12 Incremental and Decentralized Integration in FreeBSD

Niels Jørgensen

There is a tremendous sense of satisfaction to the "see bug, fix bug, see bug fix get incorporated so that the fix helps others" cycle.

—FreeBSD developer

The activity of integration in a software development project may be defined as an assembly of parts. The activity is crucial, because when fragments produced by different programmers are integrated into larger parts, many errors that were previously not visible might emerge. Steve McConnell, in his book *Rapid Development* (McConnell 1996, 406) writes that one of the advantages of doing a daily build of the entire product is that it helps debugging.

> You bring the system to a good known state, and then you keep it there. . . . When the product is built and tested every day, it's much easier to pinpoint why the product is broken on any given day. If the product worked on Day 17 and is broken on Day 18, something that happened between the builds on Days 17 and 18 broke the product.

The FreeBSD project's approach to software integration is highly incremental and decentralized. A stream of bug fixes and new features are integrated into the project's development branch, typically numbering several dozens each day. Integration of a change is the responsibility of the developer working on the change, in two ways. First, the developer has the authority to actually add the change on his own decision, without having to ask someone for approval. Second, the developer is responsible for conducting integration test of his change, including a trial building of the full system and correcting the errors this trial reveals. There is freedom and accountability—changes not properly integrated can be backed out again, and developers risk having their repository privileges revoked.

FreeBSD's delegation of commit authority distinguishes it from projects where only an individual is in control of the repository, such as Linux, and this delegation is, in my view, more in the spirit of open source.

An analysis of the pros and cons of FreeBSD's approach to integration, as attempted in this chapter, may shed light on a wide range of projects that in some way follow Raymond's "release early, release often," since integration is a prerequisite of release—that is, at least if there's anything new in the release! FreeBSD is also interesting as a case of geographically distributed software development. In such projects, integration is generally acknowledged to be an extremely difficult coordination problem; see, for example, Herbsleb and Grinter 1999.

The remainder of the chapter is organized as follows. The first two sections introduce FreeBSD and discuss the notions of integration and coordination in software projects. The remainder of the chapter traces the lifecycle of a change, from work initialization to production release. The conclusion discusses the impact on coordination that may be attributed to FreeBSD's approach to integration.

FreeBSD's Software and Organization

FreeBSD's processes for integration and other activities must support development of extremely complex software by an organization of distributed individuals.

The FreeBSD operating system is a descendant of the Unix variant developed at U.C. Berkeley, dating back to 1977. FreeBSD and its siblings, which include NetBSD and OpenBSD, were found to run 15 percent of the approximately 1.3 million Internet servers covered in a 1999 survey (Zoebelein 1999).

The project's source code repository is publicly available at a Web site (www.freebsd.org), so that any change committed creates a new release immediately. The repository's development branch (or trunk) contains approximately 30,000 files with 5 million lines of code. Approximately 2,000 changes were made to the trunk in October 2002, each typically modifying only a few lines, in one or a few files.

The project participants are the approximately 300 committers; that is, developers having repository write access, plus external contributors whose changes are inserted via committers. More than 1,200 external contributors have contributed to the project's own base of sources, and several thousand users have submitted bug reports. Project leadership is a so-called "core team" with nine members elected by the committers.

In addition to organizing the development of the operating system kernel, the project assumes a role analogous to a Linux distributor such as Red Hat. FreeBSD's release 5.0 of January 2003 included more than *8,000* ported open source programs for server or workstation use with FreeBSD. The project writes comprehensive documentation—for example, the January 2003 release notes for release 5.0 describe several hundred new, mostly minor, features. Various free services to FreeBSD's users are provided—including, most notably, monitoring security—as a basis for issuing warnings and releases of corrected software.

Part of the data underlying this chapter was collected during a survey in November 2000 among FreeBSD's committers (then numbering approximately 200) that received 72 responses. Subsequently, 10 respondents were interviewed via mail. The survey questions were typically directed at the committers' most recent work. For example, the question "When was the last time the committer had caused a broken build?" Results were 8 percent within the last month, 30 percent within the last three months. (Quotations are from the survey and interviews when no other reference is provided.)

Given the absence of hired developers and a corporate building with managers on the top floor, in what sense is FreeBSD an organization, if at all?

Within the fields of systems development and organization theory, the concept of organization is quite broad. FreeBSD's informal organization has similarities with Baskerville's and others' notion of a postmodern organization. "In an era of organizational globalization and competitive information systems, we begin to recognize that undue regularity in an organization and its information system may inhibit adaptation and survival. . . . The post-modern business organization is . . . fluid, flexible, adaptive, open. . . ." (Baskerville, Travies, and Truex 1992, 242–243).

FreeBSD's organization is in some ways also similar to Mintzberg's "adhocracy" archetype: a flat organization of specialists forming small project groups from task to task. Mintzberg considered the adhocracy to be the most appropriate form for postwar corporate organizations depending increasingly on employees' creative work (Mintzberg 1979).

There is, though, also a strong element of continuity in FreeBSD's organization. The technological infrastructure has remained essentially the same from the project's inception in the beginning of the 1990s; for example, e-mail for communication, CVS for version control, the C language for programming, and the make program for building. The operating system's basic design has remained the same for more than a decade. "FreeBSD's distinguished roots derive from the latest BSD software releases from . . . Berkeley. The book *The Design and Implementation of the 4.4BSD*

Operating System . . . thus describes much of FreeBSD's core functionality in detail" (McKusick et al. 1996) [4.4BSD was released in 1993].

Although many new features have been incorporated into FreeBSD, there is a further element of stability in that FreeBSD's usage as an Internet server or workstation has been the source of most of the demand for new features, then and now.

Mintzberg's archetypes for organizations that are more traditional than the adhocracy can be characterized on the basis of how central control is established: standardization of work processes, worker skills, or work output, respectively. FreeBSD bears resemblance with Mintzberg's divisionalized archetype, being split into relatively independent divisions to whom the organization as a whole says "We don't care how you work or what your formal education is, as long as the software you contribute does not break the build."

Integration = Assembly of Parts

In this context, *integration* is assumed to mean all the activities required to assemble the full system from its parts, as in (Herbsleb and Grinter 1999).

In software engineering textbooks, integration and testing are frequently viewed as constituting a phase involving a series of steps, from unit testing to module and subsystem testing to final system testing. Approaches presented include strategies for selecting the order in which to integrate parts, such as top-down or bottom-up, referring to the subroutine call structure among parts. (For example, see Sommerville 2001 and Pressman 2000.) The notion of integration is viewed in the sequel as independent of lifecycle context, but with testing as the major activity as in the classical context. Integration is by no means the act of merely "adding" parts together. This statement is analogous to coding being more than generation of arbitrary character sequences. Integration-related activity is viewed as completed simply when the project or individual *considers* it completed; for example, precommit testing is completed when the developer commits.

The canonical error detected during integration is an interdependency (with another part) error. This type of error ranges from subroutine interfaces (for example, syntax of function calls) to execution semantics (for example, modification of data structures shared with other parts). Some errors are easily detected—for example, syntax errors caught by the C compiler during building, or a crash of a newly built kernel. Other errors are unveiled only by careful analysis, if found at all prior to production release.

Malone and Crowston define coordination as "management of dependencies," and suggest dependency analysis as the key to further insight

into coordination-related phenomena. Producer/consumer relationships and shared resources are among the generic dependencies discussed in Malone and Crowston 1994.

Producer/consumer dependencies may be of interest for analyses of integration: If part A defines a subroutine called by part B, there is a producer/consumer relationship between the developers of A and B. Notably, these coordination dependencies involve *developers*, not software. A developer might find himself depending on another developer's *knowledge* of specific parts; for instance, to correct (technical) dependency errors. At an underlying level, developer time is a limited resource, so there may be a high cost associated with A's developer having to dive deep into part B to overcome a given distribution of knowledge.

Also, a shared resource dependency can be identified in FreeBSD involving the project's development version. At times, the trunk is overloaded with premature changes, leading to build breakage. In the spirit of Malone and Crowston's interdisciplinary approach, one may compare the human activity revolving around the trunk with a computer network: both are limited, shared resources. Network traffic in excess of capacity leads to congestion. Build breakage disrupts work, not just on the "guilty" change, but numerous other changes occurring in various lifecycle phases that rely on a well-functioning trunk.

Division of Organization, Division of Work

Work on a change in FreeBSD can be divided into the following types of activities:

- Pre-integration activities, such as coding and reviewing, where the project's "divisions," the individual developers, have a high degree of freedom to choose whatever approach they prefer; for example, there is no requirement that a change is described in a design document.
- Integration activities, such as precommit testing and parallel debugging, which are controlled more tightly by project rules—for example, the rule that prior to committing a change, a committer must ensure that the change does not break the build.

Parnas's characterization of a module as "a responsibility assignment rather than a subprogram" (Parnas 1972) pinpoints the tendency that organizational rather than software architectural criteria determine the way tasks are decomposed in FreeBSD; namely, into entities small enough to be worked on by an individual. Sixty-five percent of the respondents said that

their last task had been worked on largely by themselves only, with teams consisting of two and three developers each representing 14 percent.

The basic unit in FreeBSD's organization is the maintainer. Most source files are associated with a maintainer, who "owns and is responsible for that code. This means that he is responsible for fixing bugs and answering problem reports" (FreeBSD 2003b).

The project strongly encourages users of FreeBSD to submit problem reports (PRs) to a PR database, which in March 2003 contained more than 3,000 open reports, and is probably the project's main source of new tasks to be worked on.

Maintainers are involved in maintenance in the broadest sense: thirty-eight percent said their last contribution was perfective (a new feature), 29 percent corrective (bug fixing), 14 percent preventive (cleanup), and 10 percent adaptive (a new driver). Typical comments were, "I do all of the above [the four types of changes]; my last commit just happened to be a bugfix" and "But if you had asked a different day, I would answer differently."

Work on code by nonowners may be initialized for a number of reasons. First, regardless of the owner's general obligation to fix bugs, bugs are in fact frequently fixed by others. Nearly half the developers said that within the last month there had been a bugfix to "their" code contributed by someone else. Second, changes may be needed due to dependencies with files owned by others.

To resolve coordination issues arising from someone wanting to change code they do not own, the first step is to determine the identity of the maintainer. Not all source files have a formal maintainer; that is, a person listed in the makefile for the directory. "In cases where the 'maintainership' of something isn't clear, you can also look at the CVS logs for the file(s) in question and see if someone has been working recently or predominantly in that area" (FreeBSD, 2003a). And then, "Changes . . . shall be sent to the maintainer for review before being committed" (FreeBSD 2003b).

The approach recommended for settling disputes is to seek consensus: "[A commit should happen] only once something resembling consensus has been reached" (FreeBSD 2003a).

Motivation

Enthusiasm jumps when there is a running system.

—Brooks 1987

For work on a change to commence, a maintainer or other developer must be motivated to work on it, and for the project in the first place.

In FreeBSD, not only is the development version of the system usually in a working state, but also, the committers have the authority to commit changes to it directly. This delegation of the authority to integrate appears to be very important for motivation. As many as 81 percent of the committers said that they were encouraged a lot by this procedure: "I don't feel I am under the whim of a single person," and "I have submitted code fixes to other projects and been ignored. That was no fun at all."

This may supplement motivating factors found in surveys to be important for open source developers in general, where improvement of technical skills and some kind of altruism are among the top (Hars and Ou 2002 and Ghosh et al. 2002).

In describing why they like FreeBSD's decentralized approach, several committers pointed to the mere practical issue of reducing work. One said, "It is frequently easier to make a change to the code base directly than to explain the change so someone else can do it," and another commented, "Big changes I would have probably done anyway. Small changes . . . I would not have done without commit access."

A large part of the work committers do for FreeBSD is paid for, although of course not by the project as such. Twenty-one percent of the FreeBSD committers said that work on their latest contribution had been fully paid for, and another 22 percent partially paid for. Consistently, Lakhani and Wolf (chap. 1, this volume) found that 40 percent of OSS developers are paid for their work with open source. It is interesting that the decentralized approach to integration is appealing also from the perspective of FreeBSD's paid contributors: "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."

A complementary advantage of the delegation of commit responsibility is that the project is relieved from having to establish a central integration team. McConnell recommends in his analysis of projects that use daily building that a separate team is set up dedicated to building and integration. "On most projects, tending the daily build and keeping the smoke test up to date becomes a big enough task to be an explicit part of someone's job. On large projects, it can become a full-time job for more than one person (McConnell 1996, 408)."

FreeBSD's core team appoints people to various so-called "coordinator" tasks, on a voluntary basis of course. A subset of the coordinator

assignments can be viewed as falling within tasks of configuration management: management of the repository, bug reporting system, and release management. Another subset deals with communication: coordination of the mailing lists, the Web site, the documentation effort, the internationalization effort, as well as coordinating public relations in general. However, there is no build coordinator, team, or the like. Indeed, integrating other people's changes may be viewed as less rewarding, and assignment to the task is used in some projects as a penalty (McConnell 1996, 410).

Planning for Incremental Integration

We are completely standing the kernel on its head, and the amount of code changes is the largest of any FreeBSD kernel project taken thus far.
—FreeBSD's project manager for SMP

While most changes in FreeBSD are implemented by a single developer, some changes are implemented by larger "divisions." An example is FreeBSD's subproject for Symmetric Multiprocessing (SMP), to which approximately 10 developers contributed. SMP is crucial for the exploitation of new cost-effective PCs with multiple processors. An operating system kernel with SMP is able to allocate different threads to execute simultaneously on the various processors of such a PC. Specifically, release 5.0 (March 2003) enables SMP for threads running in kernel mode. The 4.x releases enable SMP only for user-mode threads.

A crucial decision facing large subprojects such as SMP is whether to add changes incrementally to the development branch, or to insulate development on a separate branch and then integrate all at once. In either case, the subproject is responsible for integration. The latter approach may give rise to "big bang integration" problems, causing severe delays and sometimes project failure (McConnell 1996, 406).

The FreeBSD decision in favor of the more incremental approach was influenced by a recent experience in BSD/OS, a sibling 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. . . . To have done this much development on a branch would have been infeasible" (SMP project manager).

The incremental approach implied that the existing kernel was divided gradually into two, three, or more distinct areas in which a separate thread was allowed to run. The SMP project used a classical approach, with a written plan that defined work breakdown and schedule, some design doc-

umentation, and a project manager, and was launched at a large face-to-face meeting. This planned approach may have been necessary to maintain the development version in a working state during such deep kernel surgery.

Code

Parnas was right, and I was wrong.
—Brooks 1995

Brooks, in the original 1975 version of *The Mythical Man-Month*, recommended a process of public coding as a means of quality control via peer pressure and getting to know the detailed semantics of interfaces. In his twentieth anniversary edition, he concluded to the contrary, in favor of Parnas' concept of information hiding in modules.

Coding in FreeBSD is indeed public: the repository is easily browsable via the Web, and an automatic message is sent to a public mailing list summarizing every commit.

From a quality assurance point of view, FreeBSD's public coding enables monitoring of compliance with the project's guidelines for coding, which includes a style guide for the use of the C language and a security guide, for instance, with rules intended to avoid buffer overflows. It also encourages peer pressure to produce high quality code in general. In response to the survey statement "Knowing that my contributions may be read by highly competent developers has encouraged me to improve my coding skills," 57 percent answered "Yes, significantly," and 29 percent said "Yes, somewhat." A committer summarized: "Embarrassment is a powerful thing."

From the point of view of software integration, the public nature of the coding process might compensate to some degree for the lack of design documents; in particular, specifications of interfaces. This compensation is important, because the division of work among FreeBSD's developers appears to reflect the distributed organization (as discussed in the section "Division of Organization, Division of Work"), rather than a division of the product into relatively independent modules. Thirty-two percent said that their last task had required changing related code on which there was concurrent work (most characterized them as minor changes, though). According to the SMP project manager, "One of the things that worried me . . . was that we wouldn't have enough manpower on the SMP project to keep up with the changes other developers were making. . . . [T]he SMP

changes touch huge amounts of code, so having others working on the code at the same time is somewhat disruptive."

To resolve interdependencies with concurrent work, developers watch the project mailing lists. Typical comments are: "By monitoring the mailing lists, I can usually stay on top of these things [related code work]" and "Normally I know who else is in the area," and "I usually follow the lists closely and have a good idea of what is going on."

In response to the statement "Knowing and understanding more about related, ongoing code work would help me integrate my code into the system," 26 percent agreed "Significantly" and 46 percent said "Somewhat". Interdependency with related coding is a central issue that has not been fully resolved in FreeBSD's model.

Review

The project strongly suggests that any change is reviewed before commit. This is the first occasion where the developer will receive feedback on his code. Also, all the subsequent phases in the lifecycle of a change as defined in this chapter might give rise to feedback, and a new lifecycle iteration beginning with coding.

The Committers' Guide rule 2 is "Discuss any significant change *before* committing" (in web page). "This doesn't mean that you ask permission before correcting every obvious syntax error. . . . The very best way of making sure you're on the right track is to have your code reviewed by one or more other committers. . . . When in doubt, ask for review!" (FreeBSD 2003a).

The data indicate that there are frequent reviews in FreeBSD. Code reviewing is the most widespread. Fifty-seven percent had distributed code for reviewing (typically via email) within the last month, and a total of 85 percent within the last three months. Almost everybody (86 percent) said they had actually received feedback the last time they had asked for it, although this may have required some effort. Some responses were, "I have to aggressively solicit feedback if I want comments," and, "If I don't get enough feedback, I can resort to directly mailing those committers who have shown an interest in the area."

Design reviewing is less frequent: within the last three months, only 26 percent had distributed a design proposal, which was defined in a broad sense as a description that was not a source file. Although as many as 93 percent said they actually received feedback, a major obstacle to an increase in review activity appears to be that it is difficult to enlist reviewers. All in

all, there is indication that it would be difficult for the project to introduce mandatory design documents, for instance, describing interfaces or modifications to interfaces, to aide integration: "I did get feedback . . . of the type 'This looks good' . . . but very little useful feedback," and, "Getting solid, constructive comments on design is something like pulling teeth."

Precommit Testing: Don't Break the Build

> Can people please check things before they commit them? I like a working compile at least *once* a week.
>
> —mail message to the developer's list

In the lifecycle of a change, the committer's activities to test the change prior to committing it to the development branch can be viewed as the first activity contributing directly to the integration of the change.

At the heart of FreeBSD's approach to integration is the requirement that committers conduct thorough enough precommit testing so as to ensure, at a minimum, that the change does not break the build of the project's development version. The build is the transformation of source files to executable program, which is an automated process. Breaking it means that the compilation process is aborted, so that an executable program is not produced. There is an ongoing effort to persuade developers to try to comply to this requirement, but at the same time the rule requires pragmatic interpretation.

The main purpose of keeping the build healthy is, in my understanding of FreeBSD's process, that the trunk is vital for debugging; boosting morale is secondary. At one extreme, the debugging purpose would be defeated by a demand that changes are completely free of errors. The other extreme is when the build is overloaded with error-prone changes—then it becomes difficult to identify which newly added changes have caused an error. Moreover, when the trunk can not be built, other testing than the build-test itself is halted.

The don't-break-the-build requirement is stated as rule number 10: "Test your changes before committing them" in the Committers' Guide, where it is explained 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 everything but the kernel]" (FreeBSD 2003a).

A major challenge is for the project to strike a balance between two ends: avoiding broken builds on the development branch (which disrupts the work of many developers downloading and using it), and limiting to a

reasonable level the precommit effort required by the individual developer. It appears that there is indeed 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).

The committer obtains software for precommit testing by checking out a copy of the most recent version of the project's development version, and adding the proposed change. The hardware used is (normally) an Intel-based PC at the developer's home or work.

Pragmatic interpretation seems to be particularly called for with respect to the number of different platforms on which a developer should verify the build. Due to platform differences, a build may succeed on one and fail on another. FreeBSD supports a wide range of processor platforms, four of which (i386, sparc64, PC98, alpha) are so-called tier 1 architectures; that is, architectures that the project is fully committed to support. Rule number 10 continues: "If you have a change which also may break another architecture, be sure and test on all supported architectures" (FreeBSD 2003a).

The project has made a cluster of central build machines available, including all tier 1 architectures, to which sources can be uploaded and subjected to a trial build prior to commit. However, uploading to and building on remote machines is tedious, which can be seen as a cost of the delegated approach to integration. There are frequent complaints that committers omit this step.

Developers' learning about the system as a whole, and their acquisition of debugging skills, may be a result of the delegated approach to building, as opposed to the traditional approach of creating a team dedicated to building and integration.

Correcting a broken build can be highly challenging. This is partly because the activity is integration-related: a build failure may be due to dependencies with files not directly involved in the change and so possibly outside of the area of the developer's primary expertise. Debugging an operating system kernel is particularly difficult, because when running it has control of the machine.

Typical comments were: "Debugging build failures . . . has forced me to learn skills of analysis, makefile construction . . . etc. that I would never be exposed to otherwise," and, "I have improved my knowledge about other parts of the system."

In response to the statement "I have improved my technical skills by debugging build failures," 43 percent chose "Yes, significantly" and 29 percent "Yes, somewhat."

It is difficult to assess the actual technical competencies of a development team, and even more difficult to judge whether they are enhanced by FreeBSD's approach to integration. A number of developers indicated that they were competent before joining FreeBSD. One reported, "The way you get granted commit privileges is by first making enough code contributions or bug fixes that everyone agrees you should be given direct write access to the source tree. . . . By and large, most of the committers are better programmers than people I interview and hire in Silicon Valley."

Development Release (Commit)

I can develop/commit under my own authority, and possibly be overridden by a general consensus (although this is rare).

—FreeBSD developer

Development release of a change consists of checking it in to the repository by the committer, upon which it becomes available to the other committers, as well as anyone else who downloads the most recent version of the "trunk." It is up to the committer to decide when a change has matured to an appropriate level, and there is no requirement that he or she provide proof that the change has been submitted to review.

The repository is revision controlled by the CVS tool. A single command suffices for uploading the change from the developer's private machine to the central repository. Revision control also enables a change to be backed out.

There is a well-defined process for the case in which, upon a commit, it turns out that an appropriate consensus had not been reached in advance. "Any disputed change must be backed out . . . if requested by a maintainer. . . . This may be hard to swallow in times of conflict. . . . If the change turns out to be the best after all, it can easily be brought back" (FreeBSD 2003a).

Moreover, a consensus between committers working in some area can be overridden for security reasons: "Security related changes may override a maintainer's wishes at the Security Officer's discretion."

McConnell recommends that projects using daily builds create a *holding area*; that is, a copy of the development version through which all changes must pass on their way to the (proper) development version, to filter away changes not properly tested. The purpose is to preserve the development

version in a sound state, because developers rely on it for testing their own code (McConnell 1996, 409). FreeBSD has no such filtering of the stream of changes flowing into the trunk, and so depends strongly on the committee's willingness and ability to release only reasonably tested changes.

Parallel Debugging

> We . . . don't have a formal test phase. Testing tends to be done in the "real world." This sounds weird, but it seems to work out okay.
>
> —FreeBSD Developer

Upon commit to the trunk, a change is tested; in a sense, this is consistent with Raymond's notion of parallel debugging (Raymond 2001). The trunk is frequently downloaded—for example, 25 percent of the committers said they had downloaded and built the development version on five or more days in the preceding week. There may be in principle two different reasons for FreeBSD developers to be working with the most recent changes, other than for the direct purpose of testing them. First, for precommit testing to be useful, they must use the most recent version. Second, to benefit from the newest features and bugfixes, advanced users may wish to use the most recent version for purposes not related to FreeBSD development at all.

The first test of a newly committed change is the build test. Regardless of the rules set up to prevent broken builds on the trunk, the project's members are painfully aware that there is a risk for this to happen. Broken builds will normally be detected by developers, but to ensure detection, the project runs automated builds twice a day on the four tier 1 architectures—so-called "Tinderbox builds," the result of which are shown on a Web page (http://www.freebsd.org/~des).

FreeBSD has no organized effort for systematic testing, such as with predefined testcases. There is also no regression test to which all new versions of the trunk are subjected. McConnell suggests that checking the daily build should include a "smoke test" that should be "thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly" (McConnell 1996).

It should be noted, though, that there is an element of a smoke test involved in booting the newly built operating system, and launching standard programs such as editors and compilers, as carried out on a regular basis by FreeBSD's committers.

The community's use of the development version—once it is in a working state—produces a significant amount of feedback: some respondents indicated that they receive a constant flow of problem reports; nearly half the respondents said that, within the last month, someone else had reported a problem related to "their" code. (Also there is feedback in terms of actual bugfixes, as mentioned in the earlier section "Division of Organization, Division of Work.") Thus there is indication that keeping the build healthy is valuable for debugging, in addition to the importance for precommit testing as such. However, there is also indication that the feedback generated by parallel debugging mostly pinpoints simple errors: "In actuality, the bug reports we've gotten from people have been of limited use. The problem is that obvious problems are quickly fixed, usually before anyone else notices them, and the subtle problems are too 'unusual' for other developers to diagnose." (FreeBSD project manager)

Production Release

> We were spinning our thumbs. . . . It was a really boring month.
> — FreeBSD developer, referring to the month preceding the 4.0 release

Production release is the final step in integrating all changed work. A production release is a snapshot of a branch in the repository, at a point where the project considers it to be of sufficiently high quality, following a period of so-called "stabilization". This section discusses the process leading to major production releases (such as version 5.0) that are released at intervals of 18 months or more. In addition, the project creates minor production releases (5.1) at intervals of three to four months. Radical changes such as kernel-enabled SMP are released only as part of major production releases.

During stabilization prior to a major production release, the trunk is subjected to community testing in the same manner as during ordinary development. The difference is that new commits are restricted: only bugfixes are allowed. The committers retain their write access, but the release engineering team is vested with the authority to reject all changes considered to be not bugfixes of existing features, and the team's approval is needed prior to commit.

Stabilization is also a more controlled phase in the sense that a schedule is published: the code-freeze start date tells committers the latest date at which they may commit new features, and the un-freeze date when they can resume new development on the trunk. The stabilization period for

5.0 lasted for two months, the first month being less strict with new features being accepted on a case-by-case basis. Indeed, while change initialization is somewhat anarchistic, work during the final steps towards production release is managed rather tightly. For example, the release engineering team defined a set of targets for the release, involving for example the performance of the new SMP feature (FreeBSD Release Engineering Team 2003).

The ability to create production releases merely by means of the process of stabilization is a major advantage of FreeBSD's approach to integration. The process is relatively painless—there is no need for a separate phase dedicated to the integration of distinct parts or branches, because the software has already been assembled and is in a working state.

A major disadvantage of "release by stabilization" is the halting of new development. When the potential represented by developers with an "itch" to write new features is not used, they may even feel discouraged. To accommodate, the release engineering team may terminate stabilization prematurely. This implies branching off at an earlier point of time a new production branch, where the stabilization effort is insulated from new development on the trunk. Then commits of new features (to the trunk) do not risk being released to production prematurely, or introducing errors that disrupt stabilization. Indeed, production release 5.0 was branched away from the trunk before it was considered stable. However, there is a trade-off, because splitting up into branches has a cost. First, bugfixes found during stabilization must be merged to the development branch. Second, and more importantly, the project wants everybody to focus on making an upcoming production release as stable as possible. Splitting up into branches is splitting the community's debugging effort, which is the crucial shared resource, rather than the trunk as such.

Conclusion

> Respect other committers. . . . Being able to work together long-term is this project's greatest asset, one far more important than any set of changes to the code.
>
> —FreeBSD 2003a, the Committer Guide's description of rule 1

FreeBSD accomplishes coordination across a project that is geographically widely distributed. FreeBSD's incremental and decentralized approach to integration may be a key factor underlying this achievement: it may enhance developer motivation and enable a relatively painless process for creating production releases by maturing the project's development

version. The project avoids allocation of scarce developer resources to dedicated build or integration teams, with the perhaps not-so-interesting task of integrating other people's changes or drifted-apart branches.

The project's development branch is something of a melting pot. There is no coffee machine at which FreeBSD's developers can meet, but the development branch is the place where work output becomes visible and gets integrated, and where the key project rule—don't break the build—is applied and redefined.

A disadvantage of FreeBSD's approach to integration is the risk of overloading the trunk with interdependent changes, when too many changes are committed too early. In a sense there is a limited capacity to the trunk, and one that can not be overcome simply by branching, since the underlying scarce resource is the community effort of parallel debugging.

FreeBSD's decentralized approach seems to contradict hypotheses that hierarchy is a precondition to success in open source development. For example, Raymond stressed the need for a strong project leader, albeit one who treats contributors with respect (Raymond 2001). Healy and Schussman studied a number of apparently unsuccessful open source projects, and asserted that the importance of hierarchical organization is systematically underplayed in analyses of open source (Healy and Schussman 2003). The author of this chapter would stress the need for mature processes rather than hierarchy. FreeBSD is a promising example of a decentralized organization held together by a project culture of discussion and reinterpretation of rules and guidelines.