[[1]](#footnote-1)

Immersing through Topology on Distributed Version Control Systems

Cristiano M. Cesario and Leonardo G. P. Murta

Institute of Computing

Universidade Federal Fluminense, Niteroi, Brazil

{ccesario, leomurta}@ic.uff.br

*Abstract*—Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but also bring greater complexity to administer and monitor the multiple existing repositories as well as the proliferation of several branches. In this paper we propose DyeVC, an extensible tool to assist developers and repository administrators in identifying dependencies amongst the distributed repositories in order to help to understand what is going on around one’s repository and depict the relations between the existing repositories.

*Index Terms* – Configuration management, Workspace awareness, Distributed version control, Repository evolution

# INTRODUCTION

V

ERSION Control Systems (VCS) date back to the 70s, when SCCS emerged [1]. Their primary purpose is to keep software development under control [2]. Along these almost 40 years, VCSs evolved from a centralized repository with local access, as in SCCS and RCS [3], to a client-server approach, as in CVS [4] and Subversion [5]. More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git [6] and Mercurial [7]. According to a survey conducted among the Eclipse community [8], Git and Github combined usage increased from 6.8% to 36.3% between 2010 and 2013 (a growth greater than 600%). During this same period, Subversion and CVS combined usage decreased from 71% in 2010 to 42.3% in 2013. This clearly shows momentum and a strong tendency in the adoption of DVCSs amongst the open source community.

According with Walrad and Strom [9], creating branches is essential to software development, because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms and to different customers, amongst other features that are expected by current software development teams. DVCSs include better support to work with branches [10], turning the branch creation into a recurring pattern, no matter if this creation is explicitly done by executing a “*branch*” command or implicitly, when a repository is cloned. All these branches, whether explicit or not, eventually will be reintegrated to their origin by means of merge operations, reflecting to the main development line the changes made.

Distributed software development, especially from the geographical perspective [11], brings a set of risk factors, and configuration management is affected by them [12]. The increasing growth of development teams, and their distribution along distant locations – even different continents – together with the proliferation of branches, introduce additional complexity for perceiving actions performed in parallel by different developers. According to Perry *et al.* [13], concurrent development increases the number of defects in software. Besides, Silva *et al.* [14] say that branches are frequently used for promoting isolation amongst developers. This postpones the perception of conflicts that result from changes made by co-workers. These conflicts are noticed only after a pull or a push in the context of DVCS. Moreover, Brun *et al.* [15] show that, even using modern DVCSs, conflicts during *merges* are frequent, persistent, and appear not only as overlapping textual edits (i.e., physical conflicts) but also as subsequent build (i.e., syntactic conflicts) and test failures (i.e., semantic conflicts).

By enabling repository clones, DVCSs expand the branching possibilities exposed by Appleton *et al.* [16], allowing several repositories to coexist with fragments of the project history. This may lead to complex topologies where changes can be sent to or received from any repository. This scenario generates traffic similar to that of peer-to-peer applications. In practice, projects impose some restrictions over this topology freedom. However, it can be still much more complex than the traditional client-server topology found in Centralized Version Control Systems (CVCS).

With this diversity of topologies, managing the evolution of a complex system becomes a tough task, making it difficult to find answers to the following questions:

* Q1: Which clones were created from a repository?
* Q2: What are the dependencies between different clones?
* Q3: Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work?
* Q4: Is it computational feasible to gather this information from all known repositories, keeping them available to be used when needed?

Most of the existing works deal with question Q3, giving to the developers the perception of concurrent changes. Palantir [17], FASTDash [18], Lighthouse [14], CollabVS [19], Safe-Commit [20], Crystal [15], WeCode [21], and Polvo [22] are examples of this kind of work. Amongst these, the only one that deals with multiple branches is Polvo, establishing metrics for assessing the merge effort between branches. However, it has a strict focus in CVCS, which are much less prone to branches if compared to DVCS.

To answer the above questions, we propose an approach called DyeVC[[2]](#footnote-2), a novel visualization infrastructure for DVCS, that gathers information about different clones of a repository and presents them visually to the user. This allows one to perceive how his repository evolved over time and how this evolution compares to the evolution of other repositories in the project. The main goal of DyeVC is three-fold: to increase the developer knowledge of what is going on around his repository and the repositories of his teammates; to enable repository administrators to visualize how the existing repositories of a project interact with each other; and, finally, to establish an extensible platform to present different information and metrics regarding distributed repositories.

This paper expands the concepts presented in a previous workshop paper [23], by including a more thoroughly discussion about our approach, as how we depict the relationship between the existing repositories in a project. It also include the analysis we have done to validate the approach, and a more detailed section on related work. This paper is organized as follows: Section II presents a motivational example to this work. Section III presents the approach used in DyeVC. Section V shows how this work was evaluated. Section VI discusses some related work and Section VII concludes the paper and presents some future work.

# Motivational Example

Fig. 1 shows a scenario with some developers, each one owning a clone of a repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working at Xavier Institute, led by Professor Xavier, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different site, which is constantly synchronized with the Institute. Solid lines in Fig. 1 indicate data being pushed, whereas dotted lines indicate data being pulled. Thus, for example, Rogue can both pull updates from Gambit and push updates to him, and Beast can only pull updates from Rogue.

Each one of the developers has a complete copy of the repository. Luckly, this scenario has a Configuration Management Plan in action, otherwise each one would be able to send and receive updates to or from any other developer, leading to a total of *n \* (n - 1)* different possibilities of communication (where n is the number of developers in the topology). In practice, however, this limit is not reached: while interaction amongst some developers is frequent, it may happen that others have no idea about the existence of some coworkers, as it occurs with Mystique and Nightcrawler, where there is no direct communication.



Fig. 1. A development scenario involving some developers

As an example, from a developer’s point of view, like Beast, how can he know at a given moment if there are commits in Rogue or in Nightcrawler clones that were not yet pulled? Alternatively, would be the case that there are local commits pending to be pushed to Gambit? Beast could certainly periodically pull changes from his partners, checking if there were updates available, but this would be a manual procedure, prone to be forgotten. What if a tool had the knowledge of Beast’s partners, and constantly monitored those, warning Beast of any local or remote updates that had not been synchronized yet?

From an administrator’s point of view, how can he know which are the existing clones of a project and how they relate amongst each other? How can he know if there are pending commits to be send from a staging repository to a production one?

# DyeVC Approach

We define DyeVC using the categories proposed by [24]: task, audience, target, medium and representation.

The **tasks** of DyeVC are three-fold. First, DyeVC should work as a non-obtrusive awareness tool to increase the developer knowledge of what is going on around his repository and the repositories of his teammates. Second, DyeVC should enable repository administrators / managers to visualize how the several existing repositories of a project interact with each other. Third, DyeVC should establish an extensible platform to present different information and metrics regarding DVCS.

The **audience** of DyeVC is software engineers using distributed version control systems. The main roles that DyeVC immediately focus are project managers and developers, but due to its extensibility, different views can be developed to support additional roles in the future.

The **target** of DyeVC is any DVCS clone of repositories hosted on a Git server.

The **medium** used by DyeVC range from simple messages to graph representations of the targeted repositories.

The **representation** should be appropriate to show the information in each different visualization. For instance, the various clones and their relationships use a network topology metaphor and the repository history use a directed acyclic graph visualization.

We detail in the following sub-section the information that our approach gathers. Next, we discuss how this information is presented using different levels of detail. Finally, we show what happens behind the scenes, this is, what algoriths are involved in the data synchronization process.

## Information Gathering

DyeVC continuously gathers information from a group of interrelated repositories, starting from repositories registered by the user. The information gathered is stored in a central document database (Fig. 2), and follows the model presented in Fig. 3. A Project is used to group all repositories that are clones of the same system. Repositories are stored as RepositoryInfo and are identified by an id. One can also give it a meaningful clone name. A RepositoryInfo also has a list of clones to which it pushes to and a list of clones from which it pulls from. The lists are represented respectively by the self-associations pushesTo and pullsFrom. The finer level of information is the CommitInfo, which represents each commit in the topology. A commit is identified by its hash and it can refer to its parents, in the case of a merge.

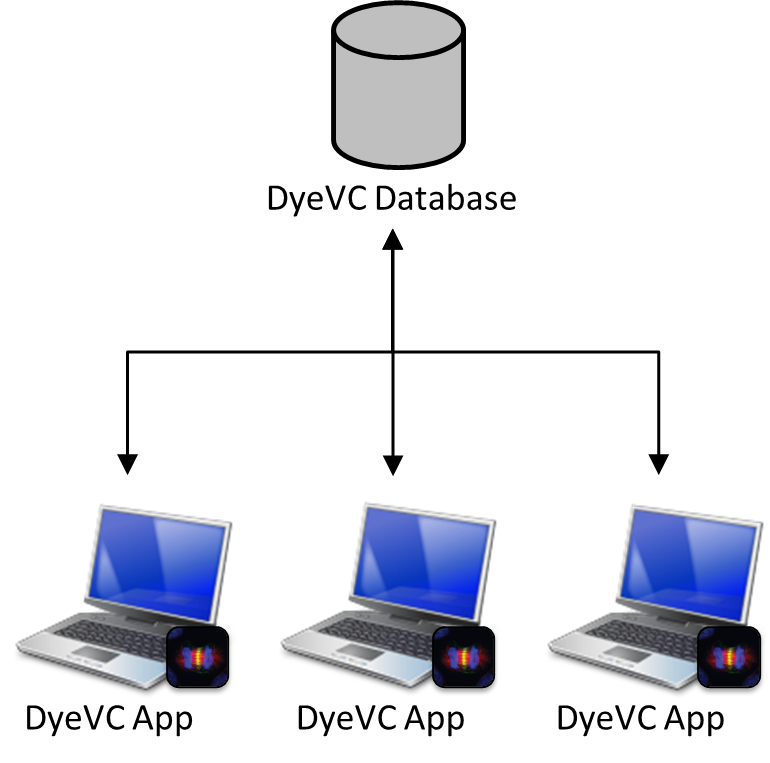


Fig. 2. How DyeVC gathers information

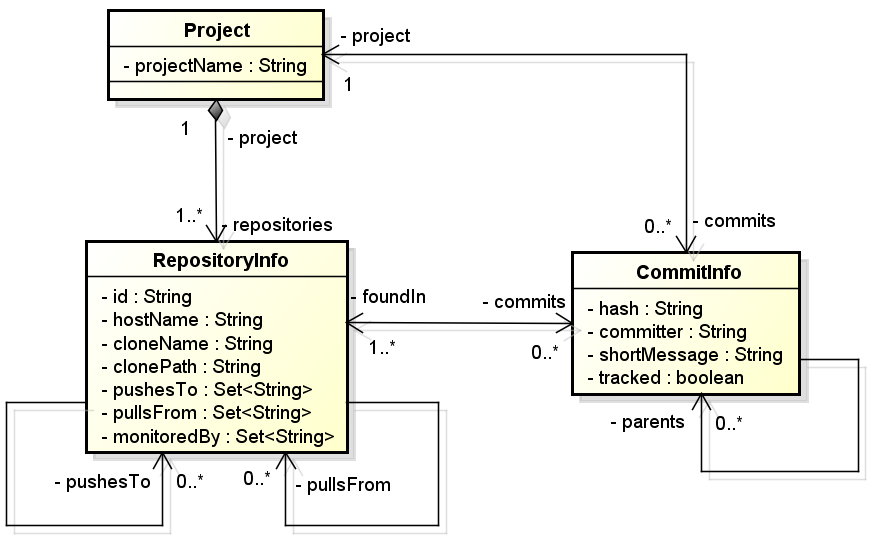


Fig. 3. Model used to store topology data

DyeVC has also the ability to gather information not only from the registered repositories in the user’s machine, but also from its partners, which are the repositories with which a given repository communicates. As there is a communication path between a registered repository and its partners (in order to push and pull data), we are able to synchronize the commits that exist in these partners. This allow us to visualize the entire topology and to know where each commit exists, even in nodes where the approach is not present, provided that these nodes communicate somehow with a node where the approach is present. Details on how data is gathered and how synchronization is performed are explained in sub-section III.C.

## Levels of Detail

Information gathered by DyeVC is presented at different levels of detail. Level 1 presents high-level notifications about the registered repositories; Level 2 present the whole topology of a given project. Level 3 zooms .Last, Level 4s

The following sections discuss each of these levels.

### Level 1: Notifications

In Level 1, our approach presents notifications whenever a change is detected in any registered repository or in any of their partners, allowing the user to start investigating what is occurring. Fig. 4 shows an example of this kind of notification.



Fig. 4. DyeVC showing notifications in the notification area

### Level 2: Topology

To help answering questions Q1 and Q2 from Section I, we present a topology view showing all repositories for a given system, as depicted in Fig. 5, where each node represents a known clone of the project. The clone from where the topology view was requested is presented as a blue computer and other clones where DyeVC is running are presented as black computers. Servers represent central repositories that do not pull from nor push to any other clone or clones where DyeVC is not running. The reason why the representation is the same for both kinds of nodes is that, once that DyeVC is not running at a given clone, our approach cannot infer if this clone pushes to or pulls from anyone. Thus, it will have empty push and pull lists and will be understood as a server. Finally, green checkmarks represent nodes that were selected to perform any operation, like moving it around the window to get a different visualization.

Each edge in the graph represents a relationship between two repositories. Edges with a continuous stroke mean that the source clone pushes to the destination clone. Edges with a dotted stroke mean that the destination clone pulls from the source clone. The edge labels show how many commits from the source clone are missing in the destination clone. If both clones are synchronized, than the edge connecting them is green, otherwise it is red.

### Level 3: Tracked branches

To answer question Q3 from Section I, our approach’s main screen (see Fig. 6) shows Level 3 information, allowing one to depict the status of each tracked branch between registered repositories and their partners. This information is complemented with that of Level 4, shown in next section.

The status evaluation considers the existing commits in each repository individually. Table I shows the possible states presented by DyeVC. Due to the nature of DVCS, old data is never deleted and commits are cumulative. Thus, if a commit N is created over a commit N – 1, the existence of commit N in a given repository implies that commit N – 1 also exists in the repository. With that said, by checking the existence of commits in the local repository not yet replicated to the remote repository, and vice-versa, it is possible to come up with one of the situations presented in Table II.

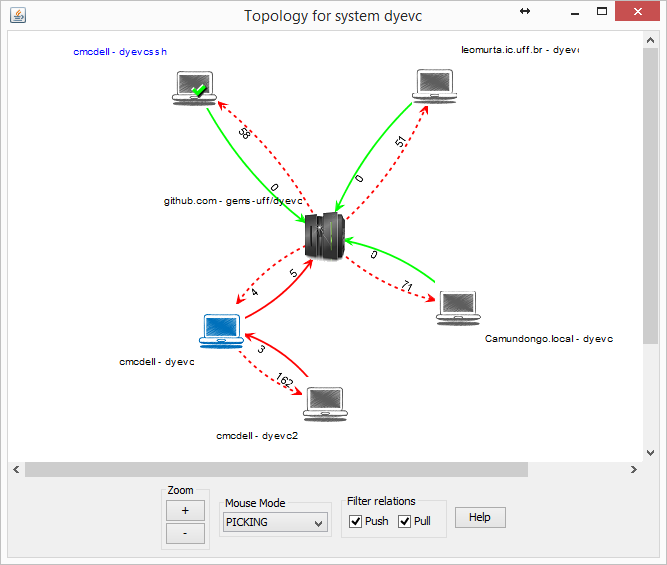


Fig. 5. Topology view for a given project



Fig. 6. DyeVC Main Screen

Table I

Possible States of a Repository

| Status | Description |
| --- | --- |
| question_32 | DyeVC has not analyzed the repository yet. |
| check_32 | Repository is synchronized with all partners. |
| ahead_ylw_32 | Repository has changes that were not sent yet to its partners (it is ahead its partners). |
| behind_ylw_32 | Partners have changes that were not sent yet to the repository (it is behind its partners). |
| aheadbehind_ylw_32 | Repository is both ahead and behind its partners. |
| nocheck_32 | Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user. |

To illustrate how this approach works, let us assume that each commit is represented by an integer number. At a giving moment, the local repositories of each developer from Fig. 1 have the commits shown in Table III.

Fig. 1Table IV

Table II

Status of a local repository regarding a remote one, based on the existence of non-replicated commits

| Existence of  non-replicated commits | | Local Status |
| --- | --- | --- |
| Local  Repository | Remote  Repository |
| Yes | Yes | aheadbehind_ylw_32Ahead and Behind (needs *pull* and *push*) |
| Yes | No | ahead_ylw_32Ahead (needs *push*) |
| No | Yes | behind_ylw_32Behind (needs *pull*) |
| No | No | check_32Synchronized |

Table III

Existing commits in each repository

| Repository | Wolverine | Gambit | Rogue | Nightcrawler | Beast |
| --- | --- | --- | --- | --- | --- |
| *Commits* | 10  11 | 10  11 | 10  12 | 10  11  13 | 10 |

Table IV

Status of each repository based on known remote repositories

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Repository | Wolverine | Gambit | Rogue | Nightcrawler | Beast |
| Wolverine | - | - | - | - | - |
| Gambit | check_32 | - | - | - | - |
| Rogue | - | aheadbehind_ylw_32 | - | - | - |
| Nightcrawler | - | - | - | - | - |
| Beast | - | - | behind_ylw_32 | behind_ylw_32 | - |

### Level 4: Commits

Level 4 complements information of Level 3 in order to provide an answer to Question Q3, by presenting a visual history of the repository (Fig. 7) as a graph. Each vertex in the graph represents a known commit for the same project, which is named after its hash’s five initial characters. A thicker border denotes that the commit is a branch’s head (e.g., commit f1a48).

For each commit, some information besides that shown in Fig. 3, such as branches that point to that commit and files that were affected (modified, deleted, inserted).

Each commit is painted according to its existence in the local repository and in the partners’ repositories. Ordinary commits that exist locally and in all partners are paint in white. Green commits are ready to be pushed, as they exist locally but do not exist in any partner in the push list. Yellow commits need attention, because they exist in at least a partner in the pull list, but do not exist locally, meaning that they may need to be pulled. Red commits do not exist locally and are not available to be pulled, as they exist only in repositories that are not partners. Finally, gray commits exist locally, but belong to non-tracked branches, meaning that they can neither be pushed nor pulled.



Fig. 7. Commit history for a given project



Fig. 8. Collapsed commit history

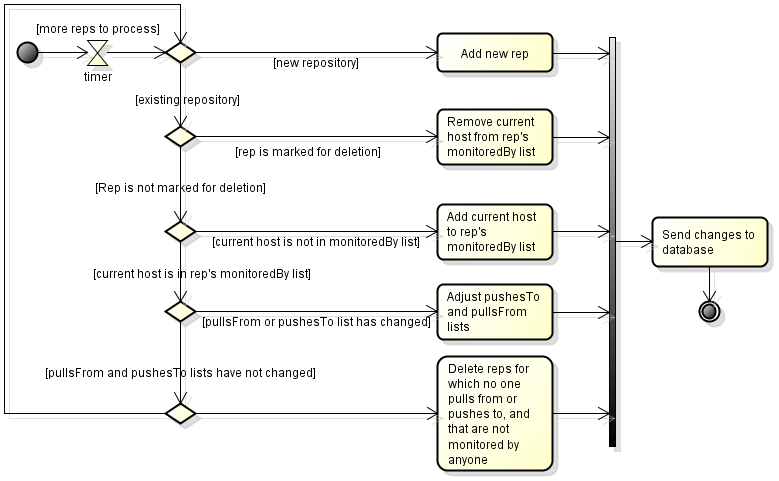


Fig. 9. Repository updates synchronization

## 

## Behind the Scenes

The synchronization that occurs behind the scenes involve depicting what are the existing repositories and commits in the topology. To synchronize repository information, we follow the process represented by activity diagram in Fig. 9. This process always begin with a list of repositories being monitored. For each given repository *rep*, we find which repositories rep pulls from and pushes to. These repositories are inserted or updated in the database, where we keep track of each hostname that references them (either by monitoring, pushing to or pulling from). Repositories that are not referenced anymore are from the topology.

Table V shows the algorithm used to synchronize commits. This synchronization finds out the existing commits and verifies where each of them can be found. The whole process is based on the Set Theory, and the sets created along the process are shown between parentheses. The “snapshot” mentioned is the set of existing commits at a local repository in a given moment and is used to identify both deleted and inserted commits. Commits to inserted or updated must be verified to check where they exist. This verification is done using the algorithm shown in Table VI. This algorithm finds out where each commit exists based on the ahead and behind lists, which are calculated periodically to show the number of commits ahead and behind (Level 3 information, see section III.B.3).

Table V

Commits synchronization algorithm

1. Retrieve previous local snapshot from disk (previousSnapshot)
2. Retrieve current snapshot from repository (currentSnapshot)
3. Identify new local commits, comparing current and previous snapshots (newCommits = currentSnapshot \ previousSnapshot)
   1. If previousSnapshot doesn’t exist, then newCommits = currentSnapshot
4. Retrieve number of existing commits from database (C)
5. If C is not 0, retrieve all commits from database that do not exist in at least one of the repositories related to rep, this is, commits that were not found either in rep or in at least one of its partners. All other commits in the database are considered to be already synchronized (commitsNotFoundInSomeReps)
6. If C is not 0, retrieve all commits from database that exist in newCommits (newCommitsInDatabase)
7. If C is not 0, identify which of newCommits commits do not exist in newCommitsInDatabase (commitsToInsert = newCommits \ newCommitsInDatabase)
   1. If C is 0 all commits should be sent to database (commitsToInsert = newCommits)
8. For each commit in commitsToInsert, update the list of repositories where it is found, according to the algorithm in Table VI
9. Insert commits from commitsToInsert in the database
10. Identify commits that were deleted locally since previous run: commitsToDelete = previousSnapshot \ currentSnapshot
11. If commitsToDelete is not empty, delete commits from commitsToDelete in the database
12. Identify local commits that were in non-tracked branches, that now are in tracked branches (nowTrackedCommits)
13. Update commits from nowTrackedCommits, changing tracked attribute to “true”
14. Remove deleted commits from commitsNotFoundInSomeReps (commitsToUpdate = commitsNotFoundInSomeReps \ commitsToDelete)
15. For each one of commitsToUpdate commits, update the list of repositories where it is found, according to the algorithm in Table VI
16. Update commits from commitsToUpdate that had the “foundIn” list changed in the database
17. Save currentSnapshot to disk

# Implementation

Our approach uses a client application launched by Java Web Start[[3]](#footnote-11) Technology, and focus on monitoring Git repositories, once that this is the most used DVCS nowadays [8]. The client application gathers information from repositories using JGit[[4]](#footnote-12) library, which allows the user to use our approach without having a Git client installed. Information gathered is stored in a central document database running MongoDB[[5]](#footnote-13). We present the information gathered as a series of graphs by using JUNG[[6]](#footnote-14) library, from which it inherits the ability to extend existing layouts and filters to create new ones, which can be dynamically attached to the graphs that it presents.

All graphs present similar behavior, allowing the window to be zoomed in or out, whether the user wants to see details of a particular area or an overview of the entire graph. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node that represents them, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines. There is also the possibility to collapse nodes to provide a better understanding of huge amounts of data. As shown in Fig. 8, the label of collapsed nodes show the number of contained nodes (there is a white node containing 118 nodes and a green node containing 24 nodes).

Table VI

Commit existence based on behind and ahead lists algorithm

1. If commit does not exist in any of the lists then it either:
   1. does not exist locally: then it does not exist in any of the repositories in the push / pull lists either; or
   2. exists locally: then its existence in other repositories will be based on whether it belongs to a tracked or to a non-tracked branch:
      1. if it belongs to a tracked branch, it means that it also exists in all repositories in the push / pull lists;
      2. if it belongs to a non-tracked branch, it means that it exists only locally;
2. If commit is in at least one behind list, then it does not exist locally and exists in all repositories that have at least one behind list containing it;
3. If commit is in at least one ahead list, then it exists locally and in all repositories in the push list that DO NOT have an ahead list containing it.

# Evaluation

In order to evaluate our approach we performed two experiments. First, we conducted a *post-hoc* analysis in a real project, to check if DyeVC can help answering questions Q1-Q3, posed in Section I. Next, we took some real projects of different sizes and from different sources, to evaluate the feasibility of processing huge amounts of information using our approach, in order to answer question Q4. The next sub-sections detail each of the experiments.

## Analyzing a real project with DyeVC

To demonstrate that our approach can help answering questions Q1-Q3, we conducted a *post-hoc* analysis using a real open source project. We used the JQuery project[[7]](#footnote-18), a project that began in 2006 and had 6,222 commits by the time of the evaluation. We used the repository history and reconstructed it, simulating the actions that occurred in the past. We do not replicate the repository history here, due to its size, but it is public available at the server that hosts JQuery project. Comments generated automatically helped us to depict specific flows. For example, the comment “**Merge branch 'master' of https://github.com/scottjehl/jquery into scottjehl-master**” tells us that there was a user named “scottjehl” and that the merge operation was done at a branch called “scottjehl-master”.

Due to the operating mode of Git, some details were missing, but these do not compromise our analysis. The first one is the moment where a named branch is removed. Named branches in Git are just pointers to a commit and they are not persistent. When a merge is performed between two branches, one of them disappears and we continue to have the pointer to just one of them. This does not affect our study, because we just do not need to know the existence of a named branch. The visual information in DyeVC’s commit history is sufficient to let us know that a branch existed.

The second missing detail was the moment when a clone arises or deceases, because this information does not exist anywhere. We inferred the creation of clones looking at the commit messages in the repository history (a commit by developer X led to the creation of a clone named X). Clones created at a given time stayed alive for the rest of the analysis.

The final missing detail was that although we had the commit dates and times in the repository history, these dates and times were not guaranteed to be correct. This is because in DVCS’s we do not have a central clock. Each commit is registered with the local time at the machine where the clone is located, which could lead us to have a commit in the history with a predecessor in the future, depending on when and where each one of them were performed. This missing detail is not relevant, because the precedence between two commits is not depicted from their commit times, but from the pointers that Git maintains from a commit to its ancestors. We can use these dates, but not as an authoritative information.



Fig. 10. Topology view showing first monitored repository



Fig. 11. aakoch’s commit history showing commits pending to be pushed

We chose a moment in time when three developers were involved, performing commits and merging changes in the repository. We created three clones for these developers, named after their author names and commit messages: *jeresig*, *adam* and *aakosh*. Fig. 10 shows the topology view when *aakosh* had 121 commits pending to be pushed to the central repository (hereafter represented as *central-repo*). Fig. 11 shows part of aakosh’s commit history and how DyeVC represents commits pending to be pushed as green nodes in the graph.

Later on, aakoch pushed his commits to *central-repo* and both *adam* and *jeresig* commited some changes. Fig. 12 shows the topology view after registering *adam* and *jeresig* to be monitored by DyeVC. Here, we can see that *aakoch* was synchronized with *central-repo*, whereas *adam* and *jeresig* had some pending actions. If one asked Q1 or Q2 at that time and participants used DyeVC, the answer would be the graph shown in Fig. 12.

Adam had 121 commits to push from *central-repo*, what is corroborated by the details of his tracked branches (Fig. 13). He also had a non-tracked commit pending to be pushed. Non-tracked commits are not shown in the tracked branches view, but we can see them in commit history views, painted in gray. Fig. 8 shows the collapsed commit history for jeresig, where we can see *adam’s* non tracked commit with hash a2bd8 (we know this is an *adam’s* commit by comparing the id of the repository in the message details with *adam’s* repository id in Fig. 12).



Fig. 12. Topology view showing the three monitored repositories



Fig. 13. Adam’s tracked branches

The repository history leads us to think that *jeresig* is a primary developer in the project, because he performed most of the merges to master branch. Looking at Fig. 12, we see that at that time he had 26 commits pending to be pushed to *central-repo*. These 26 commits can be seen at *aakoch’s* commit history (Fig. 15), as red commits, once that they could not be pulled by *aakoch* until *jeresig* pushed them to *central-repo*.There was also a commit in *central-repo* pending to be pulled by *jeresig*. This could not be commit a2bd8, because that commit was on a non-tracked branch at *adam’s* repository. If we look back at Fig. 8, we see that the only yellow commit is a0887. Fig. 8 does not show it, but this commit was made by *aakoch*. This tells us that *jeresig* pulled changes from *central-repo* at a moment before *aakoch* pushed commit a0887. If we look at Fig. 14, we see that all the pending commits (those that were pending to be pushed and those that were pending to be pulled) related to the same branch *master*. This tells us that, if *jeresig* wanted to push these commits to *central-repo*, he would have to perform both push and pull operations. This analysis helps us answer Q3.



Fig. 14. Jeresig’s tracked branches



Fig. 15. Aakoch’s commit history

## Applying DyeVC in projects with different characteristics

In order to answer question Q4, we evaluated the time spent to perform the most common DyeVC operations, by analyzing repositories of different sizes and hosted in different Git servers. The results are shown in Table VIII.

Table VIIImetrics

main of our approach: “”,invoked w. “I”,invoked w. “”, invoked wrequests to see. “”, invokedwwants to see. “”,invoked. “”,invoked periodicallythe central databaseThis last operation

Table VII e correlation between each repository metric and the measured operations, according to the *Pearson product-moment correlation coefficient* (PPMCC) [25]. This correlation coefficient measures the linear correlation between two variables X and Y and ranges from −1 to 1. Values of 1 or -1 mean that a linear equation can describe the correlation between X and Y perfectly (either positive or negative, respectively). A value of 0 means that there is no correlation between X and Y.

Looking at Table VII, we can see that is the metric with PPMCC nearest to 1Generally, took longer According to Table VIII, t“” “”, This is because the latter deals with much finer grain data than the former. The only operation with no significant variation in response times was “Check Branches”.

Table VII

PPMCC between measured operations and repository metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | # commits | Size | # files | # nodes |
| Insert 1st | 0,794961 | 0,652618 | 0,300709 | -0,44445 |
| Insert 2nd | 0,815497 | 0,645296 | 0,36267 | -0,46282 |
| Check Branches | 0,001227 | -0,28303 | -0,13048 | -0,25396 |
| Update Topology | 0,940141 | 0,169802 | 0,33009 | -0,35485 |
| Commit History | 0,94051 | 0,748865 | 0,425305 | -0,59183 |
| Topology | 0,861135 | 0,608855 | 0,592397 | -0,39253 |

# Related Work

According to [26] software visualization can be separated into three aspects: structure, behavior, and evolution. DyeVC relates primarily with the evolution aspect, more specifically with studies that aim at improving the awareness of developers that work with distributed software development. Awareness is defined by [27] as “an understanding of the activities of others to provide a context for one’s own activities”. A recent work by [28] presents a systematic review of such studies and classify them according to the Awareness Framework [29] and according to the 3C Collaboration Model [30]. The classification is not exclusive, i.e., a given tool can present elements of different awareness types. Following [29], DyeVC can be classified as a “Workspace Awareness” approach and according to [30], DyeVC fits into the “Coordination” and “Cooperation” categories.

Table VIII

Time (in seconds) spent by DyeVC in several operations for repositories with different histories and sizes

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Repository | Hosting | Repository metrics | | | | Foreground operations | | Background operations | | | |
| # commits | MB | # files | #nodes | Commit History | Topology | Insert 1st | Insert 2nd | Check Branches | Update Topology |
| DyeVC | github.com | 187 | 1.0 | 539 | 6 | 3.5 | 2.7 | 12.4 | 16.1 | 1.7 | 4.4 |
| Sapos | github.com | 702 | 7.0 | 685 | 11 | 5.6 | 3.2 | 20.8 | 22.6 | 1.8 | 5.2 |
| jgit | eclipse.org | 2979 | 10.0 | 1595 | 3 | 18.4 | 3.4 | 42.4 | 46.0 | 5.9 | 6.8 |
| egit | eclipse.org | 3775 | 27.0 | 1478 | 3 | 21.3 | 3.7 | 49.6 | 46.6 | 4.2 | 7.3 |
| jquery | github.com | 5518 | 20.0 | 253 | 3 | 65.0 | 4.1 | 40.0 | 37.4 | 1.4 | 9.4 |
| Tortoise Git | code.google.com | 6166 | 85.0 | 3220 | 3 | 68.0 | 4.2 | 39.0 | 36.0 | 1.6 | 9.6 |
| Gitextensions | github.com | 6417 | 448.0 | 1549 | 3 |  | 17.0 | 155.8 | 129.0 | 1.6 | 10.6 |
| drupal | drupal.org | 23922 | 84.4 | 9290 | 3 |  | 18.0 | 102.0 | 95.0 | 2.0 | 18.0 |
| Expresso Livre | gitorious.org | 25822 | 141.0 | 20729 | 3 |  | 18.2 | 110.0 | 102.0 | 2.1 | 19.3 |
| Git | github.com | 35260 | 98.0 | 2656 | 3 |  | 19.4 | 196.0 | 158.6 | 3.4 | 40.0 |

In order to find related work, we started with the referenced papers analyzed by [28] and used the snowball technique, looking at these papers’ citations and at papers that cited them. We also searched at the main academic digital libraries (ACM, IEEE, SpringerLink e ScienceDirect) and at the industry, using the keywords “revision”, “souce code”, “software configuration”, “source control”, “version control”, “application” and “system”, combined with “awareness”. We filtered the results found to get only studies that used any VCS. The resulting studies were divided into three groups. The first group includes tools that notify commit activities. The second group comprises of approaches that not only give the developer awareness of concurrent changes, but also inform them if conflicts were detected. Finally, the third group includes approaches that visualize repository information in a linear way.

The first group contains tools such as *SVN Notifier*[[8]](#footnote-19), *SCM Notifier*[[9]](#footnote-20), *Commit Monitor*[[10]](#footnote-21), *SVN Radar*[[11]](#footnote-22), *Hg Commit Monitor*[[12]](#footnote-23)and Elvin [31]*.* All of them focus on avoiding conflicts by increasing the developer’s perception of concurrent work. In addition, they are generally simple tools without extensive research or published work (except for [31]). They fail to identify related repositories and do not provide information in different levels of details, such as status, branches, and commits. DyeVC provides these different levels of details, as shown in Section III.B.

The second group includes tools such as *Palantir*, [17], *CollabVS* [19], *Crystal* [15], *Safe-Commit* [20], *Lighthouse* [14], *FASTDash* [18], and *WeCode* [21]. Among these studies, only *Crystal* and *FASTDash* work with DVCSs. *Crystal* detects physical, syntactic, and semantic conflicts in Git repositories (provided that the user informs the compile and test commands), but does not deal with repositories that pull updates from more than one partner*.* *FASTDash* directly, in repositories stored in Team Foundation Server[[13]](#footnote-24). According to the authors*FASTDash*. Although DyeVC primary focus is not to detect conflicts, it can be seen as a supporting infrastructure that can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

In the third group, the visualization can be done with different focuses, such as program structures [32], classes [33], lines [34], authors [35], and branch history [36]–[39]. The latter is the focus of DyeVC’s Commit History visualization.

Most of these works were applied only to CVCSs. The only exception found were [36], [38] and [39], which work with Git repositories, but look only at a local repository, not showing, for example, where a given commit can be found.

Regarding the topology view in DyeVC, we could not find any similar work.

# Conclusions and Future Work

Collaborative software development is a great challenge. If, on the one hand, parallelism in activities brings scale gains, on the other hand it tends to increase the concurrency, causing rework and productivity loss. The proliferation of branches and the distribution of repositories makes it difficult to realize parallel actions made by different developers. In this paper, we presented DyeVC: an approach that identifies the status of a repository in contrast with its partners, which are dynamically found in an unobtrusive way.

DyeVC allows one to have different views of a repository. The response time and scalability of each view varies according to the information granularity: the finer the granularity, the lower the performance.

A number of future research arises from this approach. Different visualizations can be developed to show the commit history, compacting it, for example, by automatically collapsing contiguous nodes that represent commits with the same level of accessibility. The ability to attach new layouts and filters allows the development of new visualizations, in order to present different metrics and views of the repository. These views may help answering the following questions: which repositories or which people changed a specific artifact or group of artifacts? Which commits introduced a higher amount of changes in the code? DyeVC could also work together with tools that provide awareness of existing and possible conflicts amongst work being made concurrently.

References

[1] M. J. Rochkind, “The source code control system,” *IEEE Trans. Softw. Eng. TSE*, vol. 1, no. 4, pp. 364–470, Dec. 1975.

[2] J. Estublier, “Software configuration management: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000, pp. 279–289.

[3] W. Tichy, “RCS: A system for version control,” *Softw. - Pract. Exp.*, vol. 15, no. 7, pp. 637–654, 1985.

[4] P. Cederqvist, *Version Management with CVS*. Free Software Foundation, 2005.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. Stanford, CA, USA, 2011.

[6] S. Chacon, *Pro Git*, 1st ed. Berkeley, CA, USA: Apress, 2009.

[7] B. O’Sullivan, *Mercurial: The Definitive Guide*, 1st ed. O’Reilly Media, 2009.

[8] Eclipse Foundation, “The Open Source Developer Report - 2013 Eclipse Community Survey,” San Francisco, CA, USA, Jun. 2013.

[9] C. Walrad and D. Strom, “The importance of branching models in SCM,” *Computer*, vol. 35, no. 9, pp. 31 – 38, Sep. 2002.

[10] B. O’Sullivan, “Making sense of revision-control systems,” *Commun. ACM*, vol. 52, no. 9, pp. 56–62, Sep. 2009.

[11] D.-C. Gumm, “Distribution Dimensions in Software Development Projects: A Taxonomy,” *IEEE Softw.*, vol. 23, no. 5, pp. 45–51, Sep. 2006.

[12] R. D. Battin, R. Crocker, J. Kreidler, and K. Subramanian, “Leveraging resources in global software development,” *IEEE Softw.*, vol. 18, no. 2, pp. 70–77, Mar. 2001.

[13] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel changes in large scale software development: an observational case study,” in *Proceedings of the 20th International Conference on Software engineering*, Washington, DC, USA, 1998, pp. 251–260.

[14] I. A. da Silva, P. H. Chen, C. Van der Westhuizen, R. M. Ripley, and A. van der Hoek, “Lighthouse: coordination through emerging design,” in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA, 2006, pp. 11–15.

[15] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, New York, NY, USA, 2011, pp. 168–178.

[16] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, “Streamed lines: Branching patterns for parallel software development,” in *Proceedings of the 1998 Pattern Languages of Programs Conference*, Monticello, Illinois, USA, 1998, vol. 98.

[17] A. Sarma and A. van der Hoek, “Palantir: coordinating distributed workspaces,” in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, Oxford, United Kingdom, 2002, pp. 1093 – 1097.

[18] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2007, pp. 1313–1322.

[19] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” in *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work*, 2007, pp. 159–178.

[20] J. Wloka, B. Ryder, F. Tip, and X. Ren, “Safe-commit analysis to facilitate team software development,” in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 507–517.

[21] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *Proceedings of the 2012 International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 342–352.

[22] R. Santos and L. G. P. Murta, “Evaluating the Branch Merging Effort in Version Control Systems,” in *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)*, Natal, RN - Brazil, 2012, pp. 151–160.

[23] C. M. Cesario and L. G. P. Murta, “What is going on around my repository?,” presented at the I Brazilian Workshop on Software Visualization, Evolution and Maintenance, Brasília, Brasil, 2013, pp. 14–21.

[24] J. I. Maletic, A. Marcus, and M. L. Collard, “A task oriented view of software visualization,” in *First International Workshop on Visualizing Software for Understanding and Analysis, 2002. Proceedings*, 2002, pp. 32–40.

[25] K. Pearson, “Note on Regression and Inheritance in the Case of Two Parents,” *Proc. R. Soc. Lond.*, vol. 58, no. 347–352, pp. 240–242, Jan. 1895.

[26] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, 2007 edition. Berlin ; New York: Springer, 2007.

[27] P. Dourish and V. Bellotti, “Awareness and Coordination in Shared Workspaces,” in *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*, New York, NY, USA, 1992, pp. 107–114.

[28] I. Steinmacher, A. Chaves, and M. Gerosa, “Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature,” *Comput. Support. Coop. Work CSCW*, pp. 1–46, May 2012.

[29] C. Gutwin, S. Greenberg, and M. Roseman, “Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation - Springer,” in *People and Computers XI*, M. A. Sasse, R. J. Cunningham, and R. L. Winder, Eds. London: Springer London, 1996, pp. 281–298.

[30] H. Fuks, A. Raposo, M. A. Gerosa, M. Pimentel, and C. J. Lucena, “The 3c collaboration model,” in *The Encyclopedia of E-Collaboration*, N. Kock, Ed. Information Science Reference, 2007, pp. 637–644.

[31] G. Fitzpatrick, P. Marshall, and A. Phillips, “CVS Integration with Notification and Chat: Lightweight Software Team Collaboration,” in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, New York, NY, USA, 2006, pp. 49–58.

[32] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, “A System for Graph-based Visualization of the Evolution of Software,” in *Proceedings of the 2003 ACM Symposium on Software Visualization*, New York, NY, USA, 2003, p. 77–ff.

[33] M. Lanza, “The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques,” in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001, pp. 37–42.

[34] L. Voinea, A. Telea, and J. J. van Wijk, “CVSscan: Visualization of Code Evolution,” in *Proceedings of the 2005 ACM Symposium on Software Visualization*, New York, NY, USA, 2005, pp. 47–56.

[35] E. Gilbert and K. Karahalios, “LifeSource: Two CVS Visualizations,” in *CHI ’06 Extended Abstracts on Human Factors in Computing Systems*, New York, NY, USA, 2006, pp. 791–796.

[36] S. Elsen, “VisGi: Visualizing Git branches,” in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[37] H. Koike and H.-C. Chu, “VRCS: integrating version control and module management using interactive three-dimensional graphics,” in *1997 IEEE Symposium on Visual Languages, 1997. Proceedings*, 1997, pp. 168–173.

[38] T. Hozumi, “Visugit.” [Online]. Available: https://github.com/hozumi/visugit. [Accessed: 04-May-2014].

[39] T. Preston-Werner, “GitHub’s Network Graph,” *GitHub*. [Online]. Available: https://github.com/blog/39-say-hello-to-the-network-graph-visualizer. [Accessed: 16-Feb-2014].

1. [↑](#footnote-ref-1)
2. Dye is commonly used in cells to observe the cell division process. As an analogy, DyeVC allows developers to observe how a Version Control repository evolved over time. [↑](#footnote-ref-2)
3. http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/ [↑](#footnote-ref-11)
4. http://www.eclipse.org/jgit/ [↑](#footnote-ref-12)
5. https://www.mongodb.org/ [↑](#footnote-ref-13)
6. http://jung.sourceforge.net/ [↑](#footnote-ref-14)
7. https://github.com/jquery/jquery.git [↑](#footnote-ref-18)
8. http://svnnotifier.tigris.org/ (2012) [↑](#footnote-ref-19)
9. https://github.com/pocorall/scm-notifier (2012) [↑](#footnote-ref-20)
10. http://tools.tortoisesvn.net/CommitMonitor.html (2013) [↑](#footnote-ref-21)
11. http://code.google.com/p/svnradar/ (2011) [↑](#footnote-ref-22)
12. http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009) [↑](#footnote-ref-23)
13. http://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx (2013) [↑](#footnote-ref-24)