Workspace Awareness for   
Distributed Version Control Systems

Cristiano M. Cesario and Leonardo G. P. Murta

Instituto de Computação,Universidade Federal Fluminense (UFF), 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 repository clones as well as the proliferation of several branches. In this paper, we propose DyeVC, a visualization infrastructure for distributed version control systems. It aims at enabling developers and repository administrators to understand how the repository and its clones evolve over time. DyeVC was evaluated over open source projects, showing how it could benefit the understanding of parallel development using distributed version control systems.

Keywords— Configuration management, Workspace awareness, Distributed version control, Repository evolution

# Introduction

Version 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 (e.g., SCCS and RCS [3]) to a client-server architecture (e.g., CVS [4] and Subversion [5]). More recently, distributed VCSs (DVCS) arose (e.g., Git [6] and Mercurial [7]) allowing clones of the entire repository in different locations. 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 among the open source community.

Besides these changes from local to client-server and then to distributed architecture, the concurrency control policy adopted by VCSs also changed from lock-based (pessimistic) to branch-based (optimistic). According with Walrad and Strom [9], creating branches in VCSs is essential to software development because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms / customers, among other features. 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.

However, distributed software development, especially from the geographical perspective [11], brings a set of risk factors, and Configuration Management (CM) is affected by them. 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. [12], concurrent development increases the number of defects in software. Besides, Silva et al. [13] say that branches are frequently used for promoting isolation amongst developers, postponing 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 DVCSs. Moreover, Brun et al. [14] 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 discussed by Appleton et al. [15], 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 by/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? Most of the existing works deal with question Q3, giving to the developers awareness of concurrent changes, such as Palantir [16], FASTDash [17], Lighthouse [13], CollabVS [18], Safe-Commit [19], Crystal [14] and WeCode [20]. But none of them deal with multiple branches.

To answer the above questions, we propose an approach named DyeVC[[1]](#footnote-1), 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 two-fold: increasing the developer knowledge of what is going on around his repository and the repositories of his teammates and enabling repository administrators to visualize how the existing repositories of a project interact with each other.

This paper expands the concepts presented in a previous workshop paper [21] by including a more thoroughly discussion about our approach, such as how we depict the relationship between the existing repositories in a project. It also includes an evaluation of DyeVC over open source projects, and a more detailed discussion about related work. This paper is organized as follows: Section II shows a motivational example to this work. Section III presents the DyeVC approach. Section IV presents the technologies used in our prototype implementation. Section V shows the evaluation of DyeVC. 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 the 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.



Fig. 1 A development scenario involving some developers

Each one of the developers has a complete copy of the repository. Luckily, this scenario has a CM Plan in action, otherwise each one would be able to send and receive updates to and from any other, leading to a total of 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. It occurs with Mystique and Nightcrawler, for example, where there is no direct communication.



Fig. 2 How DyeVC gathers information



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, in Gambit, or in Nightcrawler clones that were not pulled yet? Alternatively, would be the case that there are local commits pending to be pushed to Gambit? Beast could certainly periodically pull changes from his peers, checking if there were updates available, but this would be a manual procedure, prone to be forgotten. It would be more practical if Beast could have an up to date knowledge of his peers, warning him about any local or remote updates that had not been synchronized yet. On the other hand, from an administrator’s point of view, how can she know which are the existing clones of a project and how they relate among each other? How can she know if there are pending commits to be sent from a staging repository to a production one?

# DyeVC Approach

Aiming at supporting both developers and repository administrators in understanding the interaction of distributed repositories (clones), the main features of DyeVC include: (1) a mechanism to gather information from a set of repositories and (2) a set of extensible views with different levels of detail, which let DyeVC users visualize this information. We detail in the following sub-section how DyeVC gathers information from DVCSs. Next, we discuss how this information is presented using different levels of detail. Finally, we show what happens behind the scenes, discussing the algorithm involved in the data synchronization process.

## Information Gathering

DyeVC continuously gathers information from interrelated repositories, starting from repositories registered by the user. As shown in Fig. 2, data is gathered by DyeVC instances running at each user machine and is stored in a central document database. This way, information from one DyeVC instance is made available to every other instance in the topology. DyeVC reads information from the original repositories without changing them, and process this information at the central database or in working copies of the local repositories, which are stored in the user’s home folder.

DyeVC gathers information not only from the registered repositories in the user’s machine, but also from its peers, which are the repositories that a given repository communicates. Since there is a communication path between a registered repository and its peers (in order to push and pull data), we are able to analyze the commits that exist in these peers. This allow us to visualize the entire topology and to know where each commit exists, even in peers where DyeVC is not running, but communicates somehow with a peer running DyeVC. Details on how data is gathered are explained in sub-section C.



Fig. 3 Model used to store DyeVC data

The data stored at the central database follows the model presented in Fig. 3. A *Project* groups all repository clones of the same system. Repositories are stored as *RepositoryInfo* and are identified by an idand a meaningful clone name provided by the user. A *RepositoryInfo* has a list of clones to which it pushes to and a list of clones from which it pulls from. These lists are represented respectively by the self-associations *pushesTo* and *pullsFrom*. Finally, a *RepositoryInfo* stores the hostname where it resides (e.g., a server name or *localhost*) and its path (be it an operating system path or a URL).

Another element in Fig. 3 is the *Branch*. Branches are part of a *RepositoryInfo*. A *Branch* instance has a name and a boolean attribute *isTracked*, which is true if the branch tracks a remote branch. A *RepositoryInfo* may have one or many branches (it must have at least one branch, which is the main one). A *Branch* has two associations with *CommitInfo*: through the first association, a *Branch* knows which commit is its head and, conversely, a commit knows which branches point to it as a head. The second association represents which commits are reachable from a given branch and, conversely, the branches from which the commit is reachable.

The finer grain of information is the *CommitInfo*, which represents each commit in the topology. A commit is identified by a hash code and it refers to its parents (except for the first commit in the repository, which does not have any parent). As each commit may not exist in all repositories of the topology, we store the list of repositories where each commit can be found (*foundIn* association). We also store the committer, the commit message, and the information whether the commits belongs to tracked branches or to non-tracked branches.

## Information Visualization



Fig. 4 DyeVC showing notifications in the notification area

The visualization of information gathered by DyeVC is divided into four different levels of detail: Level 1 presents high-level notifications about the registered repositories; Level 2 presents the whole topology of a given project. Level 3 zooms into the branches of the repository, showing the status of each tracked branch. Lastly, Level 4 zooms into the commits of the repository, showing a visual log with information about each commit. The following sections discuss each of these levels.



Fig. 5 Topology view for a given project

### Level 1: Notifications

In Level 1, our approach periodically monitors the registered repositories and presents notifications whenever a change is detected in any known peer. The period between subsequent runs is configurable, and the notifications are presented in the system notification area, in a non-obtrusive way. Fig. 4 shows an example of this kind of notification, where DyeVC detected changes in three different repositories. Clicking on the balloon allows the user to start investigating what changed.

### Level 2: Topology

Aiming at helping answering questions Q1 and Q2 from Section I, we present a topology view showing all repositories for a given project, as depicted in Fig. 5, where each node represents a known clone of the project. A blue computer represents the current user clone and black computers represent other clones where DyeVC is running. Servers represent central repositories, that do not pull from nor push to any other clone, or clones where DyeVC is not running. The representation is the same for both kinds of nodes because, once DyeVC is not running at a given clone, we cannot infer if the clone pushes to or pulls from anyone else. Thus, it will have empty push and pull lists and will be understood as a server.

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 two numbers separated by a dash. The first number represents how many commits in tracked branches of the source clone are missing in the destination clone. The second number represents how many commits in non-tracked branches of the source clone are missing in the destination clone. The edge colors are used to represent the synchronization status: green edges mean that both clones are synchronized (i.e., both clones have the same set of commits), whereas red edges mean that the pair is not synchronized.

For example, it is possible to observe in Fig. 5 that the current user clone (blue computer) is hosted at *cmcdell* and it is named *dyevc*. This clone pulls from *gems-uff/dyevc*, which is located at *github.com*, and there are four tracked commits ready to be pulled. It also pushes to the same peer, having five tracked commits ready to be pushed.

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 peers. |
| ahead_ylw_32 | Repository has changes that were not sent yet to its peers (it is ahead its peers). |
| behind_ylw_32 | Peers have changes that were not sent yet to the repository (it is behind its peers). |
| aheadbehind_ylw_32 | Repository is both ahead and behind its peers. |
| nocheck_32 | Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user. |



Fig. 6 DyeVC main screen

### Level 3: Tracked branches

To answer question Q3 from Section I, our DyeVC’s main screen (see Fig. 6) shows Level 3 information, allowing one to depict the status of each tracked branch among registered repositories and their peers. This information is complemented with that of Level 4, shown in the 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 almost never deleted and commits are cumulative. Only in rare situations, such as removal of sensitive data mistakenly committed into the repository, commits are deleted. 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 this 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.

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 led by Wolverine (Fig. 1) have the commits shown in Table III. Considering just the synchronization paths presented in Fig. 1, the perception of each developer regarding to his known peers is shown in Table IV. Notice that the perceptions are not symmetric. For instance, as Gambit does not pull updates from Nightcrawler, there is no sense in giving him information regarding Nightcrawler.

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 | - | check_32 | aheadbehind_ylw_32 | - | - |
| Beast | - | behind_ylw_32 | 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 which the balloon is showing additional information).

Commits are drawn according to their precedence order. Thus, if a commit N is created over a commit N – 1, then commit N will be located in the right hand side of commit N – 1. For each commit, DyeVC presents the information described in Fig. 3 (gathered from the central database), along with information that is read in real time from the repository metadata, such as branches that point to that commit and files that were affected (modified, deleted, inserted).

It is important to notice that this visualization contains all commits of all clones in an integrated graph. Each commit is painted according to its existence in the local repository and in the peers’ repositories. Ordinary commits that exist locally and in all peers are paint in white. Green commits are ready to be pushed, as they exist locally but do not exist in any peer in the push list. Yellow commits need attention because they exist in at least one peer in the pull list, but do not exist locally, meaning that they may be pulled. Red commits do not exist locally and are not available to be pulled, as they exist only in repository clones that are not peers. Finally, gray commits exist locally (either on the user’s computer or on a partner’s computer), but belong to non-tracked branches, meaning that they can neither be pushed nor pulled.

There is also the possibility to collapse nodes manually to provide a better understanding of huge amounts of data. As shown in Fig. 8, the label of collapsed nodes shows the number of contained nodes (there is a white node containing 118 commits and a green node containing 24 commits).



Fig. 7 Commit history for a given project

## Behind the Scenes



Fig. 8 Collapsed commit history

What DyeVC does behind the scenes involves depicting what are the existing repository clones and commits in the topology. To update repository information, we 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. In other words, the final list of repositories comprise hosts where DyeVC is configured or hosts referenced in the push to or pull from lists of configured repositories. Repositories that are not referenced anymore are removed from the topology.

Algorithm 1 shows the algorithm to update commits in the topology. This update finds out the existing commits and checks where they can be found. The process is based on Set Theory and is executed for each repository *rep* monitored by DyeVC. The algorithm receives the repository being monitored (*rep*), the set of existing commits in the database (*db.commits*), the set of existing commits at *rep* at the previous monitoring cycle (*previousSnapshot*) and the set of existing commits at *rep* at the current monitoring cycle (*currentSnapshot*).

First of all, we subtract *currentSnapshot* from *previousSnapshot* to find *commitsToDelete*, that contains commits that were deleted since the previous monitoring cycle (line 2) and we delete them from the database, in order to cover the rare situations where a commit is deleted (line 3). Conversely, we subtract *previousSnapshot* from *currentSnapshot* to find *newCommits*, which contains commits that are new in *rep* since the previous monitoring cycle (line 5).

Next, we find out which of *newCommits* will have to be inserted into the database, by subtracting the existing commits in the database (db.*commits*) from *newCommits* (line 6). This step is necessary because some of the new commits might already been inserted into the database by another instance of DyeVC running elsewhere. Commits that might be updated are represented by *commitsToUpdate* (line 7) and they consist of those commits that exist in the database, but were not found in at least one of the repositories related to *rep* on the last monitoring cycle. These commits must be verified because since the previous monitoring cycle it may happen that they now are found in other repositories related to *rep*.

Commits to be inserted or updated must be verified to check where they exist, thus updating the *c.foundIn* attribute. This verification is done using the procedure *updateFoundIn* (lines 23-45), which is called in lines 10 and 14. This procedure finds out where each commit *c* exists based on its existence locally or in any repository in the push or pull sets. This procedure verifies if *rep* is ahead of any repository in its push list regarding *c* (line 24), i.e., if *c* exists and if there is at least one repository that *rep* pushes to that does not contain *c*. Likewise, it verifies if *rep* is behind of any repository in its pull list regarding *c* (line 25), i.e., if *c* does not exist locally and if there is at least one repository that *rep* pulls from that contains *c*. If *rep* is behind, then all repositories in *rep’s* pull list that contain *c* are added to *c*.*foundIn* (lines 27-29). If *rep* is ahead, then *rep* and all repositories in *rep’s* push list that contain *c* are added to *c.foundIn* list (lines 30-32). It may happen that *rep* is neither ahead nor behind any repository (line 33)*.* In such case, one of the following three scenarios may happen: In scenario 1, *c* does not exist in current snapshot (line 34), meaning that it also does not exist in any of the related repositories, thus we remove *rep* and all its related repositories from *c.foundIn* (line 35). For scenarios 2 and 3, we first depict if *c* is reachable from a tracked branch, i.e., if at least one of *rep.branches* is tracked and has *c* as one of its elements (line 37). In scenario 2, *c* is in a tracked branch, meaning that it also exists in all related repositories (remember that *rep* is neither ahead nor behind their partners), thus we include *rep* and all its related repositories in *c.foundIn* (line 39). Finally, in scenario 3, *c* is not in a tracked branch, meaning that it exists only in *rep*, thus we include only *rep* in *c.foundIn* (lines 41).

After updating where each commit is found, commits in *commitsToInsert* are inserted into the database (line 17) and commits in *commitsToUpdate* are updated in the database (line 18). Finally, it may happen that some commits end up with an empty *foundIn* attribute, meaning that they do not exist anywhere in the topology (line 19). These so-called *orphanedCommits* are then removed from the database (line 20) and the algorithm ends.

**Algorithm 1:** Updating commits in the topology

**input**: a *RepositoryInfo* *rep* representing the repository being analyzed and three sets of *CommitInfo* *db.commits*, *previousSnapshot* and *currentSnapshot*.



# Implementation

We implemented our approach as a Java application launched by Java Web Start Technology. It currently monitors Git repositories, as it is the most used DVCS nowadays [8]. The source code and the link to download the tool via Java Web Start can be found at GitHub. The application gathers information from repositories using JGit library[[2]](#footnote-2), 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.

We hosted our database on a free MongoDB instance provided by MongoLab. We did not use MongoDB proprietary API, which would demand opening specific ports to connect to MongoDB. Instead, we opted to use MongoLab’s RESTful (*Representational State Transfer*) API. RESTful APIs [22] have the advantage of being available using standard HTTP and HTTPS protocols. This way, our approach can be used from inside corporate and academic environments, without major problems. In order to use this RESTful API, we implemented a *MongoLabProvider*, which translates the application methods into RESTful commands and vice-versa. It also serializes and deserializes the application objects to and from JSON (*JavaScript Object Notation*) representations to be sent and received through the RESTful commands.

We present the information gathered as a series of graphs by using JUNG (*Java Universal Network/Graph*) library[[3]](#footnote-3), from which DyeVC inherits the ability to extend existing layouts and filters. 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, or simply drag them into new positions to have a better understanding parts with too many crossing lines.

# Evaluation

In order to evaluate our approach we performed two experiments. First, we conducted a *post-hoc* analysis in a real project, aiming at checking 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, aiming at evaluating the scalability of our approach.



Fig. 9 First monitored repository in Topology view (Sep 24 2010)

## Analyzing a real project with DyeVC

We conducted a *post-hoc* analysis using a real open source project to demonstrate that our approach can help answering questions Q1-Q3. We used the JQuery project[[4]](#footnote-4), 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. Automatically generated comments 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 are missing, but these do not compromise our analysis. The first one is the moment when a clone arises or deceases. This information does not exist anywhere in the repository. We inferred the creation of clones by looking at the commit messages (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 second missing detail is 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 so important, 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 parents. We can use these dates, but not as an authoritative information.



Fig. 10 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: *jeresig*, *adam*, and *aakosh*. Fig. 9 shows the topology view on Sep 24 2010, when *aakosh* had 121 commits pending to be pushed to the central repository (hereafter represented as *central-repo*). Fig. 10 shows part of *aakosh’s* commit history and how DyeVC represents commits pending to be pushed (green nodes).



Fig. 11 Three monitored repositories in Topology view (Sep 27 2010)

Later on, *aakoch* pushed his commits to *central-repo.* In the meantime, both *adam* and *jeresig* commited some changes. Before they pushed their work to *central-repo*, *adam’s* last commit had been done on Jun 21 2010 and *jeresig’s* on Sep 27 2010.At this moment, we registered them to be monitored by DyeVC. Fig. 11 shows the topology view after this registration on Sep 27 2010. Here, we can see that *aakoch* was synchronized with *central-repo*, whereas *adam* and *jeresig* had pending actions. At this point, we can revisit questions Q1 and Q2:

Q1: *Which clones were created from a repository*? DyeVC’s topology view (Fig. 11) shows all the clones where there is an instance running, and also discovers other clones connected to them, even if there is no instance running.

Q2: *What are the dependencies between different clones?* DyeVC’s topology view (Fig. 11) shows the dependencies between the peers in the topology, as well as the number of commits ahead or behind in each of these dependencies.

*Adam* had 121 commits to pull from *central-repo*, what is corroborated by the details of his tracked branches (master branch in Fig. 12). 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 the 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*.

The repository history leads us to think that *jeresig* is a core developer of this project, because he performed most of the merges to master branch. Looking at Fig. 11, we see that he had 26 commits pending to be pushed to *central-repo*. These 26 commits can be seen at *aakoch’s* commit history (Fig. 13) as red commits, once 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*. If we look back at Fig. 8 we see that the only yellow commit is a0887, 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) are 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 a pull operation before. This analysis helps us revisit and answer Q3.



Fig. 12 Adam’s tracked branches

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?* New commits in tracked branches of peers can easily be found looking at Level 3 information (tracked branches, shown in Fig. 12 and Fig. 14). This view shows to which branch these commits are related and how many new commits exist. If we want to look at each commit individually, we can look at Level 4 information (commit history, shown in Fig. 10 and Fig. 13) and notice the yellow nodes. Additionally, Level 4 information is used to find new commits in repositories that are not peers (red nodes), or new commits in non-tracked branches (gray nodes).



Fig. 13 Aakoch’s commit history

## Performance evaluation

In order to evaluate the scalability of our approach, we evaluated the time spent to perform the most common DyeVC operations, by analyzing repositories of different sizes and hosted in different Git servers. Table V shows the monitored projects (name and hosting service), the repository metrics – in terms of number of commits, disk usage, and number of files – and the time spent to run some background and foreground operations. All measurements were done in the same period of the day and from the same machine, a Core Duo CPU running at 2.53 GHz, with 4GB RAM running Windows 8.1 Professional 64 bits, connected to the internet at 35 Mbit/s.



Fig. 14 Jeresig’s tracked branches

We measured the main operations of our approach: “Insert 1st”, invoked when the user includes the first repository of a given system to be monitored. “Insert 2nd”, invoked when the user includes a repository to be monitored in a system that already have registered repositories. “Commit History”, invoked when the user requests to see the commit history of a given repository. “Topology”, invoked when the user wants to see the topology of repositories of a given system. “Check Branches”, invoked periodically to check all the monitored repositories, searching for ahead or behind commits. “Update Topology”, invoked periodically to update the topology information in the central database. This last operation updates the existing repositories, their peers, and the existing commits, marking in which repositories each commit is found.

It is possible to notice that the “Commit History” operation has no values for the last three repositories. This occurs because, as the number of commits increases, more memory is used to calculate the commit history graph. The current algorithm has an *O(x2*) space complexity (*x* being the number of commits). Our test machine was configured with a 2 GB maximum Java Heap Size, which let us analyze repositories with up to 6K commits. This is an aspect for future improvements.

Table VI shows the correlation between each repository metric and the measured operations, according to the Pearson coefficient [23]. 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 linear correlation between X and Y.

Looking at Table VI, it is possible to notice that, except for the “Check Branches” operation, all other operation times are strongly dependent on the number of commits. This is due to the nature of these operations, which update or show information about all commits in the repository.

## Threats to validity

Table V. Scalability results of DyeVC for repositories with different metrics

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Repository | Hosting | Repository metrics | | | Foreground operations | | | Background operations times (s) | | | |
| # commits | Size (MB) | # files | Commit History | | Topology | Insert 1st | Insert 2nd | Check  Branches | Update  Topology |
| Time (s) | Memory Usagea | Time (s) |
| DyeVC | github.com | 187 | 1.0 | 539 | 3.5 | 15 | 2.7 | 12.4 | 16.1 | 1.7 | 4.4 |
| SAPOS | github.com | 702 | 7.0 | 685 | 5.6 | 19 | 3.2 | 20.8 | 22.6 | 1.8 | 5.2 |
| jgit | eclipse.org | 2,979 | 10.0 | 1,595 | 18.4 | 512 | 3.4 | 42.4 | 46.0 | 5.9 | 6.8 |
| egit | eclipse.org | 3,775 | 27.0 | 1,478 | 21.3 | 559 | 3.7 | 49.6 | 46.6 | 4.2 | 7.3 |
| jquery | github.com | 5,518 | 20.0 | 253 | 65.0 | 1,121 | 4.1 | 40.0 | 37.4 | 1.4 | 9.4 |
| Tortoise Git | code.google.com | 6,166 | 85.0 | 3,220 | 68.0 | 492 | 4.2 | 39.0 | 36.0 | 1.6 | 9.6 |
| Gitextensions | github.com | 6,417 | 448.0 | 1,549 | 73.0 | 1,529 | 17.0 | 155.8 | 129.0 | 1.6 | 10.6 |
| Drupal | drupal.org | 23,922 | 84.4 | 9,290 |  |  | 18.0 | 102.0 | 95.0 | 2.0 | 18.0 |
| Expresso Livre | gitorious.org | 25,822 | 141.0 | 20,729 |  |  | 18.2 | 110.0 | 102.0 | 2.1 | 19.3 |
| Git | github.com | 35,260 | 98.0 | 2,656 |  |  | 19.4 | 196.0 | 158.6 | 3.4 | 40.0 |

1. Memory usage was measured in MB during the execution of “Commit History” operation.

While we have taken care to minimize threats to the validity of the experiment, some factors can influence the results. The usage of a *post-hoc* analysis to evaluate a real project may not reflect the exact real sequence of events that occurred, although the outcome did not change. For example, when we say that *aakosh*, at some moment, had 121 commits pending to be pushed to the central repository, these commits could have been pushed at once, or by a series of smaller pushes.

Moreover, only one project was selected to perform the *post-hoc* analysis, what imposes limitations from a statistical standpoint. Furthermore, there is a risk regarding the instrumentation used to measure the response times during the performance evaluation. As we used a database stored over the Internet, the response times may have been negatively affected by connectivity issues and network instability.

Table VI. Pearson coefficient between measured operations and repository metrics

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | # commits | Size | # files |
| Insert 1st | 0.79 | 0.65 | 0.30 |
| Insert 2nd | 0.82 | 0.65 | 0.36 |
| Check Branches | 0.00 | -0.28 | -0.13 |
| Update Topology | 0.94 | 0.17 | 0.33 |
| Commit History | 0.95 | 0.62 | 0.41 |
| Topology | 0.86 | 0.61 | 0.59 |

Finally,we used an open source project to perform the *post-hoc* analysis. However, the *modus operandi* of peers in this context may be different from that of peers in academic or industrial contexts. Besides that, it is not possible to represent all possible situations of a real project. We discussed the most common situations that occur when using DVCSs, but a more thoroughly verification is needed to evaluate the usefulness of our approach in other situations.

# Related Work

According to Diehl [24], 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. A recent work by Steinmacher [25] presents a systematic review of awareness studies which we used to perform a snowball sampling. The studies found were divided into three groups. The first group includes tools that notify commit activities. The second group comprises approaches that not only give the developer awareness of concurrent changes, but also inform them about conflicts. Finally, the third group includes approaches that visualize repository information in a linear way.

The first group contains tools such as SVN Notifier[[5]](#footnote-5), SCM Notifier[[6]](#footnote-6), Commit Monitor[[7]](#footnote-7), SVN Radar[[8]](#footnote-8), Hg Commit Monitor[[9]](#footnote-9) and Elvin [26]. The primary focus of these approaches is on increasing the developer’s perception of concurrent work by showing notifications whenever other developers perform actions. The approaches in this group do not 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 comprises approaches that give the developer awareness of concurrent changes, sometimes informing them if conflicts were detected. This group includes tools such as Palantir, [16], CollabVS [18], Crystal [14], Lighthouse [13], FASTDash [17], and WeCode [20]. 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 compiling and testing commands), but does not deal with repositories that pull updates from more than one peer. FASTDash does not detect conflicts directly, as the previous cited studies, but provides awareness of potential conflicts, such as two programmers editing the same region of the same source file in repositories stored in Microsoft Team Foundation Server. 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.

Finally, the third group includes approaches that visualize repository information. Each approach has a different visualization focus, such as program structures [27], classes [28], lines [29], authors [30], and branch history [31]–[33]. The latter is the focus of DyeVC’s Commit History visualization. Most of these works were applied only to CVCSs. The only exceptions found were [31], [32], and [33], which work with Git repositories, but look only at the local repository, not showing, for example, where a given commit can be found. It is worth noticing that we could not find any similar work showing the dependencies among several clones of a DVCS.

# Conclusions and Future Work

In this paper, we presented DyeVC: an approach that identifies the status of a repository in contrast with its peers, which are dynamically found in an unobtrusive way. We have evaluated DyeVC on a real project, showing that it can be used to answer questions that arise when working with DVCSs. We have also evaluated DyeVC’s performance when used with repositories of different sizes, and we found out that the time and space complexity of the approach are directly related to the number of commits in the repository under analysis, especially in the view levels with finer granularity.

A number of future researches arise from this work. 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 among work being made concurrently. Finally, some optimization should be done to allow DyeVC work with larger repositories.

##### Acknowledgement

We thank CNPq and FAPERJ for the financial support.

##### References

[1] M. J. Rochkind, “The source code control system,” *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 364–470, Dec. 1975.

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

[3] W. Tichy, “RCS: A system for version control,” *Software - Practice and Experience*, 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, USA: Compiled from r4849, 2011.

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

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

[8] Eclipse Foundation, “The Open Source Developer Report - 2013 Eclipse Community Survey,” Eclipse Foundation, San Francisco, USA, Survey, 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,” *CACM*, vol. 52, no. 9, pp. 56–62, Sep. 2009.

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

[12] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel changes in large scale software development: an observational case study,” in *ICSE*, 1998, pp. 251–260.

[13] 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 *OOPSLA,*  2006, pp. 11–15.

[14] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *ESEC/FSE*, 2011, pp. 168–178.

[15] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, “Streamed lines: Branching patterns for parallel software development,” in *PLoP*, 1998, vol. 98.

[16] A. Sarma and A. van der Hoek, “Palantir: coordinating distributed workspaces,” in *COMPSAC,* 2002, pp. 1093 – 1097.

[17] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *CHI,* 2007, pp. 1313–1322.

[18] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” in *ECSCW*, 2007, pp. 159–178.

[19] J. Wloka, B. Ryder, F. Tip, and X. Ren, “Safe-commit analysis to facilitate team software development,” in *ICSE,* 2009, pp. 507–517.

[20] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *ICSE*, 2012, pp. 342–352.

[21] C. M. Cesario and L. G. P. Murta, “What is going on around my repository?,” presented VEM, 2013, pp. 14–21.

[22] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” University of California, Irvine, USA, 2000.

[23] 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.

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

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

[26] G. Fitzpatrick, P. Marshall, and A. Phillips, “CVS Integration with Notification and Chat: Lightweight Software Team Collaboration,” in *CSCW*, 2006, pp. 49–58.

[27] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, “A System for Graph-based Visualization of the Evolution of Software,” in *SOFTVIS*, 2003, p. 77–ff.

[28] M. Lanza, “The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques,” in *IWPSE*, 2001, pp. 37–42.

[29] L. Voinea, A. Telea, and J. J. van Wijk, “CVSscan: Visualization of Code Evolution,” in *SOFTVIS*, 2005, pp. 47–56.

[30] E. Gilbert and K. Karahalios, “LifeSource: Two CVS Visualizations,” in *CHI*, 2006, pp. 791–796.

[31] S. Elsen, “VisGi: Visualizing Git branches,” in *VISSOFT*, 2013, pp. 1–4.

[32] T. Hozumi, “Visugit,” 2010. [Online]. Available: https://github.com/hozumi/visugit. [Accessed: 04-May-2014].

[33] T. Preston-Werner, “GitHub’s Network Graph,” *GitHub*, 10-Apr-2008. [Online]. Available: https://github.com/blog/39-say-hello-to-the-network-graph-visualizer. [Accessed: 16-Feb-2014].

1. 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-1)
2. http://www.eclipse.org/jgit/ [↑](#footnote-ref-2)
3. http://jung.sourceforge.net/ [↑](#footnote-ref-3)
4. https://github.com/jquery/jquery.git [↑](#footnote-ref-4)
5. http://svnnotifier.tigris.org/ (2012) [↑](#footnote-ref-5)
6. https://github.com/pocorall/scm-notifier (2012) [↑](#footnote-ref-6)
7. http://tools.tortoisesvn.net/CommitMonitor.html (2013) [↑](#footnote-ref-7)
8. http://code.google.com/p/svnradar/ (2011) [↑](#footnote-ref-8)
9. http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009) [↑](#footnote-ref-9)