Open Source Development:
Coordination by Means of Continuous Integration

Abstract

Continuous integration is an approach to software development that may be used as a means of coordination, replacing detailed design documentation. We explore the use of continuous integration in two open source projects with the aim of identifying key properties of this approach, presumably also relevant in the context of commercial development. The projects, FreeBSD and Mozilla, both use continuous integration in the sense that nearly all software changes are made as small increments to a single development version, the “trunk”. In addition to help coordination, a major benefit may be motivation of developers (being able to produce quick and visible results) and enhancing developers’ insight into many parts of the software (due to the delegation of responsibility of integration). Possible major costs include the time developers must spend doing integration tests prior to committing to the trunk, and the delay of development and test work due to failed builds on the trunk. A crucial challenge for a project employing continuous integration may be to achieve a balanced definition of the “don't break the build”-rule. Another is to decide whether to isolate the final bug-fixing effort prior to a release on a branch separate from the trunk; this avoids holding back new development on the trunk, but may reintroduce problems akin to “big bang” integration.

1 INTRODUCTION

Open Source software development challenges traditional views on software development; an apparently anarchistic bunch of hackers, working for free, produce widely used and high-quality software. In this paper we explore how two large-scale Open Source projects coordinate the efforts of hundreds of developers, and especially what role is played by continuous integration, which we define loosely as a process where changes are added frequently to a body of software, evolving gradually from development to production state.

1.1 Characteristics of Open Source development

Previous research on Open Source development shows a number of characteristics that may make these projects more difficult to manage and coordinate than more traditional software development projects:

Few specifications: Many of the documents traditionally regarded as essential for coordination seem to be missing from most Open Source projects, including plans and schedules (Mockus, Fielding & Herbsleb, 2002).

Geographical distribution: A study of Linux contributors showed them to "come from a truly worldwide community spanning many organizations" (Dempsey, Weiss, Jones & Greenberg, 2002), and a recent on-line survey of 2,784 open source participants (Ghosh, Glott, Krieger & Robles, 2002) showed that "most of the developers feature networks that consist of rather few people." Integration may be particularly difficult for geographically distributed projects (Herbsleb & Grinter, 1999).

Volunteers: Most developers contribute to the projects in their free time – even though as many as 40% are paid for (some of) their work (Hars & Ou, 2001; Jørgensen, 2001) – and so are presumably less likely to accept to be ordered around and perform tedious tasks.

Self-directed egoists: This characterization of Open Source developers given in (Raymond, 1998) may be crude, but is not too distant from the results from empirical studies. The FLOSS-study (Ghosh et al., 2002) found that "an initial motivation for participation ... aims at individual skills."

1.2 Continuous integration

In all large software engineering projects there is a great need for coordination. Malone & Crowston (1994) define coordination as *managing dependencies between activities*, and identify *shared resources* and *producer/consumer relationships* as two central dependencies. The pool of developers is an important *shared resource*, and there are obvious *producer/consumer relationships* between the program's modules.

Maybe the most important means of coordination in traditional software engineering is the architectural design (see e.g. Pressman, 1992; Hoffer, George & Valacich, 2002); by dividing the code into separate modules and specifying the interfaces of each module, the architectural design makes it possible to coordinate the efforts of individual programmers, working on separate modules and not needing detailed knowledge of each others' work. Also in this literature, the process of *integration testing* has only a minor role. It is generally hoped that a careful and detailed architectural design, describing in detail the interfaces of each module, will make integration a rather trivial task. It is recommended to avoid "big bang" integration, and only add one or a few modules a time.

It is logical that open source projects employing frequent releasing, in the sense of Raymond (1998), must also integrate on a frequent basis; and many Open Source projects do, including FreeBSD and Mozilla. The question remains, however, what may be the broader impact of

such an approach to integration. Several examples of successful use of continuous integration in commercial projects have been documented (Cusumano & Selby, 1997; Ebert, Parro, Suttels & Kolarczyk, 2001; Olsson & Karlsson, 1999); to our best knowledge there has been no publicized studies that establish an overall picture of possible properties of continuous integration, not even for open source projects.

It is our hypothesis that in some Open Source projects, continuous integration can serve as a means of coordination, effectively replacing the coordination role of detailed architectural design documents, often missing in Open Source projects.

1.3 Our study

Our study is a multiple, explorative case study in the sense of (Yin, 1998). We have pulled statistical data from FreeBSD's and Mozilla's repositories in October 2002, studied the projects developers' mailing lists, and drawn on a survey of 72 FreeBSD developers, performed in November 2002, that was also the basis for (Jørgensen, 2001).

The paper is organized as follows:

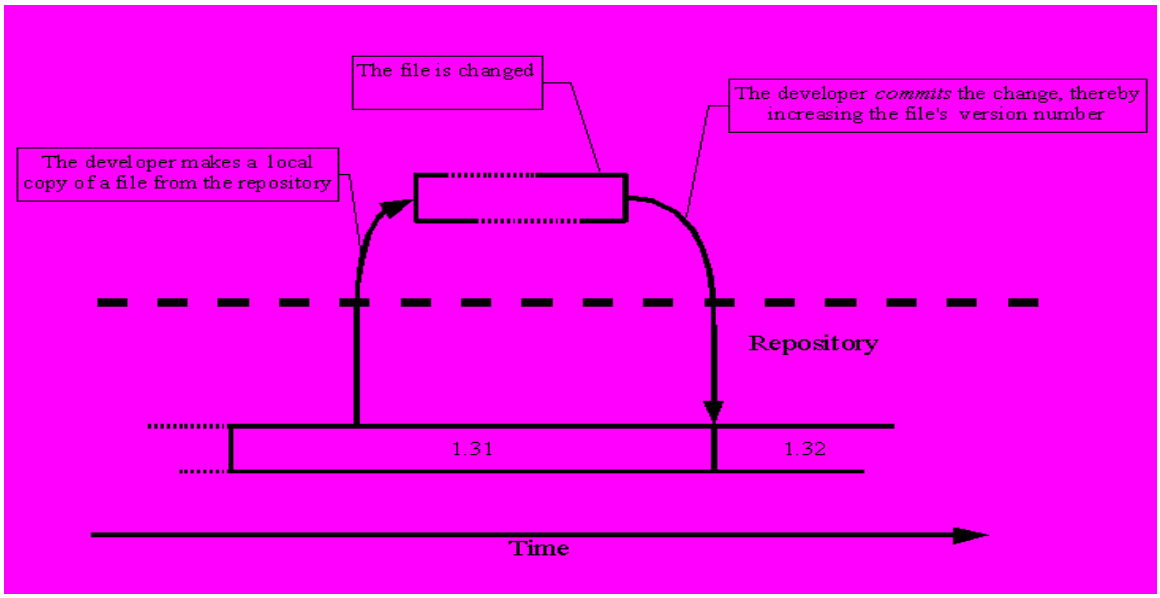
Section 2 introduces the projects and the organization of their repositories into branches. Sections 3-5 describe how continuous integration effects the life cycle of a change, from initialization (section 3), over source code development (section 4), to commit and post-commit test (section 5). Section 6 concludes.

2 THE "TRUNK" IN FREEBSD AND MOZILLA

FreeBSD and Mozilla (see table below with basic facts) each organize their source code repository (from now on: the *repository*) around a main branch (the *trunk*) into which most changes are inserted, and one (Mozilla) or more (FreeBSD) additional branches hosting the projects' production releases.

Name	FreeBSD	Mozilla
Product	Server operating system	Web browser
Major product qualities	Robustness, security	Independence, open interfaces, user interface
Major platforms	Intel x86, Alpha, MIPS, PowerPC	Windows, Linux, MacOS
Approximate size of trunk	29,000 files, 4.7 million lines	40,000 files, 6 million lines
Activity on trunk in October 2002	118 persons *committed* 2,063 changes	107 persons *committed* 2,856 changes
50% of these commits made by	12 developers	7 developers
Project management	9 person elected *core team* constituting a "Board of Directors" (*The FreeBSD Core Team*)	Netscape dominates but delegates authority, especially to 12 *Drivers* (Eich, *Mozilla Development Roadmap*, 2002)

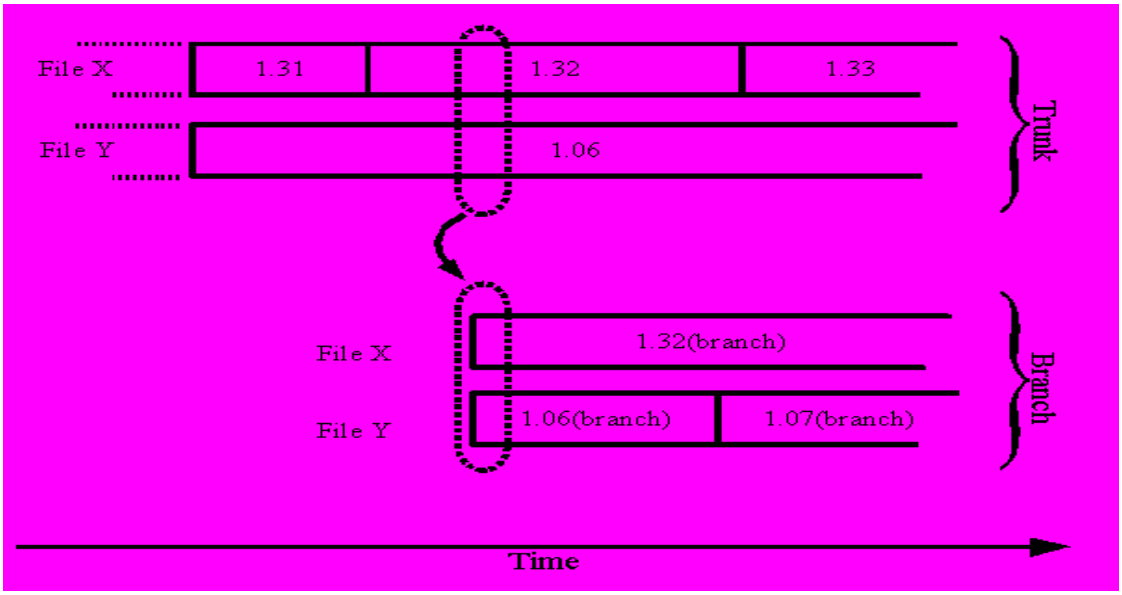
The repositories are stored at central locations, reachable via the Internet (www.freebsd.org and www.mozilla.org). Developers contribute to the projects by continually and in parallel updating (changing, adding, and deleting) repository files; a process controlled by means of CVS augmented with extra tools. Each file change (*commit*) creates a new version; all old file versions remain accessible, thereby making it always possible to go back to an older version of a file if problems or errors are introduced:



The software product that results from *building* with the newest versions of all files is called the *development version*; because files are updated continuously, the development version will also be constantly evolving. The aggregation of all files related to the development version is called the *trunk.*

Because both projects maintain production releases, and so need to isolate new development on a file from file versions part of production releases, there is a need to make branches in the repository.

The figure below illustrates the formation of a *branch.* At a certain time the branch is created as a (logical) copy of the most recent file versions from the trunk; hereafter, changes to the trunk will not directly influence the branch, and vice versa. A developer can specify from which branch he wants to download or commit source files.



In addition to development and production versions, both projects' software is available in a number of versions, adapted to a wide range of different platforms (the table only shows the most important). As a consequence, developers not only need to differentiate between releases (residing on different branches), but must also be able to commit changes intended for specific platforms; this is implemented using conditional compilation rather than branching.

3 INITIALIZATION OF WORK ON A CHANGE

A common characteristic of the way work on a change is initialized in FreeBSD and Mozilla is the lack of explicit guidelines. On the one hand, since work on a change may become visible to the project as a whole only when committed (see section 5), the set of changes on which there is pre-commit work at a given point in time is difficult to control, predict, or even obtain a summary view of. On the other hand, the rule that changes, once committed, must preserve the development version in a working state influences the way work is broken down, namely into small contributions that can be added incrementally.

3.1 Requirements collection and assignment

We use the notions of *requirement*, *task* and *change* as follows:

A *requirement* is a modification the project would like to make to its software. A *task* is the work needed to implement a requirement (by one or more developers, as one or more changes); a task includes, very notably, the responsibility of doing pre-commit testing (see section 4). A *change* is the set of modifications made to one or more files as part of a single commit.

Despite the voluntary nature of work, there is a concerted effort to prioritize, in particular in relation to bug-fixing. Both projects very actively encourage the submission of bug reports to their public bug reporting system. The reports flowing into these sophisticated tools are the projects' major source of new requirements. In contrast, the projects do not encourage the submission of "wish lists" of new features.

There is strong emphasis on classification of bugs. For example, Mozilla processes bugs using states (open, analyzed, etc.); severity (including blocker, critical, and trivial); keywords such

as `mozilla1.0` indicating fixing the bug should precede the 1.0 release; and by linking the bug as a dependency (a presumed precondition) for a release. The way a bug is classified influences the chances that scarce development resources are assigned to it:

> Feel free to propose [linking the bug as a release dependency], but please ask yourself: Is fixing this bug vital to […] 1.0? Is there no alternative to fixing the bug that frees people to work on other 1.0 bugs? What goes wrong if we don't fix the bug, and just live with it for 1.0? (Eich, *Mozilla 1.0 Manifesto*, 2002).

Both projects have a notion of code ownership, in the sense that a "maintainer" (FreeBSD) or "module owner" (Mozilla) is assigned to most files, or the directories or modules they belong to.

> The maintainer owns and is responsible for that code. This means that he is responsible for fixing bugs and answering problems reports […]. (Kamp, 1996)

3.2 Work breakdown

Code ownership does not prevent dependencies between concurrent work on different tasks involving the same files; work on a task involving files "owned" by someone else is by no means excluded. Rather, code ownership is a mechanism for coordination via a consensus process:

> [A commit should happen] only once something resembling consensus has been reached (*The FreeBSD Committers' Big List of Rules*)

In both projects there is work on tasks that involve a large number of files, giving rise to dependencies with concurrent work on other tasks involving the same files. An example of a very large task is the recent work in FreeBSD on Symmetric Multi-Processing (SMP), which enables the operating system to utilize multiple processors. Work was organized as a project in itself, headed by a project manager, and was broken down into small changes, added incrementally to the trunk. This approach was due in part to "big bang" integration problems experienced in another open source operating system:

> ... they [BSD/OS] went the route of doing the SMP development on a branch, and the divergence between the trunk and the branch quickly became unmanageable. [...] We are completely standing the kernel on its head, and the amount of code changes is the largest of any FreeBSD kernel project taken on thus far. To have done this much development on a branch would have been infeasible. (FreeBSD SMP project manager, November 2000).

Changes related to SMP in FreeBSD have been added incrementally over several years; during several months during the fall in 2000, the trunk was severely "destabilized" by build failures and other errors due to dependencies with concurrent work. The decision in FreeBSD to avoid working on a separate branch can be viewed as a preference to deal with these dependencies on the fly, rather than upon accumulating changes on a separate branch, which was seen as entailing a risk of "big bang" integration problems.

3.3 Design

There is no requirement that design should precede coding, in the sense of writing, discussing, or approving written design documents prior to coding.

Indeed in practice, for the individual requirement there is typically no design document. 31 of the 72 committers surveyed in FreeBSD responded that they had never distributed a design document (defined as a separate document, distinct from a source file).

The lack of an established practice of using design documents as a coordination mechanism should, however, be viewed in the context of the projects being largely maintenance-oriented. FreeBSD has inherited a largely unaltered, basic architectural design from its predecessors, the first versions of which were developed in the late 1970s (*About FreeBSD's Technological Advances*).

Mozilla, being a much younger project with roots that back no further than to the mid-90s, is more in need of providing its own design documentation. For example, the project has developed its own component model (XPCOM) and uses a software layer originally developed by Netscape that provides a platform-independent interface to multithreading (NSPR). These and other complex, project-specific parts of Mozilla are described at an introductory level in (*Hacking Mozilla*) as well as in more detail in a series of publicly available documents (*Core Architecture*).

4 SOURCE CODE DEVELOPMENT

The developer of a change works through the stages of code, review, and test prior to commit, using a copy of the source files residing in a private repository.

4.1 Code

Extensive coding guidelines exist in both projects. These indicate a strong emphasis on quality in a broad sense, including security and internationalization, not merely quality in the sense of avoiding broken builds.

Code is highly visible once committed:

- The repositories are browsable, providing easy public access to all sources files.
- In both projects, an automatic mail message is sent to other developers immediately upon commit of a change.

This visibility encourages developers to strive to produce code that will be perceived to be of high quality. In responding to the statement “Knowing that my contributions may be read by highly competent developers has encouraged me to improve my coding skill”, 57% of the 72 FreeBSD committers surveyed said “yes, significantly”, and 29 % “yes, somewhat”. A committer added:

> Embarrassment is a powerful thing

The visibility of the code is in part due to the project simply being open source, but in part also to the approach of continuous integration: A large number of developers are working with the most recent version of the trunk, and monitor the changes being made because their work depends on them.

4.2 Review

Both projects require that changes be submitted to review prior to commit. If the developer is a committer, this review is the only occasion where others (reviewers) are involved in approving the developer’s work prior to commit.

FreeBSD’s Committer’s Guide rule 2 is:

> Discuss any significant change before committing (*The FreeBSD Committers' Big List of Rules*)

86% of the committers surveyed said that they actually received feedback on their latest change when submitting it to review.

Mozilla has a detailed rule requiring all changes to be reviewed by another committer, and in most cases to be "super-reviewed" as well:

> This 'super-review' will look at the quality of the code itself, its potential effects on other areas of the tree, its use of interfaces, and otherwise its adherence to Mozilla coding guidelines and is done by one or more of a designated group of strong hackers (Eich & Baker, 2002)

4.3 Test

The primary goal of this phase is for the developer to comply with the requirement that his change does not break the build. Both projects state this requirement explicitly and strongly. The implication is that the committer of a change is responsible for its integration, and thus the test stage can be viewed as a stage of integration testing.

The build process is fully automated; however, correcting a broken build can be highly challenging, since the failure may be due to dependencies to files or modules outside of the area of the developer's primary expertise.

The basic "don't break the build" rule

In FreeBSD, the rule is stated as follows:

> If your changes are to the kernel, make sure you can still compile [the kernel]. If your changes are anywhere else, make sure you can still [compile the everything but the kernel] (*The FreeBSD Committers' Big List of Rules*).

In Mozilla, the committer must also run a set of simple tests. The underlying rationale is explained as follows:

> The program has to be stable enough to serve as a platform to code new features. You can't verify new features if the program is crashing at startup or has major runtime flaws. Being able to pull a tree and build it is useless unless you can execute the program long enough [to] create and debug new code (Yeh, 1999).

It is noteworthy that affected work includes not only the use of the trunk to test changes already committed (see section 5 below), but also other developers' work in the pre-commit test stage. This is because this test should be done using the most recent sources, so that the result of the build upon commit is the same as in the test.

Interpretation of the basic rule

A major challenge is for the projects to strike a balance that avoids broken builds that could have been prevented with a reasonable effort by the "guilty" committer, but leaves room for relevant exceptions:

> I can remember one instance where I broke the build every 2-3 days for a period of time; that was necessary [due to the nature of the work]. That was tolerated – I didn't get a single complaint (Interview with FreeBSD committer, November 2000).

Interpretation of the "don't break the build"-rule is particularly called for with respect to the effort a developer should invest to prevent broken builds on *any* platform. Recall from section 2 that FreeBSD and Mozilla are developed for many platforms, out of which 4 and 3, respectively, are particularly prioritized. Due to platform differences a build may succeed on one and fail on another. However, most developers have only access to a single platform, so in practice it will be very difficult to perform trial builds on each prioritized platform before check-in.

In part the problem of broken builds on other platforms is solved by making a set of central build machines available, to which sources can be uploaded and subjected to a trial build, prior to commit; however, this is tedious and is not enforced as a general rule in either project. As a middle road, Mozilla provides so-called portability guides that list a number of rules and recommendations for producing cross-platform software (e.g. (Williams, 1998)).

Thus, while FreeBSD and Mozilla in principle delegate to the developer the responsibility to integrate his change, in practice a major challenge is to strike a reasonable balance, accepting to some degree that developers from time to time break the build and thus disrupt other developers' work.

5 COMMIT AND POST-COMMIT TEST

The final stages a change goes through are *commit*, *community test*, and *stabilization*.

5.1 Commit

A commit of a change is the act of uploading to the trunk the set of file modifications that constitute the change.

Committers have write privileges to the entire repository, and it is up to the committer to decide when the change is sufficiently tested for commit. The ability to commit without awaiting approval is highly motivating:

> I don't feel I'm at the whim of a single person. (FreeBSD developer, November 2000).
>
> […] there is a tremendous satisfaction to the 'see bug, fix bug, see bug fix get incorporated' so that the fix help others' cycle. (FreeBSD developer, November 2000).

This may apply also to developers whose work is paid for by a company:

> I use FreeBSD at work. It is annoying to take a FreeBSD release and then apply local changes every time. When [...] my changes [...] are in the main release [...] I can install a standard FreeBSD release [...] at work and use it right away. (FreeBSD developer, November 2000).

5.2 Community testing

After being committed to the trunk, the change is tested. This is not a systematic test of the specific change, but a build verification and a use test.

Build verification

In FreeBSD, two central machines perform a daily build of the most recent sources. A failed build entails an automatic mail message. However, there is no established process for acting on the automatic messages, and most build errors are detected by currently active developers, reporting the problem to one or more mailing lists.

In Mozilla, there is a well-defined process for a daily verification effort using a cluster of build machines (representing all targeted platforms). At 8 AM (PST) each working day, the build machines download the newest source code, build it, and execute a small number of regression tests.

During Mozilla's build verification, developers that have committed changes since the previous day's verification are said to be "on the hook":

> If you are on the hook, your top priority is to be available to the build team to fix bustages. [...] You are findable. You are either at your desk, or pageable, checking e-mail constantly, or on IRC so that you can be found immediately and can respond to any problems in your code (*Hacking Mozilla with Bonsai*).

Due to a fear of "big bang" integration problems, Mozilla's repository is closed until successful termination of build verification, meaning that no commits (except if part of the corrective effort) are allowed until all three prioritized platforms pass the test; this may last from 2 to several hours. The reason for "closing the tree" for all platforms is given in (Yeh, 1999):

> During the development of Netscape Navigator and Netscape Communicator it was argued many times that [...] we should care less about a particular set of platforms and fix regressions on these "second-class" platforms later. We tried this once. The reason why we don't have Netscape Communicator on Win16 was the result of putting off the recovery of that platform until later. After a couple of weeks recovery became impossible. […] The problems will stack up […] as the codebase moves forward and it never catches up.

Use test

Use testing corresponds to Raymond's "parallel debugging" (Raymond, 1998) and results in a stream of bug reports, the primary source of new requirements (see section 3). Use testing is performed in part by non-developers; this is especially prioritized by Mozilla, where semi-stabilized versions of the trunk are released for this purpose (e.g. release 1.2 of November 26, 2002); and in part by developers, where use testing may coincide with work on other changes in the pre-commit test stage (see section 4).

5.3 Stabilization

A change is production released when it becomes part of a production release of the project's software. We discuss the process leading to major production releases (for example, FreeBSD's 5.0 scheduled for December 15, 2002, and Mozilla's 1.0 of June, 2002). In addition, the projects create minor production releases (FreeBSD 4.6, 4.7, etc.; Mozilla 1.1, 1.2, etc.).

Production releases are created by first declaring a period of stabilization, effectively restricting which changes that are allowed to be committed. During stabilization, changes are subjected to community testing in the same manner as in the previous stage, but only necessary changes are allowed, most notably bug-fixes. In FreeBSD, the stabilization period for 5.0 is scheduled for 7 weeks. In the first 3 weeks, "significant new features must be discussed with the release engineering team"; in the remaining weeks, commits require explicit approval by the team.

When the software is considered sufficiently stable, it is declared a production release. Thus, the stabilization period is sufficient for changing "work in progress" to "production release"; there is no need for separate stages or teams dedicated to releasing, as the software is already in a working state.

In Mozilla, the stabilization period prior to the 1.0 release can be considered to have lasted more than 8 months, beginning with the 0.9.6 release (in November 2001) upon which:

> [...] the trunk is closed to all but a relative few bug fixes, and everyone is focused on testing. (Eich, *Mozilla 1.0 Manifesto*, 2002)

There was indication of pressure from Mozilla developers to relax commit restrictions:

> [...] we're not looking for new features; we want stability, performance [...], tolerably few bugs [..]. Features cost us time [...] those implementing the features […] could instead help fix 1.0 bugs [...]. If you think you must have a feature by 1.0, please be prepared to say why to drivers, and be prepared to hear "we can't support work on that feature until after 1.0 has branched" in reply. (Eich, *Mozilla 1.0 Manifesto*, 2002)

To allow for the resumption of new development on the trunk, the final 2 months of stabilization for Mozilla's 1.0 release took place on a separate branch (created April 2002). This isolation of bug-fixing from new development is in some sense a departure from strict adherence to continuous integration, and indicates the following dilemma associated with stabilization:

Creating a separate stabilization branch allows for resumption of new development, which is otherwise halted when "destabilizing" changes are prohibited on the trunk.

However, a separate stabilization branch makes it necessary for bug-fixes to the trunk to also be made to the branch (and vice versa); it doubles the tasks related to managing a branch (assigned to "branch drivers" in Mozilla), and divides the pool of user/developers between the branches.

> This branch [Mozilla 1.0] obviously entails overhead in driving, merging, reviewing, and testing. (Eich, *Mozilla 1.0 Manifesto*, 2002)

It is obvious that decisions regarding branching should be considered carefully. In both projects it is considered very important to have only one source code trunk, keep this healthy and working at all times, and only branch in order to stabilize upcoming releases. This is very much in line with branch-by-purpose model recommended in (Walrad & Strom, 2002).

6 CONCLUSION

Our case study of FreeBSD and Mozilla as viewed from the perspective of coordination indicates a number of possible advantages, disadvantages, and challenges associated with continuous integration.

At a basic level, it is obvious that the processes actually work; in spite of distributed developers, volunteering for tasks, and without design documents, both projects produce widely used and high-quality software. Where traditional software engineering projects rely on explicit task assignment and detailed design documents for coordination, these projects rely on continuous integration.

Continuous integration may be highly motivating: developers are to a large degree free to choose which task they want to work on and can commit changes without awaiting approval – it feels unbureaucratic and lets the developer see the result of his work quickly become part of the project's software (section 4).

But we have also seen that the advantages of continuous integration don't come without costs:

- The combination of the requirements that the trunk must be "buildable", and that all new development must be committed to the trunk makes heavy demands on developers. With the source code in the trunk changing constantly, implementing a change is a bit like hitting a moving target, and demanding constantly synchronizing local source code with the repository.
- If a commit breaks the build, work on the project is disrupted. Developers will not be able to test local source code changes with the newest source code in the trunk; this

may be just a nuisance, but may also block further work on code changes. Also, users and developers are prevented from community testing the newest software version.

- Several factors help to prevent or limit broken build situations: coding guidelines and code exposure (section 4.1), demands for review before commit (section 4.2), urging developers to avoid broken builds (section 4.3), demands (in Mozilla) that committers be available (section 5.2), central test facilities (section 5.2), and Mozilla's closing of the tree while testing (section 5.2).
- There is a potential conflict between stabilization and new development. If stabilization is isolated on a separate branch, developer resources for testing and managing are divided between community testing and stabilization; a project may have to choose either such a division, or carrying through a prolonged stage of stabilization on the trunk at the cost of holding back new development (section 5).
- An additional challenge is to define in a balanced way the "don't break the build"-rule (section 4). Indication that full compliance may not be feasible was evident in both projects, where most developers do not have a reasonably convenient access to all platforms on which they are (ideally) supposed to test whether their changes may break the build.

Establishing a project culture with strong encouragement to produce code of high quality but also with tolerance and mutual assistance is perhaps the major challenge; indeed, the approach of continuous integration may provide a basis for developing such a culture, since the quality of the project's most recent source codes becomes a point of continuous, project-wide focus.

7 BIBLIOGRAPHY

About FreeBSD's Technological Advances, Available: [http://www.freebsd.org/features.html] (Dec. 1, 2002).

Core Architecture, Available: [http://www.mozilla.org/catalog/architecture/] (Dec. 1, 2002).

The FreeBSD Committers' Big List of Rules, Available: [http://www.freebsd.org/doc/en_US.ISO8859-1/articles/committers-guide/rules.html] (Dec. 1, 2002).

The FreeBSD Core Team, Available: [http://www.freebsd.org/doc/en_US.ISO8859-1/articles/contributors/staff-core.html] (Dec. 1, 2002).

Hacking Mozilla, Available: [http://www.mozilla.org/hacking/] (Dec. 1, 2002).

Hacking Mozilla with Bonsai, Available: [http://www.mozilla.org/hacking/bonsai.html] (Dec. 1, 2002).

Cusumano, M. A. & R. W. Selby (1997), 'How Microsoft Builds Software', *CACM* **40**(6), pp. 53-61.

Dempsey, B., D. Weiss, P. Jones & J. Greenberg (2002), 'Who Is an Open Source Developer? A Qualitative Profile of a Community of Open Source Linux Developers', *CACM* **45**(2), pp. 67-72.

Ebert, C., C. H. Parro, R. Suttels & H. Kolarczyk (2001), Improving Validation Activities in a Global Software Development, *in* 'Proceedings of 23rd International Conference on Software Engineering (ICSE '01)', IEEE, Toronto, Canada, pp. 545-554.

Eich, B. (2002), *Mozilla 1.0 Manifesto*, Available: [http://www.mozilla.org/roadmap/mozilla-1.0.html] (Nov. 15, 2002).

Eich, B. (2002), *Mozilla Development Roadmap*, Available: [http://www.mozilla.org/roadmap.html] (Dec. 1, 2002).

Eich, B. & M. Baker (2002), *Mozilla 'Super-Review'*, Available: [http://www.mozilla.org/hacking/reviewers.html] (Dec. 1, 2002).

Ghosh, R. A., R. Glott, B. Krieger & G. Robles (2002), Part 4: Survey of Developers 'FLOSS Final Report'.

Hars, A. & S. Ou (2001), Working for Free? Motivations of Participating in Open Source Projects, *in* 'Proceedings of 34th Hawaii International Conference on System Sciences', Hawaii.

Herbsleb, J. D. & R. E. Grinter (1999), Splitting the Organization and Integrating the Code: Conway's Law Revisited, *in* 'Proceedings of International Conference on Software Engineering (ICSE '99)', pp. 89-95.

Hoffer, J. A., J. F. George & J. S. Valacich (2002), *Modern System Analysis and Design*, Prentice Hall, Upper Saddle River, New Jersey.

Jørgensen, N. (2001), 'Putting It All in the Trunk: Incremental Software Development in the FreeBSD Open Source Project', *Information Systems Journal* **11**(4), pp. 321-336.

Kamp, P.-H. (1996), *Source Tree Guidelines and Policies*, Available: [http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/policies.html] (Dec. 1, 2002).

Malone, T. W. & K. Crowston (1994), 'The Interdisciplinary Study of Coordination', *ACM Computing Surveys* **26**(1), pp. 87-119.

Mockus, A., R. T. Fielding & J. D. Herbsleb (2002), 'Two Case Studies of Open Source Software Development: Apache and Mozilla', *ACM Transactions on Software Engineering and Methodology* **11**(3), pp. 309-346.

Olsson, K. & E.-A. Karlsson (1999), *Daily Build - the Best of Both Worlds: Rapid Development and Control*, Swedish Engineering Industries, Lund, Sweden.

Pressman, R. S. (1992), *Software Engineering - A Practitioner's Approach,* 3rd (international) ed., McGraw-Hill, New York.

Raymond, E. S. (1998), *The Cathedral and the Bazaar*, Available: [http://www.tuxedo.org/~esr/writings/cathedral-bazaar/] (Dec. 1, 2002).

Walrad, C. & D. Strom (2002), 'The Importance of Branching Models in SCM', *IEEE Computer* **35**(9), pp. 31-38.

Williams, D. (1998), *C++ Portability Guide, version 0.8*, Available: [http://www.mozilla.org/hacking/portable-cpp.html] (Dec. 1, 2002).

Yeh, C. (1999), *Mozilla Tree Verification Process*, Available: [http://www.mozilla.org/build/verification.html] (Dec. 1, 2002).

Yin, R. K. (1998), *Case Study Research: Design and Methods*, Sage Publications, Newbury Park.