tABLE OF cONTENTS

[Chapter 1 – Visualizing Distributed Version Control Systems 2](#_Toc397286998)

[1.1 Introduction 2](#_Toc397286999)

[1.2 Information Gathering 4](#_Toc397287000)

[1.3 Information Visualization 6](#_Toc397287001)

[1.3.1 Level 1: Notifications 6](#_Toc397287002)

[1.3.2 Level 2: Topology 6](#_Toc397287003)

[1.3.3 Level 3: Tracked branches 7](#_Toc397287004)

[1.3.4 Level 4: Commits 10](#_Toc397287005)

[1.4 Behind the Scenes 11](#_Toc397287006)

[1.5 Technologies Used 15](#_Toc397287007)

[1.6 DyeVC Usage 16](#_Toc397287008)

[1.7 Final Considerations 18](#_Toc397287009)

# – Visualizing Distributed Version Control Systems

## Introduction

As we have discussed in <link to chapter on related work>, DVCSs lead to a number of repository copies that may communicate with each other, receiving or sending updates. This operating mode resembles a peer-to-peer network topology (SCHOLLMEIER, 2001), where there are processing units and the flows between them through predefined connection paths. Whereas there are several approaches to discover such network topologies (DONG; GANG, 2012; LI, H. *et al.*, 2009; LI, M. *et al.*, 2013; UZAIR *et al.*, 2007; YAN, 2012; YONG *et al.*, 2010), to number a few, there is no correspondent approach that deals with DVCS, as discussed in <link to chapter on related work>.

Consider the example scenario shown in Figure 1, where there are 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 freelancer 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 Figure 1 indicate push operations whereas dotted lines indicate pull operations (See Section <link to section on DVCS concepts>. 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 (IEEE, 2012) 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. For instance, Mystique and Nightcrawler may not be aware of each other existence, as there is no direct communication between them.



Figure 1 - A development scenario involving some developers

Apart from drawing a topology, several questions may arise from this scenario. 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 there 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 the existing clones of a project are 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?

The DyeVC approach (CESARIO; MURTA, 2013) came to fill this gap in supporting DVCS usage. The goal of DyeVC is three-fold. First, DyeVC should work as a non-obtrusive awareness tool to increase the developer knowledge on what is going on around his repository and the repositories of his teammates. Second, DyeVC should enable repository administrators and/or 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.

This chapter explains the DyeVC approach, which consists of: (1) a mechanism to gather information from a set of repositories and (2) a set of extensible views with different levels of detail, allowing DyeVC users to visualize this information.

This chapter is organized as follows: Section 1.2 explains the data model used to store the information that our approach gathers. Section 1.3 shows how this information is presented using different levels of detail. Section 1.4 discusses what happens behind the scenes, presenting the algorithms involved in the data synchronization process. Section 1.5 presents the technologies used in the prototype implementation. Section 1.6 discusses DyeVC usage. Lastly, Section 1.7 presents the final considerations of this chapter.

## Information Gathering

DyeVC continuously gathers information from a group of interrelated repositories, starting from repositories registered by the user. As shown in Figure 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. 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.

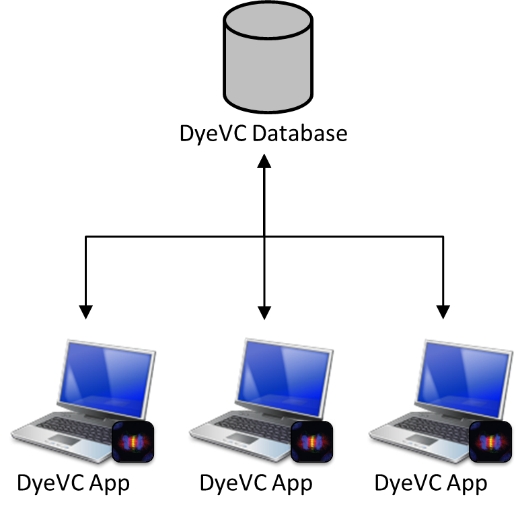


Figure 2 - How DyeVC gathers information

The data stored at the central database follows the model presented in Figure 3. A Project is used to group all repository clones of the same system, and each project is identified by a project name. Repositories are stored as RepositoryInfo and are identified by an id and the user can give it a meaningful clone name. We also store the hostname where the repository resides, as well its path (e.g., a folder name, if it is on local disk or a URL if it is on remote server. A RepositoryInfo also has a list of clones to which it pushes to and a list of clones from which it pulls. These lists are represented respectively by the self-associations pushesTo and pullsFrom. Finally, we store the list of DyeVC instances that monitor the repository, in order to remove from the topology the repositories that are no longer referenced.

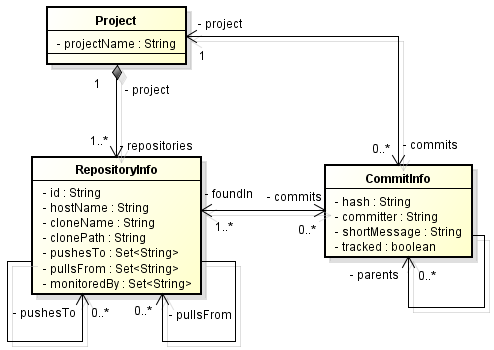


Figure 3 - Model used to store topology data

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.

DyeVC can gather information not only from repositories registered on a DyeVC instance running on the user’s machine, but also from their peers (which are the repositories that a given repository communicates), even when these peers do not have a DyeVC instance running locally. 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 Section 1.4.

## 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 begin investigating what is occurring, if desired. Figure 4 shows an example of this kind of notification. In this example, it is possible to observe that …



Figure 4 - DyeVC showing notifications in the notification area

### Level 2: Topology

In Level 2, we present a topology view showing all repositories for a given project, as depicted, where each node represents a known clone of the project DyeVC, at a given moment. The current user clone is highlighted as a blue computer and other clones are presented as black computers. Servers represent repositories that do not pull from nor push to any other clone (probably central repositories) 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 central repository.

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.

For example, it is possible to observe in Figure 5 that …

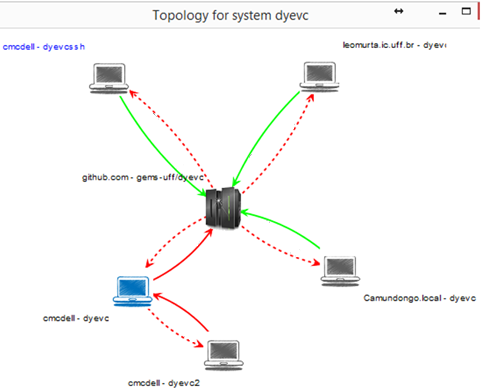


Figure 5 - Topology view of DyeVC project, at a given moment

### Level 3: Tracked branches

Level 3 information is presented at Figure 6 and it allows 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.



Figure 6 - DyeVC Main Screen

The status evaluation considers the existing commits in each repository individually. Table 1 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 2.

Table 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. |

Table 2 - 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 |

To illustrate how this approach works, let us assume that each commit is represented by an integer number and take the right portion of Figure 1, which represents developers led by Wolverine. This scenario is shown in Figure 7.



Figure 7 – Developers led by Wolverine

At a giving moment, the local repositories of each developer have the commits shown in Table 3.

Table 3 - Existing commits in each repository

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

Considering just the synchronization paths presented in Figure 7, which depend on the direction of the arrows and on the type of the lines, the perception of each developer regarding his known peers is shown in Table 4. 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 4 - 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 | behind_ylw_32 | - |

### Level 4: Commits

Level 4 complements information of Level 3 by presenting a visual history of the repository (Figure 8) as a directed acyclic graph (DAG). 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 head (e.g., commit f1a48).



Figure 8 - Commit history for a given project

Commits are drawn according to their precedence order. Thus, if a commit N is created after 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 Figure 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 by that commit (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 painted in white. Green commits are ready to be pushed, as they exist locally but do not exist in at least one of the peers 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 a large number of commits. As shown in Figure 9, 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).



Figure 9 - Collapsed commit history

## 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 follow the process represented by the activity diagram in Figure 10. This process is periodically triggered and it begins with a list of repositories being monitored. Each repository *rep* is inserted or updated in the database, where we keep track of each hostname that references them (either by monitoring, pushing to or pulling from). If the user has requested to stop monitoring *rep*, it will be marked for deletion, in which case the user’s machine is removed from *rep’s* *monitoredBy* set. Otherwise, the user’s machine is inserted in *rep’s monitoredBy* set. Next, we find which repositories *rep* pulls from and pushes to, by looking at *rep’s* configuration files, and we adjust the *pushesTo* and *pullsFrom* lists. The updates are sent to the central database, and the repositories that are no long referenced are deleted.

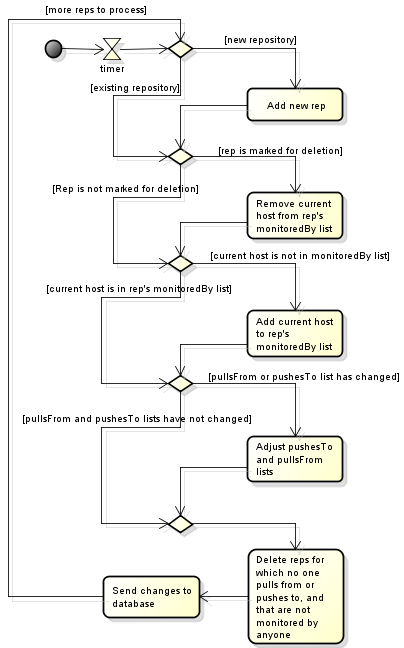


Figure 10 - Updating repositories in the topology

We use **Algorithm 1** 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*),the sets of repositories to which *rep* pulls from and pushes to (*pull* and *push*, respectively), the set of existing commits at *rep* on the previous monitoring cycle (*previousSnapshot*)and the set of existing commits at *rep* at this moment (*currentSnapshot*).

On the first time the algorithm runs for *rep*, *previousSnapshot* will be empty and thus all commits will be considered new (*newCommits,* lines 3-4). Otherwise, *newCommits* will contain the subtraction of *previousSnapshot* from *currentSnapshot*, i.e., commits that were inserted since the previous monitoring cycle (lines 5-7). Conversely, we subtract *currentSnapshot* from *previousSnapshot* to find *commitsToDelete*, that containscommits that were deleted since the previous monitoring cycle (line 8) and we delete them from the database (lines 9-11)

Next, we retrieve the number of existing commits in the database for the project that *rep* belongs to (line 12). If no commits are found, this means that all of the new commits will be inserted into the database (*commitsToInsert*, lines 13-14). Otherwise, we should verify which of the commits should be updated or inserted. We find commits to be inserted by subtracting the existing commits in the database (*db.commits*) from *newCommits* (line 16). Commits that will be updated are represented by *commitsToUpdate* and 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 (line 17). 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*.

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

**input**: a RepositoryInfo *rep* representing the repository being analyzed; four sets *pull*, *push*, *previousSnapshot*, and *currentSnapshot*.

1. **begin**
2. *Let db.commits be the set of commits in the database, for the system that rep belongs to (rep.project), respecting the data model shown in Figure 3.*
3. **if** *previousSnapshot* = { } **then**
4. *newCommits* = *currentSnapshot*;
5. **else**
6. *newCommits* = *currentSnapshot* \ *previousSnapshot*;
7. **endif**
8. *commitsToDelete* = *previousSnapshot* \ *currentSnapshot*
9. **if** *commitsToDelete* ≠ { }
10. **delete** *commitsToDelete* ***from*** *database*
11. **endif**
12. *commitCount* = #(*db.commits*)
13. **if** *commitCount* = 0 **then**
14. *commitsToInsert* = *newCommits*
15. **else**
16. *commitsToInsert* = *newCommits* \ *db.commits*
17. *commitsToUpdate* =
18. **endif**
19. **foreach** *c* ∈ *commitsToInsert* **do**
20. updateFoundIn(*c, rep, push, pull, currentSnapshot*)
21. **endfor**
22. **foreach** *c* ∈ *commitsToUpdate* **do**
23. updateFound(*c, rep, push, pull, currentSnapshot*)
24. **endfor**
25. **insert** *commitsToInsert* ***into*** *database*
26. **update** *commitsToUpdate* ***in*** *database*
27. *orphanedCommits* =
28. **delete***orphanedCommits* ***from*** *database*
29. **write** *currentSnapshot* ***to*** *disk*
30. **end**
31. **procedure updateFound**(*c: Commit, rep: RepositoryInfo, push, pull: Set of RepositoryInfo, currentSnapshot: Set of Commit*)
32. *isAhead* = { c ∈ currentSnapshot ∧ (∃ push[i] | c ∉ push[i])}
33. *isBehind* = {c ∉ currentSnapshot ∧ (∃ pull[i] | c ∈ pull[i])}
34. **if** *isBehind* **then**
35. *c.foundIn* = *c.foundIn* ∪ σc ∈ pull.commits (pull)
36. **endif**
37. **if** *isAhead* **then**
38. *c.foundIn* = *c.foundIn* ∪ rep ∪ σc ∈ push.commits (push)
39. **endif**
40. **if** (*not(isBehind) and not(isAhead)*) **then**
41. **if** *c* ∉ *currentSnapshot* **then**
42. *c.foundIn* = ((*c.foundIn* \ *rep* ) \ *push*) \ *pull*
43. **else**
44. **if** *c* ∈ *any tracked branch* **then**
45. *c.foundIn* = *c.foundIn* ∪ *rep* ∪ *push* ∪ *pull*
46. **else**
47. *c.foundIn* = ∅
48. *c.foundIn* += *rep*
49. **endif**
50. **endif**
51. **endif**
52. **end**

Commits to be inserted or updated must be verified to check where they exist, thus updating the *foundIn* attribute. This verification is done using the procedure *updateFound* (lines 31-50), which is called in lines 19-24. 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*, i.e., if *c* exists locally and if there is at least one repository *rep* pushes to that does not contain *c* (line 32). Likewise, it verifies if *rep* is behind of any repository in its pull list regarding *c*, i.e., if *c* does not exist locally and if there is at least one repository *rep* pulls from that contains *c* (line 33). If *rep* is behind, than all repositories in *rep’s* pull list that contain *c* are added to *c’s* *foundIn* list (lines 34-36). If *rep* is ahead, than *rep* and all repositories in *rep’s* push list that contain *c* are added to *c’s* *foundIn* list (lines 37-39). It may be the case that *rep* is not ahead nor behind any repository regarding *c.* In such case, one of the following scenarios may happen: In scenario 1 *c* does not exist in current snapshot, meaning that it also does not exist in any of the related repositories, thus we remove *rep* and all its related repositories from *c’s foundIn* list (lines 41-42). In scenario 2 *c* is in a tracked branch, meaning that it also exists in all related repositories, thus we include *rep* and all its related repositories in *c’s foundIn* list (lines 44-45). 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’s foundIn* list (lines 46-48).

After updating where each commit is found, *commitsToInsert* is inserted into the database (line 25) and *commitsToUpdate* is updated in the database (line 26). Finally, it may happen that some commits end up with an empty *foundIn* list, meaning that they do not exist anywhere in the topology (line 27). These so-called *orphanedCommits* are then removed from the database (line 28) and *currentSnapshot* is written to disk (line 29), to be used in the next monitoring cycle.

## Technologies Used

We implemented our approach as a Java Swing application (ELLIOTT *et al.*, 2002) launched by Java Web Start (MARINILLI, 2001) Technology. It currently monitors Git repositories, as it is the most used DVCS nowadays (ECLIPSE FOUNDATION, 2013). The source code and the link to download the tool via Java Web Start can be found at GitHub[[1]](#footnote-1). The application gathers information from repositories using JGit library (JGIT, 2014), 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 (CHODOROW, 2013).

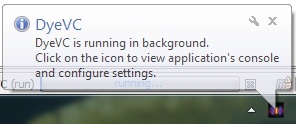
We hosted our database on a free MongoDB instance provided by MongoLab[[2]](#footnote-2). To prevent from firewall blocks when accessing the database, we did not use MongoDB proprietary API, which would demand opening specific ports to connect to MongoDB. Instead, we opted to use MongoLab’s RESTful (*Representational State Transfer*) API (*Application Programming Interface*). RESTful APIs (RICHARDSON; RUBY, 2013) have the advantage to be available using standard HTTP and HTTPS protocols. This way, our approach can be used easily from inside corporate and academic environments, without major problems. In order to use the RESTful API provided by MongoLab, we implemented a MongoLabProvider, responsible for translating the application methods into RESTful commands and vice-versa. This provider also serializes and deserializes the application objects to and from JSON (*JavaScript Object Notation*) (JSON, 2014)representations in order to send and receive them through the RESTful commands.

We present the information gathered as a series of graphs by using JUNG (*Java Universal Network/Graph Framework*) library (JUNG, 2010), 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, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines.

## DyeVC Usage

As we discussed in the previous section, DyeVC uses Java Web Start technology and thus does not need to be formally installed. After launching the application for the first time, it creates a shortcut in the Desktop (Figure 11.a), which can be used to execute the application later on. After running the application, it lies on the system tray bar (Figure 11.b). A single click on the icon will show the application window and minimizing it will take it back to the tray bar.

1. **(b)**

Figure 11 – DyeVC icon on the Desktop and on the tray bar

After maximizing the application, the main window is shown (Figure 12)

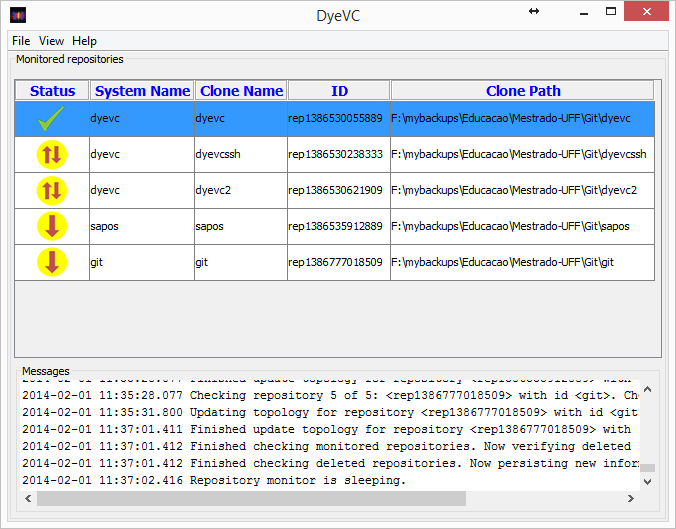


Figure 12 – DyeVC main window

The main window shows all monitored repositories, along with the following information in the Monitored Repositories panel:

* **Status**: An icon representing the clone status related to its known partners (as discussed in Section 1.3.3);
* **System Name**: The system or project name in which the clone belongs to. Clones that belong to the same project are shown together in the topology view;
* **Clone Name**: The name that the user gives to this particular clone. It must be unique on each single machine for a particular system name;
* **Id**: An internal unique id generated by DyeVC;
* **Clone Path**: The path in the local machine where this clone is found.

The application also shows relevant information regarding the monitoring status in the Messages panel. There is the possibility to see more detailed log messages by clicking on **View -> Console Window**. A new window will be opened, where messages are displayed according to their levels of criticality. The default behavior is to show INFO, WARN, and ERROR messages, but right clicking on the panel allows the user to change this behavior, allowing to also log TRACE and DEBUG messages.

The main screen will be initially empty as there is no repository being monitored. Clicking on **File -> Add Project** allows creating a new monitoring configuration (Figure 13). The user can choose a system name from the ones provided on the drop-down list, or type a new one, and click on the Explore button to choose the path where the clone is located. The Clone Name will be automatically chosen by the application, based on the folder name where the clone is located. For instance, if the user points the Clone Address to /home/users/username/myprojects/xyz, the Clone Name of this configuration will be xyz.

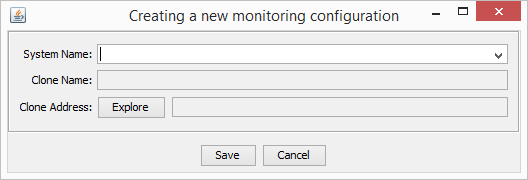


Figure 13 – Creating a new monitoring configuration

Once repositories are being monitored, the user is be able to navigate through all the visualization levels discussed in Section 1.3, where each one of them is described, with examples. Level 1 (Notifications) will be shown as notifications in tray bar; Level 2 (Topology) will be presented by right clicking on a repository and choosing **Show Topology**; Level 3 (Tracked Branches) will be shown by hovering the mouse over any monitored repository; and Level 4 (Commits) will be accessible by right clicking on a repository and choosing **Show Commit History**.

## Final Considerations

In projects that use DVCS, there may be several clones where changes are being inserted simultaneously. These clones may communicate with each other indistinctively, turning the administration of such environment into a tough task. Today, administrators have no way to visualize the various clones and their dependencies, and developers have limited choices to provide awareness regarding parallel changes.

In this chapter, we presented the DyeVC approach, which supports the development and administration under DVCS environments, providing awareness in a non-obtrusive way, enabling administrators to visualize the repository topology and establishing an extensible platform to present information and metrics. We also discussed aspects of its implementation and usage.

The next chapter presents an experimental study to evaluate DyeVC in real projects.

Bibliography

CESARIO, C. M.; MURTA, L. G. P. What is going on around my repository? In: I BRAZILIAN WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION AND MAINTENANCE, VEM’13, 29 Sep. 2013, Brasilia, Brazil: UNB, 29 Sep. 2013. p. 14–21.

CHODOROW, K. *MongoDB: The Definitive Guide*. Second Edition edition ed. Beijing: O’Reilly Media, 2013.

DONG, J.; GANG, X. A Topology Discovery Algorithm Based on the IP-Network. In: 2012 INTERNATIONAL CONFERENCE ON CONTROL ENGINEERING AND COMMUNICATION TECHNOLOGY, ICCECT’12, Dec. 2012, Shenyang, Liaoning, China: IEEE, Dec. 2012. p. 665–668.

ECLIPSE FOUNDATION. *The Open Source Developer Report - 2013 Eclipse Community Survey*. Survey. San Francisco, CA, USA: Eclipse Foundation, Jun. 2013.

ELLIOTT, J.; ECKSTEIN, R.; LOY, M.; COLE, B. *Java Swing, Second Edition*. 2nd edition ed. Sebastopol, CA: O’Reilly Media, 2002.

IEEE. IEEE Standard for Configuration Management in Systems and Software Engineering. *IEEE Std 828-2012 (Revision of IEEE Std 828-2005)*, p. 1–71, Mar. 2012.

JGIT. *JGit Project Homepage*. Available at: <http://www.eclipse.org/jgit/>. Accessed: 31 aug. 2014.

JSON. *JSON Homepage*. Available at: <http://json.org/>. Accessed: 31 aug. 2014.

JUNG. *JUNG Homepage*. Available at: <http://jung.sourceforge.net/>. Accessed: 31 aug. 2014.

LI, H.; DAN, C.; HUAIXIANG, B.; SHURONG, L. Topology Discovery Algorithm Based on Ant Colony Algorithm of Power Line Carrier Sensor Network. In: 2009 INTERNATIONAL CONFERENCE ON COMMUNICATION SOFTWARE AND NETWORKS, ICCSN ’09, Feb. 2009, Macau, China: IEEE, Feb. 2009. p. 102–105.

LI, M.; YANG, J.; AN, C.; LI, C.; LI, F. IPv6 network topology discovery method based on novel graph mapping algorithms. In: 2013 IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS, ISCC’13, Jul. 2013, Split, Croatia: IEEE, Jul. 2013. p. 000554–000560.

MARINILLI, M. *Java Deployment with JNLP and WebStart*. 1 edition ed. Indianapolis, Ind: Sams Publishing, 2001.

RICHARDSON, L.; RUBY, S. *RESTful Web APIs*. 1 edition ed. Sebastopol, Calif.: O’Reilly Media, 2013.

SCHOLLMEIER, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: 2001 FIRST INTERNATIONAL CONFERENCE ON PEER-TO-PEER COMPUTING. PROCEEDINGS, P2P’01, Aug. 2001, Linkoping, Sweden: IEEE, Aug. 2001. p. 101–102.

UZAIR, U.; AHMAD, H. F.; ALI, A.; SUGURI, H. An Efficient Algorithm for Ethernet Topology Discovery in Large Multi-subnet Networks. In: 2007 IEEE INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING, SoSE ’07, Apr. 2007, San Antonio, TX, USA: IEEE, Apr. 2007. p. 1–7.

YAN, H. The study on network topology discovery algorithm based on SNMP protocol and ICMP protocol. In: 2012 IEEE 3RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCE, ICSESS’12, Jun. 2012, Beijing, China: IEEE, Jun. 2012. p. 665–668.

YONG, W.; NAN, P.; XIAOLING, T. Network topology discovery algorithm based on OSPF. In: 2010 INTERNATIONAL CONFERENCE ON INTELLIGENT COMPUTING AND INTEGRATED SYSTEMS, ICISS’10, Oct. 2010, Guilin, China: IEEE, Oct. 2010. p. 136–139.

1. https://github.com/gems-uff/dyevc [↑](#footnote-ref-1)
2. http://mongolab.com [↑](#footnote-ref-2)