Collaboration and Governance with Distributed Version Control

PETER C. RIGBY
University of Victoria
EARL T. BARR
University of California, Davis
CHRISTIAN BIRD
University of California, Davis
DANIEL M. GERMAN
University of Victoria
and
PREMKUMAR DEVANBU
University of California, Davis

OSS projects are widely adopting distributed version control (DVC). The reasons for this shift and its effects on developer workflows and project organization are not well understood. However, there is no shortage of postings extolling the virtues or proclaiming the dangers of DVC. To shed light on this issue, we objectively examine the impact of using a DVC along three dimensions: Episodic Collaboration, Governance, and Continuous Collaboration. We interviewed developers of major OSS projects that have switched to DVC, analyzed the version history of 60 projects that switched from CVC to DVC, and examined the effect of version control on the organizational structure of the Linux kernel and FreeBSD projects. We find that DVC substantially facilitates the release process and better insulates developer teams from the pain of dealing each other’s changes prematurely. So far, however, we have found little evidence that DVC adoption is changing the way projects govern themselves or the way developers discuss and work on early versions of code. The results of our investigation are pertinent to both researchers and projects or developers considering switching to a DVC system.

Categories and Subject Descriptors: D.2.7 [Distribution, Maintenance, and Enhancement]: Version Control; D.2.8 [Metrics]: Process Metrics; D.2.9 [Management]: Life Cycle, Programming Teams, Software Configuration Management

Additional Key Words and Phrases: Configuration management, empirical software engineering, collaboration

1. INTRODUCTION

Since people have been writing code, software has grown ever larger and more complex. The size of development teams has kept pace. Adding developers has a price, as Brooks famously observed [Brooks 1995], and poses organizational and management challenges. Not surprisingly, management and organizational structure are critical to the success of development projects, both industrial and open source [Nagappan et al. 2008; Bird et al. 2008]. Although software team members can coordinate over many communications channels (such as face to face meetings, IRC, mailing lists, and voice chat), version control (VC) is the main mechanism for
Fig. 1: Hypothetical development process for one release cycle in Linux

recording, managing, and propagating source code changes. Thus, version control systems (VCS) have a direct and profound impact on a software project’s policies and processes. Indeed, the prominence of software configuration management (the role played by a VCS) in the Capability Maturity Model underscores their importance [Conradi and Westfechtel 1998; Humphrey 1988].

The client-server model has long dominated the design of VCS. RCS, CVS, and subversion (SVN) all store the history of changes to the software artifact in a single, centralized location. Developers using this model communicate their changes through this central server. We refer to these version control systems as centralized version control (CVC). In recent years, a new version control paradigm has appeared that distributes a project’s history across all developers, giving each developer his own “copy” of the change history in a local repository. This new paradigm also facilitates the propagation of changes between any two repositories. In the open source world, git, mercurial, bazaar, and darcs are examples of distributed version control (DVC).

DVCs arose to enable a development process CVCs do not support well. For instance, after the Linux kernel’s license to use BitKeeper¹ was revoked, Linus Torvalds developed git. The VCS available at the time were largely centralized and could not support the Linux development process, which makes heavy use of branching, tolerates and encourages many versions of the source code to exist simultaneously, and propagates changes to release using the “hierarchy or chain of trust.” Linux illustrates the flexibility a capable VCS like git can provide for a project. A more limited VCS might constrain the ways in which a development team operates. Figure 1 manifests the power and flexibility of the git DVC during a single development cycle in Linux. The figure abstracts and simplifies the actual Linux development process to illustrate those aspects of DVC that we examine in this

¹BitKeeper is a proprietary DVC created by Larry McVoy [McVoy 2009].
The Effect of Distributed Version Control

paper; it shows the development history of six different, interacting git repositories. Three of the repositories are owned individually: one by Linus, one by Ingo and the other by David M. Three others are shared. One is used by the scheduler team, and two by competing network teams. The flexibility of ownership is evident: a DVC allows individual, or shared, operation, with full versioning support for either. Another striking phenomenon is the frequency of branching and merging; all repositories begin their histories with 2.6.29, branch into independent evolution, and then merge in various ways, leading to the subsequent release 2.6.30.

DVC systems have local history and allow any two repositories that share content to easily merge subsequent changes (See §4 for more detail). This facility enables merging where it was not possible and generally reduces the human effort required for merging; it enables liberal branching/merging behaviour, with the attendant proliferation of parallel workflows. The parallel workflow histories in Figure 1 illustrate several important phenomena, that are the subject of our exploratory study: episodic collaboration, governance, and continuous collaboration.

The histories begin with release 2.6.29 and end with 2.6.30. Both are in Linus’ repository. The three teams, and the two individuals, pull from this release, and begin their work. Each repository evolves independently of the other. We call this work-style episodic collaboration: teams and individuals work independently and in parallel, and then merge later, leveraging git’s fast and easy merging. Linus’ repository is preeminent: all activity begins and ends there; his repository is the source of releases, indicating his control over the content of Linux. The flow of content from other repositories into Linus’ evinces the phenomenon of governance: Ingo and David M.’s repositories mediate content flow. Linus implicitly delegates governance decisions on networking to David M., and scheduling to Ingo, by pulling content from their insulated repositories. In turn, these two developers exercise that decision-making authority by pulling content into their repositories. Finally, the shared repositories for the three teams illustrate the use of the git DVC for continuous collaboration: the scheduler, and the two networking teams can coordinate work, dividing labor and sharing contributions on a common sub-task via their shared repository.

The three phenomena synergize in interesting ways: governance determines the manner in which episodically collaborating teams merge their repository: whose repositories are merged into whose? Continuous collaboration within a team is actually simplified by episodic collaboration among teams. The scheduling team co-ordinates via their shared repository, but can work independently of the two networking teams; and the two networking teams can work independently of each other. Episodic collaboration also supports governance: the ability of two networking teams to simultaneously work in isolation enables two different implementation choices to emerge. In Figure 1, David M. exercises his delegated governance authority over what ends up in 2.6.30 by selecting the first network team’s work while dropping the second team’s branch. In the ensuing discussion, we analyze aspects of these phenomena, as manifest in practice over many OSS projects.

Over the past few years, a number of large, mature, and popular open source projects have moved from using CVCs such as CVS and subversion to DVCs. Notable examples include Perl, Python, Glibc, GNOME, the Glasgow Haskell
COMPILER, XEmacs, RUBY on RAILS, Mozilla Firefox, and X.Org. Furthermore, the Debian distribution heavily uses DVC to maintain Debian specific forks of the projects they distribute. Of the reporting projects that Debian integrates, over 1,000 (roughly 27%) use a DVC.

During the same time period, a number of blogs, mailing list discussions, and other web sites have, often passionately, extolled the virtues or warned of the dangers inherent in DVCs. Our goal is not to answer the question of whether DVC is better than CVC, but rather to examine the reasons for migration from one to another and examine the impact that such a migration has on a large software project. As moving from one VC to another is difficult, time-consuming, and requires consensus, we posit that this migration is evidence that DVC offers important abilities unavailable in CVC. This paper examines the impact of DVC on individuals’ workflows, the organization’s structure, and group interactions.

We summarize the main contributions of this paper thus:

- We consider 3 essential core activities of distributed, decentralized development: episodic collaboration, governance, and continuous collaboration.
- We study how these activities are manifest in DVC, and how the features of DVC facilitates and modulates them;
- We present evidence suggesting that, ironically, DVC inherently better matches a hierarchical, dictatorial governance model, while CVC better matches peer groups.

2. THEORETICAL FRAMEWORK

In this paper, we study the effect of CVC and DVC on the work practices of teams. Team-based, collaborative development is driven strongly by software design. Ideally, large software systems are well-modularized, with loose coupling among the parts, which are internally cohesive and strongly coupled. This allows small teams of developers to form around specific parts of the system [Bird et al. 2008] and collaborate more intensively as needed, while attenuating the need for collaboration between teams working on different parts. These two styles of collaboration can be considered continuous and episodic collaboration. In any large team, disagreements about processes or artifacts will arise and necessitate a governance mechanism to resolve these differences.

We consider the effects of DVC and CVC on three main dimensions of team practices:

- Episodic Collaboration. How do people divide up the task, work independently, and then merge the results?
- Governance. How are critical decisions about the system resolved?
- Continuous Collaboration. When teams work collaboratively, rather than independently, how are changes shared and co-ordinated?

In the ensuing discussion, we elaborate each dimension, detailing the specific phenomena of interest.

Episodic Collaboration Large software project development necessarily involves a large number of tasks, with various interdependencies. With well-designed, modular
The Effect of Distributed Version Control

interdependencies (albeit not completely avoidable) are manageable, allowing developers to break off and work in isolation and in parallel, whilst also merging their work periodically. One example is when a larger task is divided up into several smaller, independent tasks, and then merged. Another example is when several alternative design choices for the same task are simultaneously and independently explored; the final choice emerges from among the alternatives. The increasing tendency towards distributed (even mobile) development also amplifies the frequency of this type of episodic collaboration.

VC plays a crucial role [Sarma et al. 2003] in episodic collaboration. Ideally, a VC supports good isolation during the episodes of parallel work. Good isolation occurs when developers desirous of working independently are able to make changes without interfering with each other; bad isolation occurs when developers unknowingly make conflicting changes, and then have to struggle to recover. DVC systems make it easier for developers to organize into a team, work relatively independently to make a complex sequence of related changes (e.g. adding an entire new feature) and then atomically commit and merge this sequence of changes with other sequences of changes made comcomitantly by other teams.

Governance. In any large project, disagreements over requirements, design, implementation, inter alia, will naturally arise, and must be resolved. We define project governance as the process of resolving such disagreements and making decisions that have broad impact on the community. Large OSS projects are governed in two ways, by a peer group (a foundation or oligarchy) or by a “benevolent” dictator. In a peer group, the content of a software system, is decided by democratic deliberation; in a dictatorship, content is picked by the dictator [Berkus 2007]. Do these different styles of governance influence the type of VCS that a project can use? Ironically, our analysis of governance shows that if a project is socially centralized (i.e. dictatorship), it is a better candidate for DVC than a more socially distributed project (i.e. peer group). To test if this result holds in large OSS projects, we perform a case study of the Linux kernel and FreeBSD.

Continuous Collaboration Small teams of developers working together on a cohesive, internally strongly coupled part of the system need continual, intensive collaboration. We examine how individuals and subgroups of developers organize themselves to share changes. The typical working style is to continually share and discuss work-in-progress code, making it ready for release. We found several significant differences in the way DVC and CVC are used for continuous collaboration. DVC simplifies certain collaborative practices. Small teams can conveniently break off and collaborate intensively using their own “sandbox” repositories. Team or individuals desirous of maintaining a slightly different version of a product (e.g. the Mac OS X version of Emacs) can do so by maintaining their own forks. Surprisingly, we also found that mailing lists continue to be extensively used for continuous collaboration in projects using DVC. Regardless of the type of VCS, the current practice is to discuss initial changes as patches on the mailing list. Unlike “perfected” changes that are committed to the VCS, the works-in-progress that lead to these commits are not typically recorded by VCS.
3. METHODOLOGY

Our objective is to understand how using DVC has changed developers’ interactions, workflows, and the project’s organizational structure. In an effort to understand the differences between DVC and CVC from an operational point of view, we observed the development activities in projects that switched from CVC to DVC and interviewed a number of lead developers from these projects regarding their switch. Following these interviews, we gathered data from the development history of these projects and quantitatively evaluated hypotheses based on their responses. The results of this investigation are interesting not only to researchers, but also to developers and projects considering switching to a DVC system.

Interviewing project leaders was critical in understanding why people switched to DVC, the perceived benefits and drawbacks of the switch, and (in cases where the projects have used DVC for some time) how it has affected the policy and development process of the projects. The data mining of the VC history and the email lists (communication) allowed us to provide less biased evidence of the effects of DVC. We interleave quotations from interviews and numerical findings from data mining to triangulate and provide a balanced perspective.

We conducted semi-structured interviews of four projects and six people. Semi-structured interviews make use of an interview guide that contains general groupings of topics and questions rather than a pre-determined exact set and order of questions [Lindlof and Taylor 2002]. Semi-structured interviews are often used in an exploratory context when there are clear research questions [Lindlof and Taylor 2002; Weiss 1995]. The responses from these interviews help develop hypotheses and focus quantitative analysis. We extracted themes from the interviews using a modified version of Creswell’s guidelines [Creswell 2003] for coding. The interview guide that we used can be found at http://janus.cs.ucdavis.edu/~cabird/vcstudy/vcinterviewquestions.pdf. We minimally copy-edited the quotes for readability. We eliminated false starts and superfluous crutch words; we used standard notation, delimiting clarifying comments with brackets and marking the suppression of unnecessary phrases with an ellipsis [Lindlof and Taylor 2002].

For the quantitative mined data, we either developed metrics or modified existing ones to best examine the impact of DVC in the context of our dimensions. The data used, the definition of the metric, and threats to validity are discussed in the section in which the metric is used.

We chose to examine 60 projects that had transitioned from use of a CVC to use of a DVC. These projects were drawn from lists of projects using DVC on Wikipedia and GitWiki and include such notable projects as Wine, Samba, Perl, Ruby on Rails, and the Glasgow Haskell Compiler. These projects vary in age from 21 years (in the case of Perl) to 6 months (pthread-stubs in X.Org) with a median of 4.5 years. The number of contributors as recorded by the repositories ranges from 1462 (Wine) to 1 (dri2proto in X.Org). The commits to these projects number from 139,187 (Samba) to just 6 (pthread-stubs in X.Org). As such, our selection of projects for analysis spans a broad spectrum of OSS projects in terms of size, age, and development activity. All projects have used DVC for at least 5 months at the time of this study;
the majority of them for over one year.

We use Linux to evaluate hypotheses and questions regarding advanced DVC usage because the Linux kernel project has never used a CVC and its developers are generally very experienced with DVC. Linux started using git in 2005; we have 3.5 years of Linux VC data and the corresponding data from Linux kernel Mailing List (LKML). The mailing list data is used to evaluate hypotheses in §6. As described in §5, we compare the Linux kernel to FreeBSD kernel on a number of high-level VC related parameters. We extract CVC and mailing list data over 3.5 year period from FreeBSD starting from 2003. Since FreeBSD contains external code that creates a complete UNIX distribution similar to, for example, Ubuntu Linux, we manually removed source code repositories and mailing lists that were not related to kernel development. Over those periods, there were 4K and 1K developers, 118K and 30K commits, and 443K and 31K mail messages for Linux and FreeBSD respectively. The FreeBSD numbers make it clear that we excluded some of the discussion relating to the kernel. However, it is natural that problems with applications will lead to changes in the kernel when the kernel and UNIX distribution are so closely linked. Since these lists are not explicitly dealing with kernel development, they were excluded from the analysis. As a result the number of mailing list messages is low for FreeBSD because we only have core kernel mailing lists. We do believe however, that the sample that we have is large and representative of FreeBSD kernel development.

4. EPISODIC COLLABORATION

As observed earlier, developing complex software requires developers to work together without interfering unduly with each other. A VCS should allow developers to work in parallel and in isolation [Sarma et al. 2003]. Consider the three steps of episodic collaboration: Step 1) begin a task starting with a reasonable version, Step 2) work undisturbed and in isolation for a while, and then Step 3) merge with others’ work with minimal conflict resolution effort. DVC supports these three steps nicely.

The positive effects of DVC arise out of two important advances that facilitate isolation between episodes of collaboration: 1) atomic branches whose sequence of commits are written to the repository in a single atomic transaction and 2) local and complete history. First, VCS have evolved toward providing increasingly coarse-grained atomic commits, starting with files (RCS ci/co) to commits of sets of files (SVN changesets), culminating in a single-shot commit of sequences of sets of files (DVC), which we term atomic branches. Second, Complete and local history allows any two branches (and by extension, repositories) to be efficiently compared and merged if they contain a common ancestor. Even repositories that have never directly communicated can merge, so long as they acquired a common ancestor through a third repository. Git realizes this feature via content addressability [Kernel.org 2009a]. Easy merging profoundly impacts how developers share and integrate changes; we defer this discussion to §6, which discusses continuous collaboration.

These advances make all three steps of episodic collaboration easier. We observed three significant positive effects on the work styles arising from the use of DVC systems, which occur at three distinct phases of episodic collaboration. First, when beginning an episode of isolated work, the developer must begin her work at some
point: DVC allows her to begin at a stable point, rather than at an unstable "head" (§4.1). Second, during her development effort, DVC provides a private, local repository to which she can commit without interference (§4.2). Finally, the convenience of atomically merging her branch (a sequence of changes) at the end of the episode allows her to avoid the nightmare of a "mega-commit" wherein she must spend considerable effort to merge a large set of changes into the main or release branch (§4.3).

4.1 Working off Named Stable Bases

In many OSS projects, developers begin their work with the latest commit in a repository, often called the "head," so that they are up-to-date with the latest changes and their commits are less likely to cause conflicts during integration. For example, Apache automatically rejects patches that are not diffed against the latest head [Rigby et al. 2008]. However, "working off the head" requires dealing with potentially unstable, and often poor quality code. In a large project, a developer working off of the head may be forced to deal with code that is both unstable and unfamiliar. While debugging, the developer will struggle to discover whether she is at fault, or if another, unrelated, inchoate, unfamiliar, untested change is causing the problem. As if these were not bad enough, working off head can also force a developer to contend with integration issues and compilation errors [Berczuk 1996]. Torvalds decries this practice:

"You should _never_ pull my tree at random points. It makes your tree just a random mess of random development. ...And, in fact, preferably you don’t pull my tree at ALL, since nothing in my tree should be relevant to the development work _you_ have to do. ...You also lose a lot of testability since now all your tests are going to be about all my random code. ...But if you want to sync up with major releases, do a git pull linus-repo [tag] v2.6.29 or similar to synchronize with that kind of _non_random_point._"

Torvalds, Linux kernel [Linus Torvalds 2009]

"Working off head" mixes integration and development changes and causes developers to be constantly distracted by unrelated changes. By supporting multiple repositories and excellent merging, DVC has the best of both worlds: developers can work off a stable, tested release instead of an unstable head, but still merge their completed work into the head of the upstream, shared repository. Any broken code or failed test cases encountered during development are more likely the result of a developer’s own changes. Berczuk [Berczuk 1996] and Coplien [Coplien 1998] name this pattern of development, _Named Stable Bases_. They claim that by working in this way, developers can better anticipate the effects of changes because they are local, and that this preserves "the rule of least surprise." Further, they point out that it is often helpful to have various code bases at different levels of stability. This is possible in DVC, because, once a developer’s changes on a branch are finished, those changes can atomically move to an integration repository where developers focus on integration issues.

Certainly, groups of developers working together on a single task will likely want to "work off a local or specific head" because their changes are related. However, in a large project, these developers will still want to be insulated from developers...
working on unrelated tasks. In §6, we expand the previous analysis to include sandboxes: a place where developers can “play” together as they develop, test, and review a particular feature or fix.

To determine if experienced DVC developers take advantage of Named Stable Bases, we examined, in the context of the Linux kernel, the number of branches that occur at tagged release commits, non-release tagged commits, and commits that were not tagged. Releases explicitly fulfill the criteria for being a Named Stable Base and other tagged commits represent milestones or release candidates which may also represent a form of a Named Stable Base. Figure 2 shows boxplots of the number of branches originating from release commits, other tagged commits, and commits with no tags. There is a median of 17.5 branches from each release commit and 7 branches from each non-release tagged commit, while the vast majority of non-tagged commits have no branches originating from them. Note that the outliers for the commits with no tags in Figure 2 represent around 1% of the total population of untagged commits. Wilcoxon tests confirmed that the difference in medians is statistically significant ($p < .001$) in all cases. We conclude that in the Linux kernel, developers more frequently update their changes (branch) at release time and are taking advantage of Named Stable Bases.

4.2 Light-weight, Atomic Branches

Developing a new feature often requires making changes that make the software temporarily unstable. Ideally, uninvolved developers should be insulated from these changes until the feature has achieved some degree of stability. At the same time, a developer working on a new feature should still have access to VC to commit incremental changes, and rollback, as necessary. Berczuk [Berczuk 1996] makes this
point in his discussion of configuration management patterns, where he argues that developers should checkpoint changes at frequent intervals to a location separate from the “team version control,” and that only tested and stable code should be integrated. When the feature is ready, its integration must not be too difficult or the productivity gained from working on an isolated branch is lost. Indeed, Perry et al. [Perry et al. 2001] claim that tool support for integration is important because “integration too often is painful and distracting” and because development lines diverge when parallel development goes on too long.

Our interviews confirm this finding. Interviewees indicated that, when using a CVC, branches were “painful and difficult” to integrate. In some cases, two branches would grow so far apart, they had to abandon one of them altogether. Under CVC, branches were typically created only for releases and not new features.

“We had branches for versions [releases]. Feature branches were VERY rare for us.”

Koziarski, Ruby on Rails [Michael Koziarski 2009]

In agreement with our interviewees, we see that few branches were created pre-DVC. Of the examined 60 projects that switched to DVC, 1.54 branches were created on average per month per project when they used CVC; after switching to DVC, the average rose to 3.67. A Wilcoxon non-parametric test of means shows that the difference is statistically significant. There are some interesting anomalies. For instance, the Samba project used branches extensively before migrating to a DVC, creating on average of 5.35 branches per month. Nonetheless, their use of branching grew dramatically to an average of 41.42 per month after switching to git. In contrast, the Wine project has only 1 branch in its repository created after the switch to a DVC and none before. Thus, we conclude that branches are used more commonly in DVC.

Our results show that branches proliferate when they are easy to merge, or light-weight. DVC supports light-weight branches by automatically discovering a common ancestor on which to graft an incoming branch during a merge. Recall that an atomic branch is a sequence of commits that are written, in sequence, to a remote repository in a single transaction. Atomic branches preserve the history of a branch through merging. Light-weight branching enables atomic branches: the benefits of atomic branches would be lost if they were difficult to merge.

In this section, we discuss the reasons for, and the use of, the light-weight, atomic branches. We hypothesize that these branches in DVC allow a developer to isolate unstable or experimental changes in branches, work on them without being distracted by the concurrent activity of other developers, and then finally merge a perfected, workable sequence of changes back into the main line of development. To measure how this phenomenon differentially manifests in DVC and CVC, we quantify the degree to which branches insulate developers from distraction and a notion of branch cohesion. In terms of our metrics, we hypothesize that 1) DVC branches better insulate developers than CVC branches do and that 2) DVC branches exhibit greater cohesion.

Unfortunately, the analysis of how developers use branches is difficult: we cannot directly compare the histories of projects when some of them use CVC and others DVC, nor can we simply compare branches before and after the transition to DVC.
We can, however, construct a naïve, branchless view of a branched history. We then compare branches across the two views of a single history. Since our investigation rests on the branchless view, we introduce it next.

4.2.1 Date-flattening DVC history. In CVC, developers must explicitly create branches in the central repository. In DVC, every developer has her own repository, which is ipso facto a set of branches. After two developers using a DVC pull changes from a shared repository, each developer commits their changes to his local branch. When these developers next exchange changes, their separate branches are automatically merged creating implicit branches in their histories [Bird et al. 2009]. In a CVC, both developers would be making their changes in the same, shared repository, thus creating one common shared history. Given the two different work practices, a direct comparison between the number and types of branches before and after the adoption of DVC is not appropriate.

Ideally, we would compare the history of the development of a single software system with and without light-weight, atomic branching. This is economically infeasible and fraught: even if one could afford to develop the same system twice, once under CVC and once under DVC, one must also control for other, confounding factors in the two version control treatments. Nor can we examine, post facto, the history of a CVC, and extract from it threaded workflows of related commits, each corresponding to a specific feature. Our approach is to flatten the history of a system using a DVC (with light-weight, atomic branches) onto a single workflow line to produce the commit sequence $D$, the system history’s branch-free twin. This branch-free twin allows us to contemplate how the same system might have been developed using a CVC, and thus gauge the difference with the actual DVC history.

An initial, parallel, and independent workflow against a single mainline is date-ordered, or precisely $D$. $D$ does not model a CVC workflow, which has adapted to handle the conflicts and problems that arise from naively working without coordination against a single mainline. $D$ does, however, capture the work to resolve conflict and avoid distraction that has engendered and is unobservable in a CVC history, like the reworking of a large patch into more manageable chunks on a project’s mailing list [Weißgerber et al. 2008].

Figure 3 illustrates our flattening technique. At the top of the figure is the original, branched history of the system; the bottom depicts the result of projecting the
commits that comprise the system’s history onto a single line in temporal order\textsuperscript{3}. The merge commit M that joins the two branches falls out since it is meaningless in the date-flattened mainline, where the work to merge the two branches occurs, piecemeal, as each commit is written.

**Threats to construct validity.** As we quantify next, interleaved branch commits leads to a fractured history \( D \). Is this fractured history similar to what might occur in CVC-style development? This is debatable. The coordination cost of a shared mainline in CVC may not be great: the required workflow may be routinely handled by regular communication between developers. A developer using CVC may use her greater awareness of the activities of others to craft their changes to splint and heal rather than crack the shared history. This discipline would reduce both the number of distractions and the level of dispersion that we report below. However, this awareness is itself a distraction and an annoyance. Because developers using a CVC are likely to commit less often to avoid conflict or breaking the build, our flattening technique, in the worst case, indicates how often developers would like to commit.

### 4.2.2 Distractions

When atomic, lightweight branches are not used, all changes occur on the mainline and a developer may need to merge and integrate changes that are unstable and transitory or only tangentially related to her work. The attendant distractions can slow development. When light-weight branches are available, a developer can isolate related changes, such as those changes required to implement a feature, into branches to minimize such distractions.

Distractions are a form of task interruption. Prior literature has shown that task interruptions have a serious effect on developer productivity. Recovering from interruptions can be difficult and time-consuming: developers must mentally juggle goals, decisions, hypotheses, and interpretations related to their task, or will risk inserting bugs. In a study at Microsoft [LaToza et al. 2006], 62% of developers said that recovering from interruptions is a substantial problem. Van Solingen [van Solingen et al. 1998] found that interruptions are the most problematic when they occur during programming activities, which is most often the case when a developer is checking in changes or updating their working code base. DeMarco observed that resuming after an interrupt often takes at least 15 minutes [DeMarco and Lister 1987]. Parnin *et al.* [Parnin and Rugaber 2009] instrumented Visual Studio and Eclipse to observe the time taken to resume development tasks. While they found some strategies for mitigating the effects, developers were able to begin editing within a minute of starting a task only 10% of the time and took over 30 minutes in 30% percent of the cases. While these papers consider the effect of interruptions in broader terms, they do support the claim that task interruptions (and thus distractions, as we define them) diminish productivity.

Using data mined from the Linux kernel, we quantitatively evaluate the number of distractions that a developer avoids through the use of DVC and lightweight branches. Here we present our methodology and results.

By analogy to numeric intervals, \( D(x, y) \) denotes the subsequence of commits

\textsuperscript{3}For simplicity, we assume no ties. In the case of git, timestamps record seconds, so ties are rare in practice and can be deterministically broken by treating a commit’s id (a hash) as an ordinal.
between $x$ and $y$ in $D$. For the commit $c$, let $a$ denote its most recent ancestor, either its branch parent or its closest grandparent on $D$ when its branch parent is a merge commit. In Figure 3, for example, commit 4’s ancestor is 2, while commit 5’s ancestor is its grandparent 4, since its parent M is a merge commit.

Reviewing Changes Since Last Commit. One simple form of distraction is the cost of finding your place after you have been interrupted. Consider the developer who commits intermediate work on Friday and, on Monday, reviews the VC history to find her place. If her colleagues worked through the weekend, she might have to wade through a number of commits before finding her Friday commit. After restarting work, she eventually makes another commit, $c$. The number of commits a developer would have to scan in this case is precisely the length of $D(a, c)$, or $|D(a, c)|$. Table I shows the proportion of commits for which $|D(a, c)| > 0$ across a selection of our projects, chosen because they have the largest code bases and communities and are widely-used and well-known. Nonetheless, these projects are representative of all projects examined. Across all projects analyzed, the average proportion of commits for which $|D(a, c)| > 0$ was 15.8%. Almost one third of the commits in Linux and the X.org X server require more work for the developer to “find his place” in $D$, the flattened history, than in DVC. GNU Autoconf and Coreutils use branching very little and are not under very active development, which may explain the small proportion of commits that require finding one’s place. Wine is an anomaly in that its use of the version control system has changed very little since moving to DVC, in terms of branching and multiple distributed repositories.

Verifying Assumptions. When a developer is not insulated on a branch, i.e. when she must work on $D$, she faces the distraction of integration work intruding into feature development work. Even when overt conflict does not occur, she still must verify any assumptions she made about code on which her new feature depends.

Figure 4 illustrates the formalism we introduce to measure these distractions. The line at the left represents $D$, the date-flattened DVC history. Ovals on $D$ represent commits. Each commit $c$ defines a changeset, a set of files that it modifies. In the figure, these modified files are the rectangles stacked above each commit. When $F$
is the set of files in a source code repository, \( f_m : C \to 2^F \) extracts the changeset from a particular commit.

Specifically, \( c \) is a commit whose nearest, non-merge ancestor in the original DVC is \( a \), and \( D(a, c) \) represents the commits that developers made to other branches in DVC in the intervening time. In particular, each commit \( w \) in \( D(a, c) \) defines a changeset. Definition 4.1 formalizes the set of files changed in a sequence of commits.

To model the distractions a developer may encounter when committing \( c \), we examine the intersection \( c \)'s changeset, \( f_m(c) \), and the files in \( D(a, c) \), \( F_i \). At the right of Figure 4, the fraction of the number of files in the intersection divided by the number of files in \( c \) is the index of similarity we use to measure distractions. If the index of similarity exceeds a threshold, \( \delta \), then \( c \) is distracted, as specified in Definition 4.2.

**Definition 4.1 (Intervening Files).** The files modified in \( D(a, c) \) “intervene” between \( c \) and \( a \), its nearest ancestor in \( D \). That nearest ancestor \( a \) is either the branch parent of \( c \) when the branch parent of \( c \) is not a merge commit, or \( a \) is one of the branch grandparents. These files therefore change the state of the file system into which \( c \) is written. The set of intervening files is

\[
F_i = \bigcup_{w \in D(a,c)} f_m(w)
\]

If \( c \) modifies \( f \in F_i \), an outright conflict could occur. Lack of overt conflict may be even more distracting as \( c \)'s author must review each file in \( f_m(c) \cap F_i \) to make sure that the non-conflicting changes do not invalidate an assumption on which \( c \) depends. For instance, one of the commits in \( D(x, c) \) could have changed the semantics of a function used in \( c \). This definition requires a commit’s context to have changed in \( D \): If \( c \) is adjacent to its parent or one its grandparents on \( D \), \( D(a, c) = \emptyset \) and therefore so does \( F_i \).
Intuitively, the commit \( c \) is *distracted* if commits fall between it and its branch parent (or grandparents if its branch parent is a merge commit) on \( D \) and one those intervening commits changed a file that \( c \) also modified. In Figure 3, all the commits except commits 1 and 5 are potentially distracted. Definition 4.2 captures this intuition.

**Definition 4.2** (Distraction). The commit \( c \in D \) is distracted if \( F_i \neq \emptyset \) and

\[
\frac{|F_i \cap f_m(c)|}{|f_m(c)|} \geq \delta, \quad \text{for } \delta \in [0..1]
\]

We require \( c \in D \) to eliminate merge commits, as the work they represent has already occurred in \( D \), the date-flattened line. In Figure 3, all the merge work \( M \) required has already occurred by the time 4 has been written in \( D \). A commit cannot be distracted by commits whose changesets do not overlap its changeset. Here, we have defined an index similarity relative to \( c \)'s changeset vs all the commits on \( D \) between \( c \) and its nearest ancestor \( a \) because the point is to try capture the work to commit \( c \) given the state of the file system in \( D \) when \( c \) would have been written into \( D \), which includes the specified commits and their changesets.

Our metric naturally follows from this definition of distraction: we plot the proportion of commits in the date-flattened line that are distracted as \( \delta \) varies. So at \( \delta = 0 \), we plot proportion of all commits whose \( F_i \neq \emptyset \), after which we show the proportion of all commits that satisfy each setting of \( \delta \). Figure 5 depicts the resulting curve for the Linux kernel. The y-axis jumps between 5 and 29 to span an order of magnitude in order to capture \( \delta = 0 \) when we ignore whether \( F_i \) and \( f_m(c) \) overlap while zooming the majority of the data points in \( 0 < y < 5 \) where \( \delta \) ranges from 0.1 to 1. Even at \( \delta = 1 \), i.e. when we require \( f_m(c) \subseteq F_i \), 2.8\% commits are distracted. After calculating the 95\% confidence intervals, we find that a commit \( c \) modifies a file that intervenes between \( c \) and its ancestor \( a \) on \( D \) with a confidence interval of 4.47\% to 4.69\% of the time. This corresponds to the point in Figure 5 with an index of similarity of 0.1. All of the files in a commit \( c \) intervene between \( c \) and its ancestor \( a \) with a confidence interval of 2.47\% to 2.93\% of the time. This corresponds to the point with index of similarity 1.0. Thus, a non-empty overlap occurs approximately once every 22 commits and a complete overlap every 35 commits. This rate is an under-approximation, as \( D \) is created from the perfected repository that does not include commits that were rejected after review or sequences of commits by the same author that were collapsed [Bird et al. 2009]. Although not all of these overlaps cause an overt conflict, they represent changes that a developer needs to examine and possibly correct, forcing a developer to interrupt development work with integration work. Clearly, atomic branches reduce the distraction of resolving a conflict or verifying assumptions about changed code. Therefore, we conclude that our hypothesis — DVC branches better insulate developers than CVC branches do — holds.

**Threats to construct validity.** As noted above, \( D \) may overstate the distractions faced by a developer. Nonetheless, developers using branches have more freedom. There is little need to consider “when should I commit to cause the least amount of disruption to my work and the work of others?” Developers are freed to work in parallel and isolation. Further, they can work within VC on “experimental”
or “speculative” code which may not ever end up being integrated into the main line. Our analysis assumes that all distractions waste time, which may not always be the case. In Figure 3, 3 may benefit from 2’s changes, i.e. if 2 deletes code that 3 would have had to uselessly modify when insulated on a branch. Our first metric, which reports the average length of $D(a, c)$, assumes that a developer, at least partially, relies on VC history to find their place. In practice, developers may turn to other means, perhaps as mundane as a hand-written notes or a “todo” email sent to self.

4.2.3 Cohesion. Cohesion measures the extent to which related aspects of an artifact are kept together and unrelated aspects are kept out. If atomic branches are being used for isolated development with episodic collaboration, we should find them to be cohesive; they should encapsulate a set of related changes. A highly cohesive branch is easier to revert if there is a problem and to backport to an older release with which it shares a common ancestor. Branch cohesion is so desirable that git includes a feature, called rebasing, for “perfecting” the history of a branch and to combine, break apart, remove, and modify commits so that the history is easier to understand [Kernel.org 2009b].

Large systems, like the Linux kernel, structure their files in a modular manner. Files that perform similar functions are closer in the directory hierarchy than files that perform dissimilar functions [Bowman et al. 1999], thus the directory structure loosely mirrors the system architecture. To determine how “cohesive” a set of changes are, we measure how far source files are from each other in the directory tree. Two Files in the same directory have a distance of zero (i.e. the highest level of cohesion), while the distance for files in different directories is the number of directories between the two files in the hierarchy. We only include `.c` source files as...
Bowman [Bowman et al. 1999] found that header files for the entire system often are located in one directory.

Let $d : F \times F \rightarrow N_0$ denote the directory distance of two files. Each commit defines a set of files, its changeset. The cohesion of a single commit is the multiset of directory distances formed from the files in its changeset. A branch is a “straight line” sequence of commits, $B = c_1, \ldots, c_n$, without a merge or a branch. For the branch $B$, let $B_d$ be the multiset of directory distances formed over the union of all its changesets in Equation 1.

$$B_d = \{d(f, f') : f, f' \in \bigcup_{c \in B} f_m(c)\}$$

(1)

**Definition 4.3 (Branch Cohesion).** The *branch cohesion of $B$ is the average of the directory distances in $B_d$:*

$$B_c = \sum_{d \in B_d} \frac{d}{|B_d|}$$

To determine if developers use atomic, lightweight branches to isolate cohesive changes, we compare the cohesion of observed branches in the history of the Linux kernel against the cohesion of simulated branches of equivalent length over the date-flattened line, $D$, using Monte Carlo simulation. Figure 6 depicts this simulation. We first measure the length of each branch in the observed Linux kernel DVC (left) and create a multi-set of branch lengths (middle). We then randomly project these branch lengths (which sum to precisely the length of $D$) onto $D$ to partition $D$ into simulated branches (right). Thus, the distribution of branch lengths is exactly the same as the observed distribution of branch lengths in the Linux kernel history; specifically, this is the distribution shown in Figure 7a. We then compute the branch cohesion for each simulated branch of each length. If developers generally work together on cohesive sets of files in branches then the branch cohesion for branches of length $n$ in the observed DVC history will be higher than the cohesion for sequences of commits with length $n$ in $D$. We performed 1,000 rounds in our Monte Carlo simulation.

Figure 7a is a boxplot of the lengths of observed branches in the history of the Linux kernel. As Figure 7a makes evident, the distribution is positively-skewed.

Since 90% of the Linux kernel branches have length less than 35 commits, we truncated Figure 7b at 35. Branches longer than 35 commits had fewer than 25 instances, giving too small a sample to produce meaningful results.

Figure 7b plots the mean of cohesion of observed Linux kernel branches (black diamonds) against the mean of the means of the cohesion of the simulated branches (black circles). We report the mean of the means at each branch length for the 1,000 simulations and provide a 95% confidence interval (the vertical lines). With the exception of branch length 34, which is not statistically significant (red square), the observed branches are more cohesive than the simulated branches at each length with $p < 0.05$.

Examining the magnitude of the differences in cohesion, we see that at branch length two (the minimum), pairs of files committed on observed branches are 0.12 directories closer together on average than pairs files in date-flattened commits, while the difference is 1.5 directories at branch length 32 (the maximum). These differences may appear small, but note that a difference of 1 means that for each pair of files the distance between them is at least one directory further apart in the code base on a simulated branch than on the observed branch. This effect looms larger when one recognizes that most branches have modifications of tens to hundreds of files in them.

This point is further underscored by correlating this difference to the branch length. As can be seen from Figure 7b, as branches become longer, the observed branches become increasingly more cohesive relative to the simulated branches (Spearman correlation: $r = .69, p \ll .001$). It is clear that developers group related changes on branches and that the larger the number of changes the more important this grouping becomes. Thus, we conclude that our hypothesis, that DVC branches exhibit greater cohesion, holds.

Threats to construct validity. This result depends on \( D \), the date-flattened line, and inherits its threats to validity. Cross-cutting concerns, by definition, are rarely well-correlated with directory distance. Thus, our use of directory distance as a cohesion metric does not capture the cohesion of a cross-cutting change; however, the fact that we found a significant difference in spite of understating the atomicity of a lightweight branch strengthens our result.

4.3 Mega-commits

Without easy branch and merging facilities, our interviewees reported that developers would “pass around large patch sets” or “brain dump” a mega-patch that was almost impossible to review. These large patch sets would contain multiple, sometimes unrelated changes, and it was impossible to “consider each on their own merits without having to swallow the whole thing” (Turnbull, XEmacs [Stephen Turnbull 2009]). This problem was compounded for new developers who did not have commit access and so could not work and commit incremental work in the course of making large changes. Under CVC, developers without commit privileges, as well as core developers who refused to use “painful” (Sperber, XEmacs [Michael Sperber 2009]) feature branches were effectively reduced to working in a time before version control. The following quotation illustrates problems with mega-commits.

“Because we’d have these large changes that would go in all at once, it would be really difficult to find the source of problems. For example, if you wanted to find a change that was responsible for certain problems, you would often go back [in history] . . . and pretty soon you’d find one of these ‘mega’ patches . . . that would essentially change every file in the system and would lump together sets of unrelated changes . . . [these mega changes made it] really, really difficult to track down what change was responsible for a given problem, it makes software maintenance really difficult. ”

Sperber, XEmacs [Michael Sperber 2009]

Quantitatively, we found mixed support for these views. We looked at three measures of mega-commits: the median size of all commits, the median size of the largest commits, and the number of commits until a mega-commit was reached (a deadend).

Commit size. Although statistically significant, the median size of a commit was only two lines smaller after switching to a DVC [Bird et al. 2009]. This difference is so small as to be uninteresting. This result is not unexpected as OSS projects already require commits to be “small, independent, and complete” before they are committed [Rigby et al. 2008]. The median of 15 changed lines is so small that it is unlikely that commits will become even smaller as a result of using a DVC.

Largest commit size. We did, however, expect the size of the largest commits (i.e., mega-commits) to be smaller post-DVC than pre-DVC because with DVC commits can be individually inserted onto a branch instead of lumped into one large patch. To examine this we looked at the top 5% and 10% of commits pre- and post-DVC. We examined them at monthly intervals because the pre-DVC history was typically longer and would, by virtue of including a longer history, have a larger mega-commit size. Although some projects bore out our expectation, many did not, leaving an inconclusive result.
Deadends. As the previous quotation illustrates, when looking back through history for a defect, a mega-commit became a stopping point in the search — a deadend. To assess whether DVC produced fewer deadends, we count the number of commits before a mega-commit. Based on the above quotation, we would expect pre-DVC to have fewer human understandable sized commits before a deadend. We varied the size of a mega-commit from 100 to 1000 changed lines. Again our results were inconclusive. Many projects did not have a sufficiently long history using DVC to yield statistically significant results at all mega-commit sizes. When the results were statistically significant, there was no clear trend. Although our qualitative findings indicate a clear trend toward a reduction in the number of mega-commits, our quantitative results remain inconclusive. There are a number of confounds that may have influenced these results. First, many projects did not have a long enough DVC history to produce statistically significant results in all of our measures. Second, developers are still adjusting to DVC and may not use branches to break up larger commits as the project leads, the people we interviewed, would like them to do. Finally, as we discuss later (see §6), there continues to be a large number of patches posted to the mailing list even when the project uses a DVC. The mailing list continues to serve as a place where a patch is broken down into smaller, perfected components before being committed.

5. GOVERNANCE

OSS licenses allow developers to freely take the source code and start another (perhaps competing) project without requiring permission from the original project [German and Hassan 2009]4. Although a well-established, reputable project cannot stop others from modifying and releasing the code, it typically has a reputation for quality, and owns several trademarks. For example, although anyone can release their own version of the Apache httpd server, only the Apache Foundation can use the trademark “Apache” on the code it chooses for releases. Thus, governance in large OSS projects is critically concerned with who has write access to the repository that produces releases that are approved by the known and respected project. This section examines how different OSS governance structures influence the choice and use of DVC and CVC.

5.1 Centralization vs. Distribution

DVC may not be compatible with every organizational structure. We argue for the following conjectures:

(1) DVC better serves a centralized (dictatorship) social structure.
(2) CVC better serves a decentralized (community of peers) social structure.

In this discussion, \( n \) is the number of programmers actively working on a project. A pull occurs anytime a programmer requests and merges another programmer’s changes into his or her working copy or repository. A push occurs when a programmer adds changes to a repository they do not own. An exchange is a push or a pull. In Figure 8, we examine the number of exchanges required for a programmer to obtain the most recent changes under different combinations of VC and governance.

---

4However, certain licenses do enforce other requirements.

In Figure 8a, a programmer needs only pull once to get the most recent code base when using CVC. Figure 8b examines the use of DVC in a peer setting. Since every programmer has their own copy of the project, in the worst case (i.e., when every programmer makes changes) a programmer must make $n - 1$ exchanges to acquire the most recent code base. The number of pulls in a peer group using DVC grows with the number of programmers. For a small group, $n - 1$ pulls may not prohibitive, but once the group grows too large, the effort required to stay up to date will become unmanageable or, in the best case, tedious and time-consuming. Using CVC with a peer group also requires that each programmer is fully aware of everyone in the peer group so as not to miss any important changes.

The exploding pull problem disappears in a project with a strong hierarchy, centered on a dictator who is a star programmer or integrator, as Figure 8d depicts. We use $h_p$ to denote the lieutenants of the programmer $p$ in the hierarchy shown in Figure 8d. This hierarchical social structure restricts the number of other programmers with whom $p$ exchanges changes. Thus, $p$ only needs to pull changes from each of his lieutenants and from the dictator $D$, resulting in $h_p + 1$ exchanges. By definition, the dictator has the most recent changes published outside of $p$’s hierarchy. Hierarchies form to keep the number of individuals one has to deal with on a human level [Simon 1997]. In practice, as a project grows, $h_p$ therefore remains
Fig. 9: Exchanges needed for a dictator to update his repository from the rest of the development team using CVC and DVC.

much smaller than the fully connected graph — \( h + 1 \ll n - 1 \).

The number of exchanges differs for the dictator. In Figure 9a, the dictator obtains and reviews changes from all other programmers, yielding \( n - 1 \) exchanges to ensure that his repository is up-to-date when using CVC. Like the peer scenario using DVC (Figure 8b), this does not scale well. In contrast, a dictator needs only pull and review changes from those immediately beneath him, \( h_D \), in the hierarchy when utilizing DVC, depicted in Figure 9b. This structure relies on a chain-of-trust in the hierarchy, such that the dictator’s lieutenants are trusted to review and make decisions about changes that flow through them to the dictator (as occurs in the Linux kernel). This means the number of exchanges the dictator requires to obtain the most recent changes remains constant as the community of programmers grows.

It is certainly possible for a project using CVC to form a hierarchy by allowing different team members to maintain branches and move changes from one branch to another towards a dictator’s branch. However, our interviews with developers and our surveys of discussions on project mailing lists indicate that this does not happen in practice: the use of branches in CVC in such a flexible fashion is difficult.

5.2 Case Study

Perhaps it is not surprising that OSS developers are taking to DVC with such enthusiasm; most projects have a very small number of developers and usually one developer does most of the coding [Krishnamurthy 2002], a natural dictatorship. However, most OSS projects are not successful [Wang 2007]. Here, we examine two large, successful, mature projects. Linux is the obvious large project that uses DVC, viz. git. To balance Linux, we select FreeBSD, which continues to use CVS, a CVC. We do not compare these projects technically or in terms of productivity, since such comparisons are fraught with confounds. Instead, we use them to assess the validity of our theoretical argument that a democratic project will use a CVC while a dictatorial project will use a DVC.

We first examine the policies of the two projects. Using the policy to determine the supposed organization structure of the projects, we develop a metric to determine the degree to which a project is a hierarchy.

FreeBSD policy: The FreeBSD project is organized as a foundation or group
of peers [FreeBSD.org 2009]. Developers who have demonstrated their aptitude can be voted into the foundation and receive commit access to the central VC repository. Within the foundation, consensus and voting determine policy and resolve controversial decisions. Developers, who do not have commit access, must convince at least one of the core developers to commit their code. In effect, FreeBSD is an oligarchy in which core developers have a vote, while developers outside of the core group can only voice their opinion.

Linux kernel policy: The Linux kernel is organized as a hierarchy or chain-of-trust. At the top of the chain, the “benevolent” dictator, Torvalds, ultimately controls what is put into an “official” release. Beneath Torvalds are a small number of his “lieutenants” whom he trusts. Each lieutenant is responsible for a section of the project (e.g., Miller maintains the networking aspects of Linux) or a previous release of Linux. In turn, these lieutenants trust a small group of individuals. Code flows from less well-known individuals through a series of progressively more trusted individuals. As the code moves up through the chain-of-trust, each individual vets and signs off on it [Torvalds 2004].

5.2.1 Hierarchy vs Peer Group. Considering our conjectures and the policies of Linux and FreeBSD, one could hypothesize that Linux, which uses a DVC, is organized in a more hierarchical manner than FreeBSD, which uses CVC.

To quantitatively test this hypothesis, we define a metric to assess the degree to which a social network is a hierarchical. Our data source is the VC commit logs. We use a specific event, “signing-off” as evidence of a hierarchical relationship. Typically a developer of higher status reviews and signs off on code written by a lower-status developer. VC systems record these actions in the commit logs. One can often observe a chain-of-trust, where code moves up through the chain of developers, as each developer adds his or her sign-off information to the commit log. From these chains of sign-offs, we can create links between the author of the code and the individuals who signed-off on the code.

In a pure hierarchy, no individual will review or sign-off on any individual above them in the hierarchy. If Linux is a pure hierarchy, nobody would ever sign-off Torvalds’ work. In contrast, in a peer group, a pair of individuals would sign-off on each other’s work. On the continuum from pure hierarchies and pure peer groups, our metric examines the degree to which relationships between pairs of developers are hierarchical. Our metric is similar to Krackhardt’s [Krackhardt 1994], but takes into account the magnitude of the relationship (i.e. how many reviews exist between pairs).

For any pair of developers, let \( x \) be the number of times developer \( X \) has signed-off on developer \( Y \), and let \( y \) be the number of times developer \( Y \) has signed off on developer \( X \). For every pair of developers where \( x > 0 \lor y > 0 \), Equation 2 defines the degree of hierarchy.

\[
h = \frac{|x - y|}{x + y}
\]  

For a perfect hierarchy, \( h = 1 \) and for a perfect peer group \( h = 0 \).

Results: We now employ our hierarchy metric \( h \) to evaluate, in the context of
this case study, whether our hypothesis holds.

The number of purely hierarchical pairs of developers \( i.e. \ h = 1 \) dominates both distributions: 73\% for FreeBSD and 94\% for Linux. However it is clear that FreeBSD has far fewer purely hierarchical developer pairs. Although OSS projects typically have a small number of core developers that do most of the work, there is a much larger group of developers that submit a small number of contributions [Mockus and Weiss 2008; Dinh-Trong and Bieman 2005]. These developers do not have the authority to sign-off code in either Linux or FreeBSD.

We examine pairs of developer who have reviewed each other at least once \( i.e. \) have a reciprocal relationship. Figure 10 shows that, conditioned on reciprocal relationships, FreeBSD, whose median is \( h = 0.75 \) is again less hierarchical than Linux at \( h = .96 \). A Wilcoxon test indicates that this result is statistically significant at \( p < .001 \).

These results require us to revise our hypothesis.

1. Hierarchical relationships dominate OSS projects. Large OSS projects are oligarchies or dictatorships that have a large number of external developers who do not have sign-off authority. This hypothesis is supported by previous literature on project structure [Mockus and Weiss 2008; Crowston et al. 2006].

2. Socially central projects using DVC (Linux) are organized in a more hierarchical manner than socially distributed projects using CVC (FreeBSD).

Our interviews echoed these findings. Despite claiming to be using a DVC, most of our interviewees were using a DVC system, such as git or hg, in a centralized manner. Each project had a “centralized,” “upstream,” “blessed,” or “golden” branch. What they wanted was easier merging, local history, and the other VC advances discussed in §4 that are now gradually making it into CVC systems, such as SVN [Tigris.org 2008]. They did not want to change their governance structure, nominate a dictator, or have multiple “official” repositories: they kept their central repository, which many implementations of DVC allow. VC usage conforms to a project’s organization and governance structure, not the other way around.

6. CONTINUOUS COLLABORATION

In this section, we examine how developers share changes during development. First, we examine how patches are vetted on the mailing list. Different versions of patches are stored on the mailing list making it a quasi-VC system. Second, we provide evidence that DVC allows developers to form workgroups around development “sandboxes.” These sandboxes allow developers to partition the codebase in ways that may crosscut the directory structure and other partitions. Finally, we describe how developers can maintain their own slightly different, specialized version of a software product — what Gonzalez-Barahona coined as “light forks” [Gonzalez-Barahona 2008].

6.1 VC within Email

OSS prides itself on being a meritocracy where status is earned by good work. However, in order to deal with the large number of contributors and the variable quality of contributions from unknown developers, all large projects using CVC
control the group of contributors who can commit code to the repository from which releases are later produced. In DVC, this is achieved by the “pull” relationship between repositories, as discussed in §5.

The version control repository is not the sole locus for co-ordination between developers, however. Much discussion on the merits of a particular code contribution occurs on the mailing lists. The mailing list archives records all the individuals who have participated in a discussion regardless of their importance. The open, free-rolling, vigorous technical discussions on the mailing lists are joined by both experienced, central developers as well as relatively new hands.

We hypothesize that email discussion of patches is more democratic than VC commits.

Before we test our hypothesis, we examine how email functions as a version control system. The VC repositories for a project typically contain “perfected” changes. The discussion surrounding the change is lost and the history may have been modified (perfected) so that it will be easier for future developers to understand or revert. In contrast, the mailing list, which contains many contributions in the form of patches,
contains the raw, unaltered history of a change, and the related discussion. These discussions occur in email threads and allow developers to discuss, review, and test a change before adding it to the VC system. Developers may post multiple versions of a change to the same thread.

Ideally, a developer makes a new branch for each task [Appleton et al. 1998]. With CVS or SVN it was difficult to create, manage, and merge these branches. As a result, developers did not branch per task (See §4). In contrast, the developers’ mailing list of an OSS project contains thousands of patches [Bird et al. 2007]. Each of these threads represent a “branch per task.” For example, over the last 3.5 years, the Linux kernel Mailing List contains over 75,000 email threads and 75% of these discussions contain at least one patch.

**Results:** To test our hypothesis, we compare the degree of hierarchy found in VC to that of patch discussion in the Linux kernel and FreeBSD projects.

We use the following data to create the email patch social network. The individual who posts a patch to the mailing list is the author and anyone who responds in this email thread is a “reviewer.” If a developer sends a new patch to the same thread (e.g. an updated version of the patch), then the sender, who may be different from the original author, becomes the author and all subsequent responses review this new author. This heuristic works well in practice because threads are typically restricted to a single topic [Rigby et al. 2008].

Again we see that most of the relationships are purely hierarchical: 77% for FreeBSD and 83% for Linux. Although Linux is still clearly has more purely hierarchical pairs of developers than FreeBSD: Linux has 11% fewer pure hierarchical relationships than it does in the VC context. When restricted to developers who have a reciprocal relationship, Linux goes from .96 in the VC case to .46 in the email case. The change in FreeBSD is also substantial: .75 for VC and .43 for email. A Wilcoxon test indicates that these differences are statistically significant at $p ≪ .001$.

With the same caveat from §5.2.1, i.e. most developer relationships are hierarchical, our hypothesis that the email patch discussion network is less hierarchical than VC signed-off-by network appears to be sustained.

As our quantitative results and the following quotation show, patches on the mailing lists or in a bug tracker seem to be an equitable manner for sharing code, and DVC has not yet replaced this approach.

“For some feature branches people have clustered [shared changes directly through git] . . . but on the whole it’s still patches on lighthouse [a bug tracker] to a single [central] repository.”

Koziarski, Ruby on Rails [Michael Koziarski 2009]

**6.2 Sandboxes**

Fogel, a prominent SVN developer, states that DVC is much more difficult for people to grasp than CVC [Fogel 2006]. With CVC changes flow up and down (and publicly) via a central repository. In contrast, DVC facilitates a style of collaboration in which work output can flow sideways (and privately) between collaborators, with no repository being inherently more important or central. These sideways flows are a relatively new concept. As developers start to use DVC, they may start to
cluster around “sandboxes;” repositories where developers can work together on a particular topic, isolating their changes from other developers.

These “sideways” changes represent collaborations between small groups of individuals who are task focused. In previous work [Bird et al. 2008], we found that OSS developers naturally group themselves into task-based groups who disband after the task is completed. These developers work in similar areas of the system and often collaborate via related threads on the developer mailing lists. This formation of teams in order to complete tasks and features is similar to traditional commercial software development in which management creates teams of developers by fiat. In our study of naturally forming teams, we did not observe branches within the repository created for these work topics. In contrast, DVC allows developers to easily create a repository through which they can work on a particular topic and have it promoted to a release branch when it is finished.

In this section, we provide evidence that this is already occurring in the Linux kernel, a project whose developers are very familiar with the DVC model of development. We examine sandbox repositories in the Linux kernel and contrast them with the traditional repositories used in FreeBSD project. We measure how many repositories exist for each project, and how these repositories are being used. In the final section of this paper, we describe how the VC sandboxes could be combined with email threads to produce a richer history of how the system has evolved.

In the period examined, FreeBSD has used 4 different repositories. As Table II shows these repositories have a large number of committers and cover very general topics. Individuals may commit to more than one repository; in total there are 399 committers. Only the “Source” repository Table II is directly related to FreeBSD kernel development.

In contrast, Linux has 62 official git repositories, 3 quilt repositories, and one web page with no repository that cover 650 distinct, specific topics related to kernel development. 557 individuals maintain these topics. Unlike FreeBSD, these individuals do not all contribute to the same repository. Table III provides a sample of the topics with the largest number of repositories and maintainers. It is clear that each repository (sandbox) keeps its maintainers well separated and insulated from each other.

From this analysis, we conclude that that Linux is modularized in two ways. Like FreeBSD the system is modularized in the directory hierarchy [Bowman et al. 1999]. Unlike FreeBSD, Linux also isolates development into sandboxes: repositories dedicated to a particular topic.

Table II: The FreeBSD repositories: a committer may belong to more than one repository.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Committers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source (cvs-src)</td>
<td>309</td>
</tr>
<tr>
<td>Documentation (cvs-doc)</td>
<td>285</td>
</tr>
<tr>
<td>Ports (cvs-ports)</td>
<td>280</td>
</tr>
<tr>
<td>Other projects (cvs-projects)</td>
<td>23</td>
</tr>
<tr>
<td><strong>Totals:</strong> 4 Repositories</td>
<td>399</td>
</tr>
</tbody>
</table>
Table III: A sample of topics and sandboxes in Linux.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Repository</th>
<th>Maintainers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI SUBSYSTEM</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>KERNEL BUILD</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>GFS2 FILE SYSTEM</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9P FILE SYSTEM</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>INFINIBAND SUBSYSTEM</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>PARISC ARCHITECTURE</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>TIPC NETWORK LAYER</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>X86 ARCHITECTURE</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>I2C SUBSYSTEM</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>AUDIT SUBSYSTEM</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>BTTV VIDEO4LINUX</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CRYPTO API</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DISTRIBUTED LOCK MGR</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>IEEE 1394 SUBSYSTEM</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>INPUT DRIVERS</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>INTEL WIRELESS WIFI</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>IPWIRELES DRIVER</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LINUX FOR POWERPC</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td>650</td>
<td>66</td>
</tr>
</tbody>
</table>

6.3 Light-Forks

Under the licenses of most open source projects, anyone can acquire a copy of the project source code, modify it, and create a different, competing project. Historically, forks have been frowned upon because they split communities and developer bases. Examples include XEmacs, a fork of the popular Emacs project, and the three BSDs, OpenBSD, FreeBSD, and NetBSD, which all forked U.C. Berkeley’s original “Berkeley’s Software Distribution” of UNIX. For a more in-depth discussion of forks, including notable instances, their reasons, and repercussions, see Moen [Moen 1999].

In contrast to a fork, a light-fork exists when a developer makes specific changes to part of the system, but keeps the core or most of the code the same. An example of this style of fork is a Linux kernel modified for handheld devices. Google maintains a Linux kernel git repository for Android\(^5\) which contains their own modifications to the kernel. However, because of distributed nature of the git, updates to relevant subsystems are regularly pulled from other trees into the Android tree. The main project usually does not accept changes into the main repository if the changes are too specialized or too narrowly focused, so the developer must maintain his or her own light-fork. Without good branching and merging (See §4), developers would often start from scratch instead of updating their code because the merging would be too painful and messy. The following quotation illustrates the difficulty of working outside the main repository.

Debian also uses the *light-fork* model heavily. A number of build files need to be added and source code modifications are sometimes required to create a Debian package from a project codebase. As these are Debian specific, the project maintainers have little desire to keep these in their own official code base. A large portion of the packages have a debian maintained and hosted DVC which contains the needed added files and modifications in addition to the project codebase. These *light-forks* afford the debian maintainers the ability to keep their codebase alterations in VC while also staying up to date by pulling changes from the official codebase as they occur.

“Having a DVC system is very encouraging to . . . people who are developers in other projects who might want to add a small feature . . . personally I’ve contributed [to many other projects] . . . if it’s [in] CVS or SVN, then I really don’t feel comfortable making a lot of changes . . . because it’s not automatically synchronized to upstream . . . things can get messy.”

Turnbull, XEmacs [Stephen Turnbull 2009]

For the remainder of this section, instead of using quotations, we summarize our interviewees’ experiences with light-forks.

**The maintainers is unavailable:** Behdad Esfahbod of the GNOME project [Behdad Esfahbod 2009] recounted how he was unable to get his patches into a project because the maintainer was too busy. So he created his own DVC repository, made significant changes, and without the need for approval from the maintainer was able to produce his own release. This process would have been far more complicated and the previous maintainer would have had difficulty merging back Behdad’s changes without DVC.

**Maintaining Ports:** In the XEmacs project, a developer maintains a separate distributed repository that contains changes specific to Mac OS X. “Every couple of weeks he pulls everything and makes Carbon [OS X] specific adjustments” (Sperber, XEmacs [Michael Sperber 2009]).

**Industry specific repositories:** Companies associated with the GTK+ project maintain company specific repositories according to their product offerings. Isolating changes on branches makes it easier to keep the non-specific code up-to-date.

**Giving back to the community:** Behdad maintains a project that Google Chrome has started using. Google employees made their own DVC repository and told Behdad about the changes they had made. He was easily able to pull a large number of changes into his main project repository. Often companies will not release code immediately to OSS projects, but hold onto it for some time. Having good merging facilities enables bringing in these large changes.

In summary, when a contribution was not immediately included in the core of a software product, it was very difficult to maintain an up-to-date patch set. By using a separate distributed repository, as in the preceding examples, it was much easier to maintain these specialized changes. Future work is need to assess how much effort is expended in maintaining light-forks, but these examples clearly show that developers already use them and find them valuable.
7. FUTURE DIRECTIONS AND CONCLUSION

We have studied three dimensions of distributed collaborative software development: episodic collaboration, governance, and continuous collaboration and presented a discussion of how they manifest in DVC settings, as contrasted with CVC settings. Our findings and the following quotation make it clear that DVC is not a “silver bullet”.

“I think the community can be slowed down by a poorly matching VCS tool, but fundamentally it’s the human interactions which matter much much more. If you have a productive relationship with your contributors, a tool will help you work better with them. But if the relationship is dysfunctional, it won’t save you. I guess I’m saying that a bad one can make things worse, but a good one can’t really help...”

Koziarski, Ruby on Rails [Michael Koziarski 2009]

While we do not take a strong preferential position of one over the other, there will always be ideologues who will fight over the value of DVC; this paper has attempted to provide a balanced account of this relatively recent technology. Our results indicate that the technology is useful and we hope that developers will use this paper to understand if DVC is right for them and other researchers will expand our work. In this section, we fuse our dimensions together, discuss future directions, and conclude.

Developer’s workflows (episodic collaboration) have impacted the way developers interact and share changes (continuous collaboration). The concept of a sandbox, where developers can share changes around a particular topic, seems promising. Our initial hypothesis was that sandboxes would mean that fewer patches would appear on the mailing list. This does not appear to be the case with Linux; 75% of email threads still contain a patch.

It is clear that sandboxing has not yet moved from perfect or nearly perfected changes, to the realm of unperfected changes. This is a shame, since unperfected changes could substantially benefit from stronger change versioning, and perfected changes could benefit from a link to the discussion that lead to the perfected change. In the future, DVC may incorporate this linkage. The following quotation illustrates the desire for sandboxes that allow for feedback on initial changes.

“[I’m] hoping a lot more development will happen in branches out in the open [i.e. sandbox repositories] ...lots of times people develop something, if they don’t bother to do an SVN branch and work in there ...things just suddenly pop in ‘oh, I’ve been working on something, here it is’, I would hope that even for short [features] ...[by] doing it in the open ...[core and non-developers would get and] give more direct ...[and] continual feedback ...people can watch, get help, etc., etc.”

Cannon, Python [Brett Cannon 2009]

As the previous quotation also illustrates, large “hidden” changes make integration and development more difficult. DVC allows developers to use the Named Stable Bases configuration management pattern, thereby separating integration and development. Integration becomes easier because features are perfected before hitting the release repository. Instead of mixing all the changes by “working off of
head”, new features mixed with bug fixes, developers can isolate their changes into logical groups on a branch. This branch may be shared (i.e. a sandbox) or private. So when a developer does work in the central (release or integration) repository, it is much less likely that preliminary and buggy fixes will be interleaved with the perfected changes he or she is trying to integrate. In this way, developers are less likely to see code with which they are unfamiliar until it has been properly tested and reviewed. DVC can have a profound impact on the way developers work.

Although the organization of OSS projects (governance) seems to be influenced more by the community than the type of VC system used, it is clear that as projects grow larger, they become hierarchical [Crowston et al. 2006]. Within this hierarchy, however, specialized subgroups can work (collaborate) in a distributed manner on specific problems while still maintaining a centralized repository that is used for integration and releases.

The flexibility afforded by DVC essentially makes it a VC superset that includes CVC. Although VC is itself an “essential” complexity of software development, Brooks [Brooks 1987] would call the transition from CVC to DVC a manifestation of “accidental” complexity. Nonetheless, DVC has potential to make releasing, developing, and coordinating large software projects much less painful than its more rigid, exclusively centralized predecessor.

REFERENCES


Linus Torvalds. 2009. Email to dri-devel@lists.sourceforge.net.


