Peer-to-Peer based Version Control

Patrick Mukherjee, Christof Leng, Wesley W. Terpstra, Andy Schürr
Technische Universität Darmstadt
{mukherjee@es, cleng@dvs, terpstra@dvs, andy.schuerr@es}.tu-darmstadt.de

Abstract

Many software projects are developed by globally distributed teams. The nature of the peer-to-peer paradigm fits to such an application scenario. However, existing tools to support developer require a central instance, like it is the case in version control, which is crucial for software development. This paper address the challenges version control faces in a purely peer-to-peer environment and presents a peer-to-peer based version control system we developed. We will evolve this basic approach, adding more features like change-set version control.

1. Introduction

Today's software development is often done in a distributed environment. Currently used tools to support software development are mainly based on the client-server communication paradigm. Applied to a setup where the users are globally distributed some drawbacks arise, which can be solved by porting the tools to use peer-to-peer communication, as shown in [4]. There is a communication bottleneck due to the fact that all data exchanges have to be transmitted through a server, often making communication slow, with poor scalability. If two clients, which might be geographically close to each other, want to exchange information, they cope with additional latency introduced by a distant server. If the server fails, the whole system stops working and data could be lost, making the server a single point of failure.

Additionally global software development has to cope with the lack of appropriate support tools. Currently there is no development environment specifically designed to support distributed projects; the available tools are designed for on-site development where multiple clients are using one server, i.e. the client-server communication paradigm.

Mature peer-to-peer applications like instant messaging or file sharing can give significant support to such a setup. While file-sharing offers the basic need to share working results, it lacks in providing version control. Using a version control system is crucial in a software development project. A file sharing system can not be used as a substitute as files are assumed to not change and concurrent updates are not supported.

Many version control systems have been developed, but very few are built purely in a peer-to-peer fashion. Solutions which are not entirely client-server based (e.g. distributed version control systems like GIT [11]) solve just some of the drawbacks the classical approaches have (like robustness issues).

The contribution of the paper is the following:

- We present a purely decentralized peer-to-peer version control system and
- discuss the benefits this approach brings in the comparison to other existing version control systems.

After summarizing the basic features a version control system should offer (Section 2), we take a closer look on existing approaches in Section 3. We propose a version control system in Section 4 which utilizes the FreePastry framework[1] for data exchange. We discuss the systems improvement over existing solutions in Section 5 and give a peek view on the running implementation in Section 6 to conclude our work in Section 7.

2. Basic Requirements

A version control system should offer the following basic functionality, in order to fully support the needs of distributed software development teams:

- **Exchange Revisions:** After a developer edited an *artifact* (the product of his work, e.g. source code or design documents), he stores his changes (*commit*) in a *repository*, which is technically realized differently in existing solutions (see Section 3). Another developer can get a version of this artifact from the repository (*check-out*). The version control system should not only offer the most recent version, but any (older)

[1] http://www.freepastry.org/freepastry

version committed earlier as well. If the artifact a developer edited did not include the most recent changes which were committed in the meantime by other developers, he can integrate those changes (*update*) and commit his local file as the newest version without loosing any changes.

The commit procedure should be *atomic*. Especially when the committed file gets stored redundantly all copies should be identical, i.e. the system has to be in a *consistent* state. If a set of changed files is committed, either all or no file commits should be successful, which might require the system to roll back already applied changes. Once changes are committed they should be *persistent* in the system and *available* for a check-out at any time.

- **Support Collaborative Work:** A system should be able to handle *concurrent changes* on an artifact. Older systems use (so called) *pessimistic locking* of the edited document for the first developer, so that others have only read-access. A more flexible way is to lock the document when it is committed: the first author's changes form a new version, while all subsequent authors who edited the same file's *base version* have to integrate his changes first. The term base version refers to the version of a file where an author based his modifications on. A check-in is only valid, if the changes are based on the most recent version.

  Artifacts should be protected by *access control*, which denies access to unauthorized persons. If access is granted stored files should always be retrievable. The system should be usable as a *backup* for all stored versions. Sometimes groups work on different parts of the same set of files. It should be possible to commit the changes of a group without the need to integrate the concurrent modifications of another team before a stable state is archived. This *parallel modifications* should be tracked using *branches*, which are (in most systems shallow) copies of the main repository. By *merging* a branch into the main repository (named HEAD) the independent modifications can be united.

- **Identify Changes:** To *chronicle modifications* in an historical order each change has to get an ascending unambiguous *version number* which identifies the state of an artifact when its changes were committed. By this unique identifier a document should be retrievable at any time in the same state it was originally committed. Two additional notes allow to identify stored modifications better: *Comments* explain stored changes. *Tags* allow labeling all files in a specific version with a meaningful name. They are used to mark a stable release in practice. Usually multiple, semantically connected files are modified and committed at once. It is helpful, if these *change sets* are *supported*, which enables a developer to check-out multiple files in the correct versions. For grouping them the described tagging mechanism could be used, but they should be additionally committed atomic, like single files, avoiding an inconsistent repository.

  By storing the author's identifier with committed changes he can be *held responsible* for the modifications he made. This implies the need for an authentication mechanism.

- **Reorganizing the Repository:** Keeping track of *file path*, *folder structures* and *name changes* is an important feature of version control systems, especially after refactoring a project.

3. Related Work

Since the first version control system, [8], many other systems[2] have been developed. In this Section some exemplary approaches are discussed.

3.1. Client-Server based Version Control Systems

The most mature version control systems are based on the classical client-server communication paradigm. The repository for storing the artifacts under version control is centralized on a server, often realized as a database (e.g. subversion [6]) or in a file system with special description files (e.g. CVS [1]). Some systems utilize several machines, i.e. a server cluster, to provide version control. This machines are synchronized to behave like a logical single machine.

This technique proved to be useful for on-site development, where all participants are close to the central server, i.e. in the same building, floor or even office. But the client-server architecture brings some obvious drawbacks. If the server fails, the whole system is unusable. Additionally it might become a bottleneck in the system as users have to access it for every operation. Maintenance and setup costs are a burden especially for small projects. Projects with globally distributed members have to cope with communication delay due to physical distance to the server.

3.2. Distributed Version Control Systems

In the last ten years, version control systems with a distributed repository (DVCS [7]) arose. Every participant stores the whole repository on its local machine, breaking the centrality of client-server based solutions. In these

[2] A comprehensive overview: http://better-scm.berlios.de/comparison/comparison.html

systems the paradigm shifts from working cooperatively on a single repository to working on individual file-sets (branches) [11]. There is no concurrent update conflict, as everybody works on his own local repository. Changes of other developers get integrated into the local repository (pull) or local changes are sent to the repositories of others (push) initiated by the user. To get a repository with all changes integrated, all local repository would have to be merged, which is not the desired way to use this kind of systems. Usually in DVCS one of the repositories is known to provide the latest stable version, but not the most recent committed modifications. Prominent examples include mercurial [3] and git [11].

Distributed version control systems overcome some of the problems client-server based systems have to cope with, but they introduce additional problems. Some versions of the files are accessible all the time, as they are on the local machine. There is no bottleneck or delay, except when updates of a single participant gets pulled by many user. The weaker control over the changes can be problematic. Modifications have to be actively pulled from other developers and merged with the own changes, which creates an unique repository. Changes on files get only backed up if another developer pulls them into his repository.

Code Co-op [4] is a variant of this approach, which claims to be peer-to-peer based. Like the other solutions every participant has the whole repository on his local machine. Upon committing a change to form a new version, an update-script is sent to all other developers. This is sent using e-mail or local network shared folders, instead of a peer-to-peer overlay protocol. The machines of the other developer would acknowledge the change only if they did not change the committed file. When all other developers give the confirmation, a new version is created by applying the change to all local repositories of all developers. Therefore, all repositories should always be in the same state. All changes are omnipresent and all repositories are up-to-date. However, the acknowledgment-protocol can slow down the system significantly introducing many messages.

3.3. Peer-to-Peer based Version Control Systems

Borowsky et al. [2] present an inspiring approach to a peer-to-peer based version control system. The main focus of this work is to show how traditional client-server based systems can be modified to use peer-to-peer communication. A complete CVS server was installed on each peer. By assigning an ID to each file it gets distributed among the peers following the normal DHT approach. A responsible peer's CVS system provides mechanisms for a file version control. This results in a truly distributed repository, where each peer has a small set of files under control.

[3] http://selenic.com/mercurial/

[4] http://www.relisoft.com/co_op/index.htm

Unfortunately it lacks in practical usage: user management is nearly impossible, as an user account would have to be setup on every peer. Forgery (e.g. using multiple accounts) is hard to avoid. The proposed approach seems to be extremely difficult to extend with new features, which has not been the focus of this work.

In pastwatch [12] all artifacts in a given version form a repository revision, as they do in subversion [6]. If some of these artifacts are changed, a new revision is formed and a unique revision-id is given. The revisions are stored in form of a *revision tree (revTree)*, where a revision points to the revision it is based on (the *parent revision*). Similar to the DVCS approaches (in 3.2) this revTree is mirrored on each peer. Changes are first applied to the local repository and then synchronized with the repository on other peers. When changes on the same parent revision are synchronized, a branch is created, as these changes might conflict. The local repositories can be synchronized using a server, which acts like a rendezvous point. By applying local changes to this machine and receiving changes applied by other peers, the repositories on all peers become eventually identical copies. Alternatively, the peers can exchange their changes directly. When no rendezvous server is used, a member list is stored in the DHT, assigned to a peer and replicated by its neighbors. This list contains all participating peers. When the newest version is requested, all peers on this list are asked for their latest revision.

When a rendezvous server is used, pastwatch shares the disadvantages client-server systems have, i.e. dependence on a single point of failure which additionally might be a bottleneck. Without this central synchronization point, requesting the latest version is quite expensive with a growing number of messages in a large-scale network, as each participant has to be asked for its latest local version. In both setups many branches might be produced, especially when a high churn prevents the distribution of the latest changes to all peers. It is likely that changes are based on a revision which is not the latest of all revisions on all peers, sequentially leading to a branch when the revisions get eventually synchronized.

4. Our Approach

In this section we present our peer-to-peer based approach to version control which addresses the problems of existing solutions and fulfills the requirements presented in Section 2.

We build our solution on top of the p2p-framework FreePastry , which implements the Pastry overlay routing and maintenance [10]. However, our solution is not limited to use the pastry protocol, as its overlay implementation is

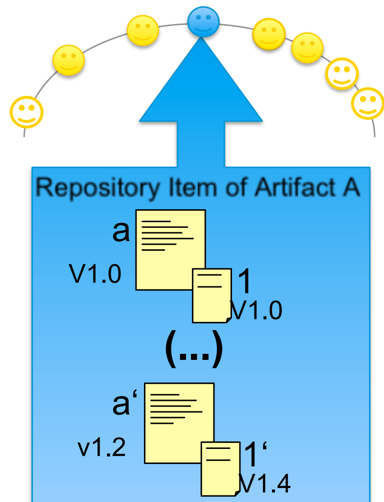

Figure 1. A peers local repository item.

abstracted by the KBR [3] interface. Our application can use any overlay which routes key-based, i.e. which retrieves the same peer when routing for a key from an arbitrary peer (assuming the network did not changed).

4.1. Distribution of the Repository

In our approach we are not distributing copies of a single repository. Instead the repository itself is distributed among several peers. Each artifact and all its revision form a *repository item*, which is stored under an unique identifier in the system. Figure 1 illustrates the structure. All revisions of an artifact are stored with an ascending revision number on a peer, pictured by the box labled *a* (revision v1.0) and *a'* (revision v1.2). To each revision a companion file (labled *1* and *1'* in the figure) is added. It stores meta information about the file, like the last author, a comment, its revision number, etc. *Branches*, *comments* and *Tags* are realized by storing them directly in this companion file. We implemented this file in a way that any additional information can be stored and retrieved as a key-value pair.

Similar to the idea of DHTs each file gets mapped to the peer (*the primary owner*) whose identifier (*peer id*) is closest. To find this peer a message with the artifact's identifier as the destination is routed. The overlay routing algorithm takes care to find it among all peers which are online.

4.2. Fail-safe Persistence

The autonomy of the peers makes robustness an important issue that each peer-to-peer system has to cope with. Each peer can leave the system at any time. We first present the mechanisms to provide system robustness as it directly influences the system operations which provide consistency and coherence. DHT-like data storage systems are robust by utilizing replication: Data is not only stored by the primary owner but also by its *replica peers*, i.e. the peers with the next smaller or bigger key value (*neighborhood*). The number of replica peers is configurable - the higher it is, the more maintenance messages will occur, the smaller, the less failure-save the system is.

The neighborhood of a peer might change when peers are joining, leaving or fail. As a peer is sending periodical messages to check the state of its neighbors it will detect changes. If a failed peer was among the peers hosting a repository item (i.e. the primary owner or a replica peer) its closest neighbor takes his position. In order to keep the number of replica peers constant a new peer among the peers in the neighborhood is assigned to be a replica peer. The latest status of the stored repository item is copied to this peer, where only the missing revisions are transmitted. A leaving peer updates his substitute before it leaves. If a peer joins whose id is closer to the repository item id then a hosting peers id it displaces one of them. The joined peer gets updated, the displaced peer can keep its repository item, but will no longer be informed about new revisions. As revisions are immutable it does not harm the system if a peer keeps its repository item when leaving or failing from the system. A leaving or failing primary owner can be handled like a replica peer. Its closest neighbor will notice its absence and take over its duties.

In the unlikely case where all hosting peers fail simultaneously an artifact and all its revisions are temporally irretrievable, until one of the failed peers rejoins. If missing its locally stored revisions will be added. It is very likely that it will be responsible for these files again, so in most cases no additional data transfere would be necessary. Only updates will be transfered. The described algorithm makes our system *resistant to peer failures*; files are stored *permanently*, allowing every user to *backup* his data and retrieve it at any time from the distributed repository.

The implementation PAST [9] takes care of replication. However, as it replicates whole files only, we modified this mechanism to transfere only the latest revision of an artifact to the replicating peers.

4.3. Check-out

The main function of a version control system, retrieving a file, is done by sending a CHECK-OUT message to its primary owner. If the latest revision of an artifact should be retrieved this request is forwarded to its primary owner, as only this peer and its replica have the most actual state. The request for a specific version gets routed in the same manner to the primary owner. Since a revision is immutable (changes result in a new revision) a peer can answer the re-

quest using its local cache instead of forwarding it to the next hop on the route to the artifacts primary owner. It is likely that a requested revision is in a peers local cache as a developer usually checks out a bunch of files to modify a subset of them.

Multiple artifacts are checked-out from different peers (depending on the distribution). Especially when the transmitted files are big and the upload bandwidth of the peers are small checking-out files from different peers might be faster then receiving all files from a single peer.

We currently save the whole file revision as a new version. We could apply an algorithm which calculates a difference (*delta*) between the versions. By providing the version number of the local file, the primary owner could compute a *patch* which converts this file to the desired version.

4.4. Commit

As described before each artifact is assigned to a peer, the primary owner. If a user submits changes to form a new revision a message is routed to the artifact's primary owner. Only this peer can process the request, as a single decision point is needed. The primary owner checks if the changes are valid. In the commit request the revision number of the revision the changes are based on (the *base revision*) is sent along with the changed artifact. Only if this complies to the latest version of the repository item the changes can be applied. If the changes are based on an older revision the latest revision is sent to the committing peer. His changes have to be merged with this latest version before a new commit can be initiated.

Concurrent commits, i.e. if two developer changed the latest version and commit their changes, are serialized on the primary owner. The overlay algorithm ensures that any two peers committing modifications of the same artifact are routed to the same primary owner. The first received changes get applied, subsequent commit requests are handled as if they where based on a outdated revision.

If a commit request is valid, i.e. the modifications are based on the latest revision, it gets applied. The local repository item is updated by assigning a new revision id and adding the committed artifact as the latest revision. Likewise all replica peers get updated. After their acknowledgment is received, the peer who initiated the commit gets informed that his changes were applied to form the most recent version.

In the case a replica peer fails during this process, it will be replaced ad-hoc by the next closest peer. If a primary owner fails the replicas can be in an updated or outdated state. The new primary owner will update all replicas with its state, which might be the old state. If the committed changes were already applied to its local repository item storage it gets copied to all other replicas. Otherwise the state without the committed changes is copied to the replica peers, forcing them to discard the probably already received update. The initiator of the commit process will not get an acknowledgment, as he was only known to the former primary owner. Hence he will commit the same changes again. In the case the new primary owner is up-to-date, he sends an acknowledgment without further action. Otherwise it propagates the changes to its replica peers as if the files where committed for the first time.

If the network falls into disconnected partitions a primary owner in each partition would handle the version control process. Different revisions based on the same latest revision could be committed, a conflict would not be noticed until the network rejoins. Only one peer is the closest peer to the repository items id. This peer keeps its history - the other peers' history is moved to a branch. The last author of a revision moved after a successful commit in this manner will be informed. We have not implemented this yet, as we consider network partitions in highly connected overlays to be extremely rare.

Concurrent changes handled in the described way provides *consistency* of the repository. The described committing procedure is *atomic* relative to single files.

5. Discussion of the System's Properties

Peer-to-peer based systems are *resistant to system failure*. As discussed in Subsection 4.2 data loss is very unlikely. Maintenance and setup *costs are low*, as every user maintains his machine himself and peer-to-peer systems are self-organizing using a maintenance procedure. No manual action has to be taken, if the network changes its size.

With the distribution of the data among the peers a *bottleneck* is unlikely to happen. However, it depends on the mapping strategy of the files to their ids. Popular files will be naturally distributed among multiple peers, enabling a peer to answer a request directly instead of forwarding it. However, the commit of changes can only be answered by the primary owner and the query for the latest revision number by the replica peers (including the primary owner), as only they are up-to-date.

In contrast to the distributed version control systems (section 3.2) all changes are immediately accessible and stored in one distributed repository. While using the same idea as Borowsky et al. [2] to distribute the repository we implemented the mechanisms to provide version control from scratch. This enables us to complement our system with more functionality, like an authentication mechanism or change set based version control, which we plan to develop.

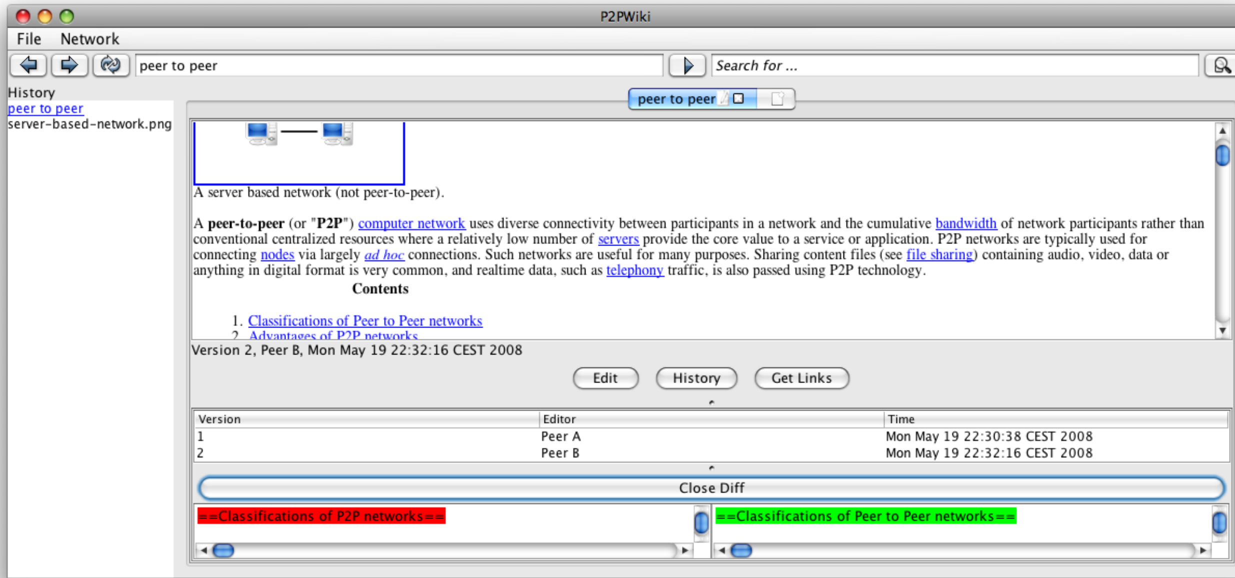

Figure 2. Proof of Concept: Integration of the Version Control System in a Wiki-Engine.

6. Proof of Concept

As a proof of concept we integrated our version control system in the wiki-engine PIKI [5], shown in figure 2. Here we track the history of wiki articles. The repository item id is retrieved by hashing the articles name. A visual diff can be shown for any two revisions.

The companion file has as an additional entry listing all wiki articles which are linking to the current one. The type of a link is listed as well.

7. Conclusion and Future Work

In this paper we presented a purely decentralized peer-to-peer based version control system which operates on single files. We discussed the benefits it brings in comparison to other existing solutions. Currently the system offers a CVS-like functionality to track the evolution of single files. We will focus our efforts to extend it to fulfill a similar functionality subversion offers, which implies integrating change sets. Further planed extensions include handling of disconnected partitions, a transactional logic to atomic commit a bundle of files, support for a folder structures, load-balanced file distribution and access control.

The version control system will be embedded into our peer-to-peer based environment for distributed development PIPE [4].

References

[1] B. Berliner. CVS II: Parallelizing Software Development. In *USENIX 1990*, pages 341–352.

[2] E. Borowsky, A. Logan, and R. Signorile. Leveraging the Client-Server Model in P2P: Managing Concurrent File Updates in a P2P System. In *AICT-ICIW'06*.

[3] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *IPTPS'03*, pages 33–44.

[4] P. Mukherjee, A. Kovacevic, M. Benz, and A. Schürr. Towards a Peer-to-Peer Based Global Software Development Environment. In *SE'08*, pages 204–216.

[5] P. Mukherjee, C. Leng, and A. Schürr. Piki - A Peer-to-Peer based Wiki Engine. In *IEEE P2P'08*, pages 185–186.

[6] M. Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[7] O. Robert. DVCS or a new way to use Version Control System for FreeBSD. In *BSDCan'06*.

[8] M. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1 No. 4:364–370, 1975.

[9] A. Rowstron and P.Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *ACM SOSP'01*.

[10] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01*, pages 329–350.

[11] L. Torvalds. http://lwn.net/articles/246381/. Clarification on GIT, central repositories and commit access lists.

[12] A. Yip, B. Chen, and R. Morris. Pastwatch: A Distributed Version Control System. In *NSDI '06)*.