code Control System (SCCS) and Concurrent Versions System (CVS) (Grune, 1986), which are based on a centralized client-server model. From that time till today, the definition of change-sets has been delegated onto the user, forcing him to spend time to record the changes, while the system merely keeps track of the different versions. The third generation of centralized systems is currently embodied by Subversion (Pilato et al., 2004). All these systems are based on a central repository storing all the revisions; any operation to update the history is performed with an active connection to the server. Once a change-set has been committed to the server, and becomes part of it, it cannot be modified anymore. This implies that history is immutable, and can only progress forward. This is generally viewed as an advantage, but it forces artifacts to correct mistakes when these have been pushed to the central repository. Given their structure, they also impose a limit of one commit operation performed at a time on a repository, thus limiting the potential scalability of the infrastructure.

Moreover, in Subversion, there is a single point of failure due to the presence of a local .svn directory in the user's local disk. If this directory is tampered with or deleted, the local modifications may become impossible to commit, or, in the worst scenario, corruption could be propagated to the central repository. Given the presence of complete snapshots of the repository into local disks, the central server does not have access to all the knowledge. In particular, in our view, this limits the applicability of automatic tools capable of refining the history.

More recently, distributed repositories like Git (Loeliger, 2009) or Mercurial (Mackall, 2006) have started to appear as a substitution to centralized repositories, and have become extremely popular. A key aspects of their success is their capability to perform many operations locally, even when connectivity is not available. This approach has the disadvantage that tracking what users are doing has become even harder since users can now maintain locally not only their most recent modifications, but a much larger number of change-sets. This hinders even more what automatic tools can do to improve the revision history.

As mentioned earlier, in all versioning systems, the history is supposed immutable and projects should only move forward. In fact, the history can be arbitrarily changed by contributors in any way. For example, they can apply a new set of changes to an old version, and then overwrite the repository safety checks. When the history is modified in such way, no trace will be left of the original history once the new one has propagated. This poses a serious problem for the accountability of changes for future reviewing process.

Several additions to distributed systems have enabled them to be seamlessly shared in the cloud. GitHub (GitHub, 2008) or SourceForge (Geeknet, 1999) are two such example. These also alleviate another problem of distributed VCSs, which is the management and coordination of all the clones of the same repository. On top of these hosting facilities, other efforts are integrating repositories with online software development, where users edit their code directly in the cloud. One example is Cloud9 (Daniëls and Arends, 2011). Even in this case, due to the limitations of current VCSs, a new copy of the data is made per user. The user then accesses and commits changes only to this personal copy. This imposes an unnecessary stress to the cloud infrastructures, requiring many copies of the same repository to exist, well beyond what is required for fault tolerance purposes.

As mentioned earlier, all the VCSs seen so far require users to explicitly take charge of the versioning of their codes by defining the changes they made to the code, and committing them to the VCS. This process of committing change-sets has been automated in domains other than software development. Google Docs (Google, 2006) and Zoho Writer (Zoho, 2005) are two examples where a history is maintained transparently to the users in the realm of office productivity tools: users write their documents online, and

revisions are automatically created upon save. This history is limited to a single branch, and merges are always resolved immediately. Unfortunately, for software development, merges cannot yet be resolved automatically, and branches are very important.

Our cloud versioning tool positions itself between the two systems just described: between automatic versioning used in office productivity, and explicit revision management used in software engineering. History is to be written automatically by the system upon save, and updated transparently as the user continues to modify the source code. The close resemblance to traditional version control tools enables our system to support branches and all the necessary issues typical of software engineering. Automatic tools can harvest the history and change it to present it to users in a better format; users can also specify how revisions should be created, if needed. In particular, the extreme flexibility at the basis of our design enables new automatic tools to play an important role in the future.

The field of software engineering has been vibrant with new ideas on accurate automatic detectors of relevant change-sets. For example, new tools are being proposed to discover code changes systematically based on the semantic of the source code (Kim and Notkin, 2009; Duley et al., 2010). These tools are complementary to our work, and can use the cloud VCS to refine the history to something that can better describe the changes made to the code, without requiring user intervention.

3 DESIRED REQUIREMENTS

After we studied in detail current solutions for software revision control, and the structure of cloud computing, we realized that these did not match. We therefore analyzed the features of a VCS residing in the cloud, and consolidated them in the following six areas.

Security. Content stored in the system may be confidential and may represent an important asset for a company. In a cloud environment, content will be exposed to possible read, write and change operations by unauthorized people. This naturally poses security concerns for developers and companies using this VCS.

We identified three key issues for security: access control, privacy, and data deletion. When new content is created, appropriate access control policies must be enforced, either automatically or manually to limit what operations different users can perform on the data. When existing content is updated, these policies may need to be updated accordingly, for example by revoking access to an obsolete release of a library. Privacy is a closely related issue, and it refers to the need to protect user assets against both unauthorized access, such as from other users of the cloud, and illegal access, such as from malicious parties or administrators. Finally, some data stored in the VCS may be against local government regulations or company policies, in which case all references must be permanently destroyed.

Fault Tolerance. Fault tolerance is always a benefit claimed for cloud-based offerings. Current VCSs do not support fault tolerance by themselves. Instead, backups of the systems where they reside is used to provide higher reliability to failures. We foresee the need of making fault tolerance an integral part of the VCS itself. Issues to be addressed in this process of integration include the methodology to create replicas of the repositories transparently, the granularity at which repositories are partitioned, the speed of propagation of updates, and the location of the copies retained.

Scalability. As the number of developers using the VCS changes, the cloud needs to adapt by increasing or reducing the amount of resources allocated. In particular, codes and their revisions can easily increase to the extent that more nodes are needed even to hold a single project. A distributed storing architecture is therefore a natural option. Furthermore, scalability entails the efficient use of the allocated resources. If this was not the case, and too many resources were wasted, the scalability as a whole could be severely hindered. For example, if the development team of a project is geographically distributed, it would be beneficial to respond to requests from servers located nearby the specific users, and not from a single server.

History Refine. As explained earlier, after changes are made to the source code, and these have been committed to the repository, developers may find it useful or necessary to improve their presentation. For example, if a bug was corrected late in the development of a feature, it is an unnecessary complication to show it explicitly to reviewers; instead, it could be integrated at the point in time where the bug first appeared. Naturally, the original history is also important, and it should be preserved alongside the modified one. In traditional VCSs this is impossible to accomplish, and even the simpler action of rewriting the history (without maintaining the old one) may not be easily accomplishable.

Accountability. Every operation performed in the system should be accountable for, be it the creation

of a new version, the modification of access permissions, the modification of the history, or the deletion of protected data. In traditional VCSs, only the first two are correctly accounted for, and only if the system is not tampered with. All the others are ignored; for example, if the history is modified, no trace is left of who modified it or why. Clearly, even who has access to the accounting information needs to be correctly regulated.

Concurrent Editing. Simultaneous writes is a highly desirable feature in cloud versioning. When many developers of a project are working on the same items, it is likely that they will commit to the repository at the same time, especially if commits are automated, say every auto-save. In this case, there will be many simultaneous writes to the same objects on different branches. (To note that here we are not trying to automatically resolve merge conflicts, since different users will never be sharing the same branch. Merges are still resolved manually.) There are tradeoffs between whether we want a highly available system or a highly consistent one. The key, as we shall see, is to determine the minimum consistency required to match user expectations, and provide therefrom the highest availability possible.

4 SYSTEM DESIGN

To address the requirements described above for our cloud VCS, we divided the design into three layers as shown in Figure 1. At the topmost level the access API is responsible for coordinating the interaction between system and external applications. At the core of the VCS, the logical structure defines how the data structures containing the users' data and the relative privileges are implemented. At the lowest level the physical storage layer is responsible for mapping the logical structure onto the physical servers that embody the cloud infrastructure; it takes care of issues like replication and consistency. Each shall be described in the remainder of this section.

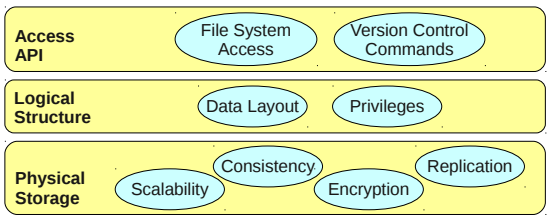

Figure 1: System design layout.

4.1 Access API

This layer is the one visible to other tools interacting with the VCS. Its correct definition is essential to enable third-parties to easily build powerful tools.

Version Control Commands. As any VCS, our cloud solution has a standard API to access the stored versions and parse the (potentially multiple) histories. This API is designed to allow not only traditional operations like creation of new commits and branches, but also more unconventional operations, like the creation of a new version of the history between two given commits.

File System Access. Files in the cloud are generally accessed through web interfaces that do not require to see all the files in a project. For example, in a project with one thousand files, a developer may be editing only a dozen of them. The other files do not need to occupy explicit space in the cloud storage—other than that in the VCS itself. Therefore, the cloud VCS provides a POSIX-like interface to operate as a regular file system. This empowers the cloud to avoid unnecessary replications when files are only needed for a very short amount of time, such as during the compilation phase when executables are produced.

4.2 Logical Structure

The logical structure of the system is the one that is responsible for the correct functionality of the VCS. It defines how data is laid out in the system to form the revision histories, and how users can access projects and share them.

Data Layout. Our VCS is based on the concept of snapshot: each version represents an image of all the files at the moment the version was created. The basis of the system is a multi-branching approach, where each user is in charge of his own branches, and he has exclusive write access to them. This implies that during the update process, when new revisions are created, the amount of conflicts is reduced, allowing easier implementation of concurrent editing. Given the lack of a traditional *master* or *trunk* branch, any branch is a candidate to become a release or a specially tagged branch. Privileges and organization ranks will determine the workflow of information. Due to the requirement of history refining and accountability for its changes, each component stored in the system is updatable inline, without implying a change in its identifying key. In addition, to enable automatic tools, as well as human, to better interact with the system, each node has tags associated, which describe how the node was created and what privi-

leges the node itself grants to different tools and users.

An important feature of the data layout is the capability to protect its integrity. In this context, we need to worry only to detect a corruption in the system, since data correction can be performed with help from the lower layer, and in particular of the data replication mechanism. Corruption may occur due either to bugs in the implementation of the API at the layer above, or to malfunctioning of the hardware devices. The latter problem can be solved by adding simple error detection codes like SHA or CRC. As for the former, automatic tools will check the status of the system after risky operations to ensure the integrity of the system. In all cases, when a corruption has occurred, another copy will be used to maintain the cloud service active, and an appropriate notification will be issued.

Privileges. We define each individual user as a *developer*. Each developer can create his own projects and, at the same time, participate in collaborative projects. He can also link other projects to use as external libraries or plugins. Each collaborative project, or simply project, may involve a large number of developers, each with his own set of privileges. Some developers may be actively developing the software, and therefore be granted full access. Others may be simply reviewing the code or managing releases; in this case they will be granted read-only access.

When a project is made available for others to use as a library, a new set of privileges can be expressed. For example, the code inherited may be read-only, with write permissions but without repackaging capability, or fully accessible. Moreover, some developers may also be allowed to completely delete some portion of the stored data, change the access privileges of other developers or of the project as a whole. Finally, automatic tools, even though not directly developers, have their specific access privileges. For example, they may be allowed to harvest anonymous information about the engineering process for statistical analysis, or propose new revision histories to better present the project to reviewers.

4.3 Physical Storage

The underlying physical storage design is critical to the overall performance, especially to support a large scale VCS in the cloud, where failures are common. Our design considers the various aspects of a distributed storage architecture: scalability, high availability, consistency, and security.

Scalability. Having a single large centralized storage for all the repositories would lead to poor performance and no scalability. Therefore, our design contemplates a distributed storage architecture. User repositories will be partitioned, and distributed across different storage nodes. As the number of users grows, new partitions can be easily added on-demand, thus allowing for quick scale out of the system. Naturally, the definition of the principles driving the repositories partitioning needs careful design, so that when new storage nodes are added, the data movement necessary to recreate a correct partitioning layout is minimized.

Replication. Users expect continuous access to the cloud VCS. However, in cloud environments built with commodity hardware, failures occur frequently, leading to both disk storage unavailability and network splits—where certain geographical regions become temporarily inaccessible. To ensure fault tolerance and high availability, user repositories are replicated across different storage nodes and across data centers. Therefore, even with failures and some storage nodes unreachable, other copies can still be accessed and used to provide continuous service. The replication scheme eliminates the single point of failure present in other centralized, or even distributed, architectures. In our scheme, there are no master and slave storage nodes.

Consistency. Software development involves frequent changes and updates to the repositories. In a traditional scenario, changes are immediately reflected globally (e.g., a *git push* command atomically updates the remote repository). Consistency becomes more complicated when the repository is distributed as multiple copies on different storage nodes, and multiple users have access to the same repository and perform operations simultaneously. If strong consistency is wanted, the system's availability is damaged (Brewer, 2010). In many distributed cloud storage systems, eventual consistency has been used to enable high availability (meaning that if no new updates are made to an object, all accesses will eventually return the last updated copy of that object). In our scheme, we guarantee "read-your-write" consistency for any particular user. This means that old copies of an object will never be seen by a user after he has updated that object. We also guarantee "monotonic write consistency": for a particular process all the writes are serialized, so that later writes supersede previous writes. Finally, we guarantee "causal consistency", meaning 1) that if an object X written by process A is read by process B, which later writes object Y, then any other process C that reads Y must also read X; and 2) that once process A has informed process B about the update of object Z, B cannot read the old copy of Z anymore.

Encryption. Source codes are valuable to developers and they should be unreadable to users who have not been explicitly granted access to them. We envision encryption deployed at all levels of the storage system to protect users' information and source codes. At the user level, users create credentials to prevent unauthorized access (for example in the form of RSA keys). At the source code level, source codes are protected with an encryption algorithm before being written to persistent storage. At the hardware level, individual disks are encrypted to further prevent loss or disclosure of users' data.

5 CONCLUSIONS

In this paper we discussed how online collaboration, a key technology to enable productive software development and timely-to-market deliverables, can be enhanced to meet user needs. We surveyed current tools available for collaboration, and how these have shortcomings that can hinder their effectiveness. We proposed a new solution based on a cloud version control system, where the entire repository is hosted securely in the cloud. Users can access their codes from anywhere without requiring pre-installation of specific software, and seamlessly from different devices. Moreover, the cloud VCS is planned with flexibility at its foundation, thus enabling automatic tools, as well as humans, to modify the state of the history in consistent manners to provide better insight on how the software has been evolving.

REFERENCES

Brewer, E. (2010). A certain freedom: thoughts on the CAP theorem. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 335–335, New York, NY, USA. ACM.

Daniëls, R. and Arends, R. (2011). Cloud9 IDE - Development-as-a-Service. http://cloud9ide.com.

Duley, A., Spandikow, C., and Kim, M. (2010). A program differencing algorithm for verilog hdl. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 477–486, New York, NY, USA. ACM.

Geeknet (1999). Sourceforge. http://sourceforge.net.

GitHub (2008). - social coding. http://github.com/.

Google (2006). Google docs & spreadsheets. Google Press Center. http://www.google.com/intl/en/press/annc/docsspreadsheets.html.

Grune, D. (1986). Concurrent versions system, a method for independent cooperation. Technical report, IR 113, Vrije Universiteit.

Kim, M. and Notkin, D. (2009). Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA. IEEE Computer Society.

Loeliger, J. (2009). *Version Control with Git*. O'Reilly Media, Inc., 1st edition.

Mackall, M. (2006). Towards a better SCM: Revlog and Mercurial. In *Proceedings of the Linux Symposium (Ottawa, Ontario, Canada)*, volume 2, pages 83–90.

Pilato, M. C., Collins-Sussman, B., and Fitzpatrick, B. W. (2004). *Version Control with Subversion*. O'Reilly Media, Inc., 1 edition.

Zoho (2005). Zoho writer. http://writer.zoho.com/.