Awareness over Distributed Version Control Systems

Cristiano M. Cesario and Leonardo G. P. Murta

Institute of Computing

UFF - 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.

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, 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 computationally 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. This perception of concurrent changes is known as “awareness”, and it is defined by [23] as “an understanding of the activities of others to provide a context for one’s own activities”.



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 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.

Fig. 1 A development scenario involving some developers

This paper expands the concepts presented in a previous workshop paper [24] by including a more thoroughly discussion about our approach, as how we depict the relationship between the existing repositories in a project. It also includes the analysis we have done to evaluate DyeVC over open source projects, and a more detailed discussion about 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 **Erro! Fonte de referência não encontrada.** discusses some related work and Section **Erro! Fonte de referência não encontrada.** concludes the paper and presents some future work.

# Motivational Example

Fig. 1shows 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. Luckily, 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, 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. It occurs with Mystique and Nightcrawler, where there is no direct communication.

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 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 peers, 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 peers, and constantly monitored those, warning Beast of any local or remote updates that had not been synchronized yet?

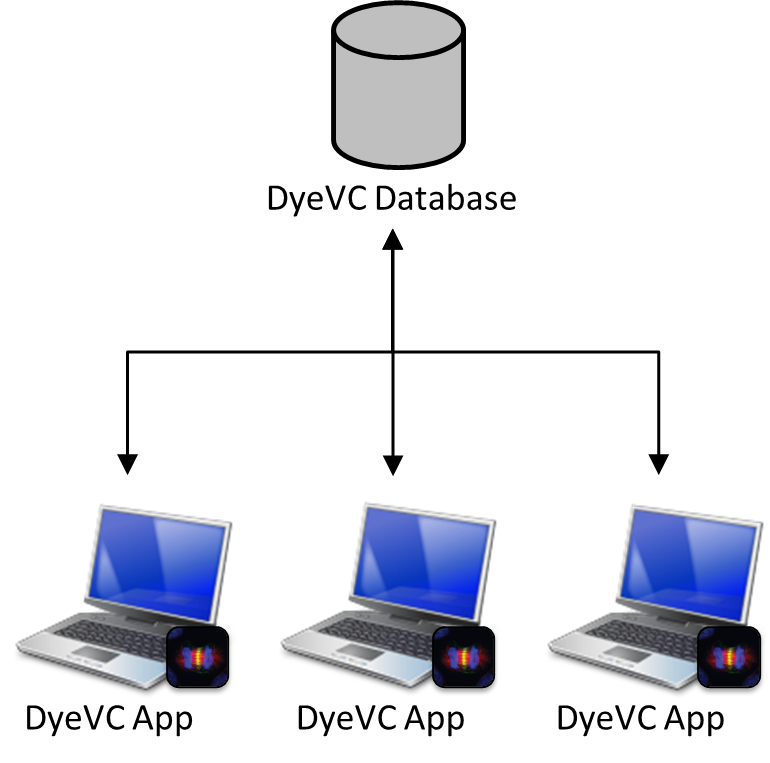
From an administrator’s point of view, how can he know which are the existing clones of a project and how they relate amongst each other? How can he know if there are pending commits to be send from a staging repository to a production one?

# DyeVC Approach

As we mentioned in Section I, the goal of DyeVC is 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.

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 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, discussing the algorithm 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. As shown in Fig. 2, data is gathered by DyeVC instances running at each user machine and it is stored in a central document database. This way, information from one DyeVC instance is made available to every other instance in the topology. It is important to notice that DyeVC does not change anything in the local repositories. The approach only reads information in the repositories, and all the work is done at the central database or in working copies of the local repositories.

DyeVC has also the ability to gather 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. As 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, provided that these peers communicate somehow with a peer running DyeVC. Details on how data is gathered are explained in sub-section C.

The data stored at the central database follows the model presented in Fig. 4. A *Project* is used to group all repository clones of the same system. Repositories are stored as *RepositoryInfo* and are identified by an idand the user can 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. These lists are represented respectively by the self-associations *pushesTo* and *pullsFrom*.



Fig. 2 How DyeVC gathers information

Fig. 3 DyeVC showing notifications in the notification area



Figure 2

Another element in Fig. 4 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 can refer to its parents, in the case of a merge. 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.

Fig. 4 Model used to store data

## Information Visualization

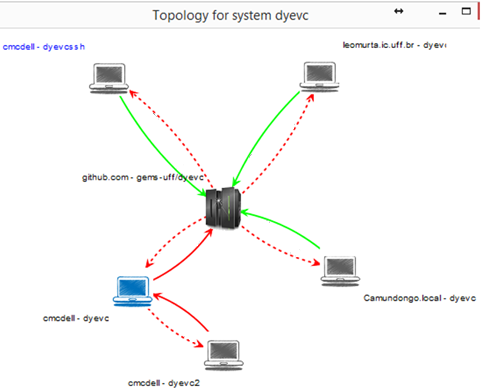
The visualization of information gathered by DyeVC is divided into four different levels of detail: Level 1 presents high-level notifications about the registered repositories; Level 2 present the whole topology of a given project. Level 3 zooms into the branches of the repository, to see the status of each local branch that tracks a remote branch. Lastly, Level 4 zooms into the commits of the repository, to see a visual log with information about each commit. 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 peers. The notifications are presented in the system notification area, in a non-obtrusive way, allowing the user to start investigating what is occurring. Fig. 2 shows an example of this kind of notification.

### 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. 3, where each node represents a known clone of the project. The current user clone is highlighted 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 DyeVC is not running at a given clone, we cannot infer if the clone pushes to or pulls from anyone. 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 from the source clone are missing in the destination clone. The second number represents how many commits in non-tracked branches from 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.

1. 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. |

1. 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 |

1. Existing commits in each repository

Fig. 5 Topology view for a given project

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

1. Status of each repository based on known remote repositories

Fig. 6 DyeVC main screen

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 3: Tracked branches

To answer question Q3 from Section I, our approach’s main screen (see Fig. 4) shows Level 3 information, allowing one to depict the status of each tracked branch between 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 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.

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. 1have the commits shown in TABLE III.

Fig. 7 Collapsed commit history



Fig. 8 Commit history for a given project



Considering just the synchronization paths presented in Fig. 1, which depend on the direction of the arrows and on the type of the lines, 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.

### 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. 6) 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).

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 shown in Fig. 4 (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).

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 repositories that are not peers. Finally, gray commits exist locally, but belong to non-tracked branches, meaning that they can neither be pushed nor pulled.

There is also the possibility to collapse nodes to provide a better understanding of huge amounts of data. As shown in Fig. 5, 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).

## Behind the Scenes

What DyeVC does behind the scenes involves depicting what are the existing repositories 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 (either by monitoring, pushing to or pulling from). Repositories that are not referenced anymore are removed from the topology.

Fig. 7 shows the algorithm to update commits in the topology. This update finds out the existing commits and depicts where they can be found. The whole process is based on the Set Theory and is executed for each repository rep being monitored by the user. The algorithm receives the repository being monitored (*rep*, a *RepositoryInfo* instance, the set of existing commits in the database (*db.commits*), the set of existing commits at *rep* on 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 (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).

**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*.

1. **begin**
2. 
3. **delete** *commitsToDelete* ***from*** *database*
4. 
5. **
6. 



2. **foreach** *c* ∈ *commitsToInsert* **do**
3. updateFoundIn(*c, rep, currentSnapshot*)
4. **endfor**
5. **foreach** *c* ∈ *commitsToUpdate* **do**
6. updateFoundIn(*c, rep, currentSnapshot*)
7. **endfor**
8. **insert** *commitsToInsert* ***into*** *database*
9. **update** *commitsToUpdate* ***in*** *database*
10. 
11. **delete***orphanedCommits* ***from*** *database*
12. **end**
13. **procedure updateFoundIn**(*c: CommitInfo, rep: RepositoryInfo: Set of RepositoryInfo, currentSnapshot: Set of Commit*)
14. 
15. 
16. **if** *isBehind* **then**
17. 
18. **endif**
19. **if** *isAhead* **then**
20. 
21. **endif**
22. **if** ( **then**
23. **if**  **then**
24. ****
25. **else**
26. 
27. **if** *isTracked* **then**
28. 
29. **else**
30. 
31. **endif**
32. **endif**
33. **endif**
34. **end**

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). Commits that will 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, than all repositories in *rep’s* pull list that contain *c* are added to *c*.*foundIn* (lines 27-29). If *rep* is ahead, than *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).

Fig. 9 Algorithm to update commits in topology

After updating where each commit is found, *commitsToInsert* is inserted into the database (line 17) and *commitsToUpdate* is 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.

# Implementation

Our approach uses a client application launched by Java Web Start[[2]](#footnote-2) Technology, and focus on monitoring Git repositories, once that this is the most used DVCS nowadays [8]. Source code can be found at GitHub[[3]](#footnote-3). The client application gathers information from repositories using JGit[[4]](#footnote-4) 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-5). We present the information gathered as a series of graphs by using JUNG[[6]](#footnote-6) 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.

# 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 **Erro! Fonte de referência não encontrada.**. 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-7), 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 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 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 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 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. 8 shows the topology view when aakosh had 121 commits pending to be pushed to the central repository (hereafter represented as central-repo). Fig. 9 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. 10 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. At this point, we are able to revisit questions Q1 and Q2:

Q1: Which clones were created from a repository? DyeVC’s topology view (Fig. 10) 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. 10) 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 (Fig. 11). 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. **Erro! Fonte de referência não encontrada.** 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. 10).

The repository history leads us to think that jeresig is a core developer in the project, because he performed most of the merges to master branch. Looking at Fig. 10, 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. 12), 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 **Erro! Fonte de referência não encontrada.** 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. 13, 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 both push and pull operations. This analysis helps us revisit and answer Q3.

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. 11 and Fig. 13). This view will show 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. 9 and Fig. 12) 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. 12. Topology view showing the three monitored repositories



Fig. 13. Adam’s tracked branches



Fig. 14. Aakoch’s commit history



Fig. 15. Jeresig’s tracked branches

## Performance evaluation

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 **Erro! Fonte de referência não encontrada.**.

**Erro! Fonte de referência não encontrada.** shows the monitored projects (name and hosting service), the repository metrics – in terms of number of commits, disk usage, number of files and number of peers in the measured topology –, and the time spent by DyeVC 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.

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.

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

Table I

Pearson coefficient between measured operations and repository metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | # commits | Size | # files | # peers |
| Insert 1st | 0.79 | 0.65 | 0.30 | -0.44 |
| Insert 2nd | 0.82 | 0.65 | 0.36 | -0.46 |
| Check Branches | 0.00 | -0.28 | -0.13 | -0.25 |
| Update Topology | 0.94 | 0.17 | 0.33 | -0.35 |
| Commit History | 0.95 | 0.62 | 0.41 | -0.61 |
| Topology | 0.86 | 0.61 | 0.59 | -0.39 |

##### Acknowledgment *(Heading 5)*

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g.” Avoid the stilted expression “one of us (R. B. G.) thanks ...”. Instead, try “R. B. G. thanks...”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

##### References

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first ...”

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors’ names; do not use “et al.”. Papers that have not been published, even if they have been submitted for publication, should be cited as “unpublished” [4]. Papers that have been accepted for publication should be cited as “in press” [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

1. G. Eason, B. Noble, and I.N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529-551, April 1955. (*references*)

We suggest that you use a text box to insert a graphic (which is ideally a 300 dpi resolution TIFF or EPS file with all fonts embedded) because this method is somewhat more stable than directly inserting a picture.

To have non-visible rules on your frame, use the MSWord “Format” pull-down menu, select Text Box > Colors and Lines to choose No Fill and No Line.

1. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.
2. I.S. Jacobs and C.P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G.T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271-350.
3. K. Elissa, “Title of paper if known,” unpublished.
4. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
5. Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740-741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
6. M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

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://docs.oracle.com/javase/6/docs/technotes/guides/javaws/ [↑](#footnote-ref-2)
3. https://github.com/gems-uff/dyevc [↑](#footnote-ref-3)
4. http://www.eclipse.org/jgit/ [↑](#footnote-ref-4)
5. https://www.mongodb.org/ [↑](#footnote-ref-5)
6. http://jung.sourceforge.net/ [↑](#footnote-ref-6)
7. https://github.com/jquery/jquery.git [↑](#footnote-ref-7)