A Case Study of a Corporate Open Source Development Model

Vijay K. Gurbani
Bell Laboratories
Lucent Technologies, Inc.
Naperville, IL 60566 USA
+1 630 224 0216
vkg@lucent.com

Anita Garvert
Bell Laboratories
Lucent Technologies, Inc.
Naperville, IL 60566 USA
+1 630 713 1567
agarvert@lucent.com

James D. Herbsleb
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412 268 8933
jdh@cs.cmu.edu

ABSTRACT
Open source practices and tools have proven to be highly effective for overcoming the many problems of geographically distributed software development. We know relatively little, however, about the range of settings in which they work. In particular, can corporations use the open source development model effectively for software projects inside the corporate domain? Or are these tools and practices incompatible with development environments, management practices, and market-driven schedule and feature decisions typical of a commercial software house? We present a case study of open source software development methodology adopted by a significant commercial software project in the telecommunications domain. We extract a number of lessons learned from the experience, and identify open research questions.

Categories and Subject Descriptors
K.6.3 [**Computing Milieux**]: Software Management – Software development, Software maintenance, Software process.

General Terms
Management, Documentation, Design, Experimentation, Human Factors.

Keywords
Open Source, Software Development, Session Initiation Protocol, Architecture

1. INTRODUCTION
Open source practices and tools have proven potential to overcome many of the well-known difficulties of geographically-distributed software development [10], and to allow widely distributed users of software to add features and functionality they want with a minimum of conflict and management overhead [12]. Some reports have appeared in the literature describing experiences with open source tools in an industry setting [7], and in fact there has been a workshop focused specifically on open source in an industry context [2].

It is not immediately obvious, however, that open source tools and practices are a good fit to a corporate setting. To be sure, open source software is used extensively in the industry, and the recent acceptance of Linux and the Apache project are excellent examples of this phenomenon. However, what needs further study is whether the industry as a whole can benefit from adopting the *methodology* of the open source software development. Is the open source development methodology conducive to the manner in which corporations develop their software, or are there only certain industrial projects that are amenable to the open source development methodology? Dinkelacker et al. [6] discuss Progressive Open Source as a set of tools and techniques for a corporation to host multiple open source projects within a corporation and between third parties. Our work adds to the state of knowledge by providing detailed analysis combed from interviews of multiple developers and quantitative analysis of data pertaining to a corporate open source project where multiple organizations contributed synergistically to further and use a common asset. Our lessons learned consist of how to make corporate open source successful in the face of multiple organizations using different internal development tools and techniques.

In this paper, we report on the continuing case study on a project we have been involved in that uses open source tools and practices in the development of a commercial telecommunication software [6]. The project is an Internet telephony server originally built by one of the authors (vkg), and later administered as an open source project inside Lucent Technologies in order to speed development and quickly add functionality desired by different project groups who wanted to make use of it in their product lines. We describe the effort's experiences over a four-year period and present a number of lessons learned about how to make such projects succeed.

The rest of this paper is structured as follows: in section 2, we compile a set of characteristics that while common to all open source projects, may be exhibited differently under a commercial environment. Section 3 describes the software project we used in the case study. In section 4, we describe the initial development of the software and its use inside the company, the open source style setup and quantitative results from analysis of archival data. Section 5 follows by presenting the results of extensive interviews conducted with the developers and users of the software. Section

ICSE'06, May 20-28, 2006, Shanghai, China.

6 presents lessons learned from this case study, and Section 7 looks at some open issues and concludes the paper.

2. OPEN SOURCE PROJECT CHARACTERISTICS

While there is, of course no definitive set of characteristics that all open source project necessarily share beyond permitting legal and pragmatic access to source code, there are many practices which are common across a large sample of open source projects (e.g., [8]). Some examples of how these practices seem potentially incompatible with commercial development are the following:

2.1 Requirements

Commercial projects typically devote considerable effort to gathering and analyzing requirements, in a process that often involves several disciplines including marketing, product management, and software engineering. Open source projects, on the other hand, rely for the most part on users who are also developers to build the features they need and to fix bugs. Other users generally have to rely on mailing lists and change requests [14] to communicate feature requests to developers, who then may or may not address them, depending on their interests and the perceived importance of the requests. In commercial environments, management, often operating through a change control board, makes decisions about changes based on business needs.

2.2 Work Assignments

In firms, developers are generally assigned by management to projects and development tasks. There is usually an effort to match tasks with developers' skills, and often an attempt to match their interests if possible, but developers' choices are generally rather limited. In open source, developers typically choose what they want to work on. Generally, they begin building something they themselves need as users of the software. Those who continue to contribute tend to begin taking on jobs because of their perceived importance to the overall project [15].

2.3 Software architecture

It has often been argued that open source projects require a more modular architecture than commercial projects, and there is now some evidence that this is the case [11]. In fact, the architecture of the Netscape browser became much more modular after it was released as open source [11]. More generally, it is widely recognized that the structure of the organization is a critical determinant of the structure of the code [4, 9]. It is not clear how well architectures designed for a commercial environment will support the sort of collaboration that open source practices must support.

2.4 Tool compatibility

Most open source projects exist independently, or coexist on hosting services other projects that have all decided to adopt the same set of tools. In commercial environments, however, the situation is generally more complicated. There is often a wider range of tools used, and it is not clear how to support open source practices in heterogeneous environments.

2.5 Software processes

Many commercial environments have various levels of defined processes, often accompanied by stage gate systems where projects are evaluated at various critical points along the development path. These process are generally seen as critical to assuring software quality. Open source, on the other hand, generally has very little in the way of formal process, and instead insures quality through the "walled server" [8], placing control over what goes into releases in the hands of a "benevolent dictator", or small group of proven technical experts. These two approaches may prove to be incompatible.

2.6 Incentive structure

Commercial development is profit-driven, while open source is driven by a complex set of motives, including the desire to learn new skills, the desire to create features one needs, philosophical beliefs about contributing to the general welfare, for enjoyment of the freedom to build what one wants, and sometimes as a political statement about commercial business practices. The practices that make the very different open source and commercial practices succeed may rest in complicated ways on the developers' differing motivations.

3. THE SOFTWARE: A TELECOMMUNICATIONS SIGNALING SERVER

The specific software we use in our case study is a telecommunication-signaling server. The server is a faithful implementation of the Internet Engineering Task Force (IETF) Session Initiation Protocol (SIP [13]). SIP is an Internet telephony signaling protocol that establishes, maintains, and tears down sessions across the Internet. SIP is a text-based protocol that operates on the notion of a transaction. A transaction is a request issued by a client followed by the receipt of one or more responses (from that viewpoint, SIP is like any reply-response protocol like HTTP, SMTP, or FTP).

By the early 2000, the telecommunications industry was starting to coalesce around a cellular telecommunications architecture called the 3rd Generation Internet Multimedia Subsystem (3G IMS). IMS imposed additional requirements on SIP beyond what the IETF standards dictated.

A SIP system has many entities: proxy servers help end points (called user agents) rendezvous with each other; registrars exist to register user agents so they can be found easily. Integral to a SIP entity is the notion of a transaction. Thus, in a typical SIP software stack, a transaction manager (defined above) that is scalable and provides the many services that the standard requires is essential. Residing on top of the transaction manager would be specific SIP entities called transaction users: proxies, user agent servers, user agent clients, and registrars, are all transaction users.

The source code was written in the C programming language and Concurrent Version System (CVS) was used for source code control and versioning. The code executed on the Solaris and Linux operating systems. The original version of the software was written as a server, however, as we will discuss later, the code was re-factored to create a general purpose SIP library, which currently hosts the server.

What we have described so far suffices as a technical context for the rest of the paper; however, interested readers can refer to Rosenberg *et al*. [13] for more information on the protocol and detailed workings of it.

4. THE OPEN SOURCE EXPERIENCE

In this section, we will give an overview of how the code and the development process evolved, in order to clarify the experience base from which our lessons learned were derived.

The timeline of the project is characterized by three distinct phases. The first phase – Development Phase – spanned the time between April 2000 to November 2001. The next phase – Ad-hoc partnering and user-initiated change requests – followed thereafter and lasted until April 2004. Between May 2004 and May 2005, the software entered its last, but most crucial phase, the Open source development phase. Here, the project benefited proactively from the varied experience of many people from different backgrounds and projects working simultaneously on the software.

4.1 Phase I: Initial Development

The initial work on developing the software was conducted by one of the co-authors of this paper (vkg) at Lucent Technologies by closely following the work progressing in the IETF SIP working group. At this point in time, the development was mainly an effort lead by the author of the code and an additional developer. The author was in close touch with the work progressing in the IETF by contributing to and deriving a benefit from the discussions about the protocol. Once the code had enough features in it, it was taken to a number of interoperability events to ensure its compliance to the protocol as well as other implementations.

4.2 Phase II: Ad hoc partnering

As the code grew stable and achieved feature parity against the functionality specified in the protocol, the author started to distribute the binary to a wider audience inside Lucent Technologies[1]. An internal website advertised new binary releases of the server for others within the company to download and experiment with. The maturity of the server implementation coincided with the burgeoning acceptance of SIP as a protocol of choice in the telecommunications domain (1999-2001).

As internal interest in the server grew, the capabilities of the server were demonstrated by closely partnering in an opportunistic way, with select groups. For instance, the author extended the programmability of the server by providing callbacks when certain SIP events occurred in the server (arrival of a SIP request or a response). Using this programmability, the server was tied to a collaboration- and presence-related framework that was the focus of research in other groups within Lucent Technologies [3]. Partnering of this type benefited many research projects within the company. At this time, such partnering was mainly limited to integration with existing frameworks and jointly staging demonstrations.

4.3 Phase II: User-initiated change requests

As the server matured, it moved beyond a research-only project and was being productized as part of a standard Lucent Technologies. offer. Initially, even though select groups within the company had access to the source code, there weren't any contributions from them beyond the users reporting their experience to the author. Most internal users were simply downloading the compiled version of the server and using it for their work. Expanding the class of users in this way created a positive feedback loop in which the original code author implemented new features these users needed. The author encouraged other users within the company to use the software and report feedback and wishes for new features. This communication was conducted in an ad-hoc fashion, primarily over email and an updated web page. Requests for new features were ordered according to the business needs of the group productizing the server and the research interests of the author (often time, luckily, these coincided).

As SIP continued to gain industry adherents and as the general field of Internet telephony became more important, the server was viewed as a critical resource by many groups. Certainly, having access to the source code of a standard compliant server was extremely advantageous, more so since the standards were in a state of flux as SIP further evolved to touch other aspects of Internet services such as instant messaging and presence. By 2003, the server's source code was studied extensively by other groups within Lucent Technologies. Requests started to arrive on evolving the server to serve as a framework for many SIP-related groups within the company.

4.4 Phase III: Establishing open source development project

At about the same time that requests for product-specific changes began to accelerate, others within the company started to contribute code and ideas back to the author. The stage was set to enter the traditional open source development model, albeit within an industrial setting.

The author of the original code (vkg) assumed the role of a "benevolent dictator" controlling the code base to ensure that the contributions coming in and features that other groups were proposing to build into the code matched the architectural principles of the software.

The author re-factored major portions of the server code to create a transaction library that could be used by any project within the company (since all SIP entities need a transaction manager). Working in close co-operation with two other projects, APIs and interfaces between the transaction manager and the transaction users were defined for information to flow between the manager and the transaction user. Re-factoring the software in this manner was very successful and enabled rapid creation of user agents [1] that executed on top of the transaction manager. Since the user agents were using the services of a transaction manager that was already implemented and tested, the programmers of these user agents could concentrate on the task of implementing the specific behavior of the user agent itself instead of worrying about the details of handling SIP transactions and other protocol-related

[1]While the server was not made available for download outside the company, for the sake of interoperability, it was hosted on a machine accessible to the public. Implementers outside Lucent Technologies can use the server to benchmark their implementation even today.

minutiae. The re-factoring has been so successful that the initial server now runs on top of the transaction manager as well. Other groups that want specific transaction users can build them over the transaction manager by simply adhering to the APIs and interfaces.

Once re-factored, the code base evolved as follows. The initial release of the SIP transaction library was developed in a CVS archive (CVS and its derivatives remains the preferred source code control mechanism in the open source community; the author of the original code chose it since he was most familiar with it). As the code moved into Phase III, a CVS branch was created to allow the additional developers to assist in development and product evolution. Contributions at this time consisted of manageable extensions to the base product as well as the APIs and interfaces built around it. Two projects were taking delivery from the main CVS trunk and one project took delivery from the CVS branch.

This structure was effective initially, since the modifications were relatively independent and the number of developers was limited. However, in the later stages of Phase III, the nature of the features being put into the code, the maturity of the product, the number of contributors, and the experience of the contributors, all lead to the need for a different model to evolve the code base.

It is very important to point out that in corporate software development, each project has an affinity for a certain set of tools (see Section 2.4). The set of contributors now adding features to the code were accustomed to their organization's development environment. Thus, some organizations took a copy of the CVS archive and replicated it in their local software environment to closely model what the developers in that organization were accustomed to. Of course, since none of the organizations used CVS for source control, the source files were put under the source code control system that the particular organization was well versed with. At the same time, the original CVS load line and the initial CVS branch were still being supported. It was at this time that the concept of an independent and common source code repository was born. An open source group was consequently formally formed to co-ordinate the independent and common source code repository. This group, which we call the Common SIP Stack (CSS) group in the rest of this paper, is being lead by one of the co-authors of this paper (agarvert).

The goal of the CSS group is twofold: one, maintain an independent and common source code repository such that all projects within the company take their deliverables from the CSS group. This is not an easy task. Not only must the CSS group maintain such a repository, but it must also be the final arbiter of what feature goes into the code and ensure that any feature added does not break existing functionality pertinent to a different project. In addition, the CSS group must also have a vision of evolving the code and deciding which of the many SIP extensions should be supported in a timely manner and in such as way as to not adversely impact the performance of the code. The synergy that resulted in the complexity lead to the replication of another well known phenomenon in the open source community: the role of a "trusted lieutenant." Management identified strategic personnel in different groups and assigned them to manage key portions of the code while working closely with the author of the original code.

The second goal of the CSS is to evangelize the technology and the implementation by creating awareness of the resource within the company. To this extent, a SIP Center of Excellence (COE) has been established that acts as a central web site from which other projects within the company can get information on the shared asset and instructions on how to download, compile, and execute the source code. The COE acts as a one-stop shop for all SIP needs that any project within the company may need.

4.5 Quantitative results

We now present results from several quantitative analyses of the archival data from the project. This data is correlated with the evolution of the project through the three-phased timeline outlined previously.

4.5.1 The Size of the Development Community

During Phase I, the author was the sole contributor to the code. Towards the latter end of the development phase, another developer was provided to aid the author in productizing the server. Phase II did not witness any marked increase in the size of the development community. The author was still the sole developer, with 1-2 developers rotated in and out of the project depending on other needs and priorities.

The size of the development community increased exponentially in Phase III. As more projects got involved in the software and started to contribute substantial portions of the code, the size of the development community increased to a high of 30 developers working concurrently. In reality, this number tends to fluctuate because the developers belong to different organizations with their own management chain; thus depending on the needs of a particular organization, developers may be put into or taken out of the project. However, the main development community consists of about 20 developers, including the original author of the code. Each of these 20 developers is responsible for certain subsets of the system; some of them have added substantial features to the software core and are thus responsible for the upkeep of those features, while still others own the core of the software and are occupied in almost all aspects of the evolution of the software, including providing guidance on how new features are best added into the system.

4.5.2 Normalized Lines of Code

We define normalized lines of code as the subset of the source code tree that is required to compile the software base completely. Specifically, this count does not include all the support software that was built in parallel to test the functionality of the server. Due to the complexity of the software, the test scripts and test programs themselves were about 60% of the normalized lines of code.

Table I demonstrates the growth of the normalized lines of code from the inception of the server to the current state. The last column indicates one of the three phases of the software enunciated previously. The column with the "Delta" heading contains the count of lined added to (+) or subtracted from (-) the number of normalized lines of code from the previous release.

The largest "Delta" value occurs between April 2000 and Nov 1, 2001, during the formative stage of the software. In the approximately year and a half that the time period represents, the software was being actively developed, taken to the

interoperability events. By the latter date, it was released as the first productized version. The delta values after Nov 1, 2001 follow a predictable pattern: they inflate when major features are added to the software (Jan 6, 2005; Dec 2002, etc.) and in some cases, they shrink as dead pieces of code are taken out or optimized away (Aug 2004; Jan 21, 2005).

4.5.3 Software Release Frequency

Figure 1 depicts the software release frequency per year. It is instructive to note that the release frequency starts to climb rather steadily once the software reaches Phase III (open source development phase) during the 2004-2005 time period. At this time, there are many more developers doing active and parallel development to the software, thus necessitating in an increased release cycle. Until Phase III, it is hard to characterize the frequency of releases per year. This is primarily due to the fact that until Phase III, the software was in its formative stages, thus the release schedule was driven by how quickly the original author (vkg) could add new features and fix bugs working alone.

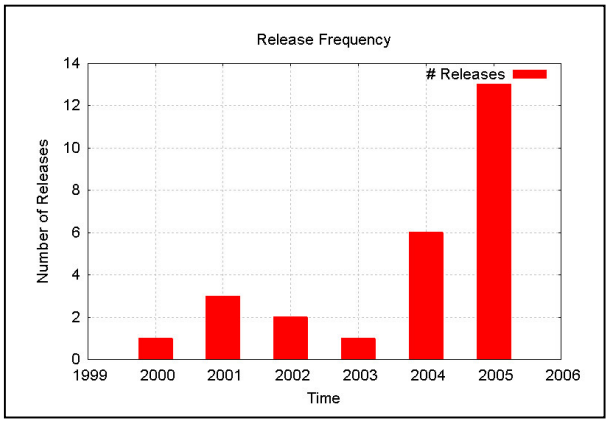

Figure 1. Software Release Frequency.

4.5.4 Number of Software Downloads

Figure 2 plots the number of unique software downloads across the three phases of the software. In Phases I and II, the original author (vkg) of the software implemented a licensing mechanism simply to track the number of internal uses of the software. Each time a new project or person wanted to use the software, a license key was provided. In Phase III, as the software became more open source, a web site for downloads was provided that keeps a log of the number of downloads.

As can be observed in Figure 2, Phase II was the most download period of the software. A total of 87 unique licenses were requested during that phase. Phase III has witnessed about half the number of downloads (40) when compared to Phase II. This can be attributed to the fact that the software was used on a more project-wide basis as opposed to being downloaded and used by individuals. There are about 20 projects throughout the company that are using the software.

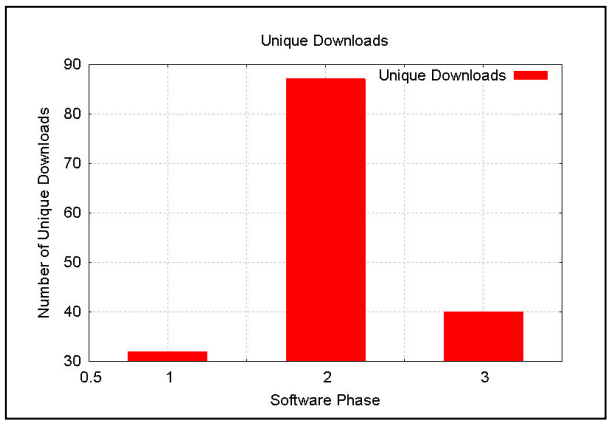

Figure 2. Number of Software Downloads.

Table 1: Normalized Lines of Code (LoC).

Date	Normalized LoC	Delta	Phase
April 20, 2000	6163	-	I
Nov 1, 2001	21,239	+15,076	I
Nov 9, 2001	21,397	+158	I
Nov 29, 2001	24,497	+3,100	I
May 22, 2002	26,335	+1,838	II
Dec 20, 2002	32,089	+5,754	II
Aug 12, 2003	33,907	+1,818	II
Apr 27, 2004	38,666	+4,759	II
Aug 26,2004	35,849	-2,817	III
Sept 16, 2004	36,140	+291	III
Sept 24, 2004	34,669	-1,471	III
Oct 1, 2004	36,347	+1,678	III
Oct 6, 2004	34,669	-1,678	III
Dec 10, 2004	36,398	+1,729	III
Jan 6, 2005	43,401	+7,003	III
Jan 20, 2005	45,709	+2,308	III
Jan 21, 2005	44,216	-1,493	III
Feb 4, 2005	44,416	+200	III
Feb 15, 2005	44,413	-3	III
Mar 9, 2005	47,078	+2,665	III
Mar 21, 2005	47,212	+134	III
Mar 30, 2005	47,358	+146	III
Apr 18, 2005	47,457	+99	III
Apr 28, 2005	47,649	+192	III
May 4, 2005	47,547	-102	III
May 31, 2005	47,853	+306	III
June 13, 2005	48,145	+292	III

5. RESULTS FROM INTERVIEWS

In order to understand the internal open source development experience from all relevant perspectives, we interviewed 14 developers who were contributors or users of the SIP server software. The interviews were typically one hour long, and were tape recorded with the interviewees' permission. They were semi-structured, meaning that the interviewer had a specific set of topics to cover, but the questions were not fully scripted. In order to encourage the interviewees to be as candid as possible, the co-author not affiliated with the company (jdh) conducted all the interviews, and the notes and tapes were not shared – the results were shared only in aggregated form to protect confidentiality.

The interview results were analyzed using typical qualitative techniques. During the interviews themselves, the interviewer frequently summarized the interviewee's comments in order to check his interpretations and eliminate any misunderstandings. He then went through all of the detailed interview notes to identify themes, and compiled all comments related to each theme, to see if all comments were consistent, and to make sure that all views relevant to a theme were included. Any remaining uncertainty was resolved with the help of the insider information of the first two authors.

We organize the interview results around four themes: the initial decision to build an open source resource, adjustments made to the organizational process to accommodate the open source development, how the contributions were managed, adjustments needed to coordination open source and product development activities, publicity and communication, and additional benefits.

5.1 Choosing to build an open source resource

The decision to build an open source resource has basically two components. First there is the basic build vs. buy decision, and second, it must be decided whether an open source cross-business-unit strategy is better or whether the resource should be built in a traditional way within a product group. If there are commercial versions available, when does it make sense to simply purchase one of those versus building your own? As the results of our interviews indicate, the choice in this case was generally easy, since the internal system had already been largely developed by the time product groups began to see the market demand for the SIP stack. Since the internal version already existed, and was viewed as a high-quality solution, it was nearly universally preferred inside the company. Interviewees cited the flexibility of having your own version that you can modify at will, the timeliness of changes that you can make yourself without being concerned with a vendor's release cycle. One interviewee also cited the political goodwill accrued by the group that developed the internal product. In the end, the decision whether to manage the resource as open source depends on a number of characteristics that we will discuss in the lessons learned section.

Building a cross-business-unit resource also has implications for supply chain and sourcing strategies. The departments responsible for purchasing and for technology evolution need to be aware of, and to evaluate internal resources just as they do external, and should participate in decisions about what kinds of resources to create. Our interviews revealed an initial disconnect between the resource builders and supply chain managers, since developing resources in this way was not typical in this (or most other) companies. Additionally, there are challenges in comparing internally developed resources with commercially-available ones, since it is difficult to determine the actual cost of the internal software, or to measure the benefits, such as modifiability, that come from owning the code.

5.2 Organizational process adjustments

In order to create an effective relationship between the open source effort and the various product efforts, several kinds of adjustments in software process were made. As in many development organizations, the software process varied considerably across product groups. The CSS group, operating relatively independently of these product groups, had to design or choose a process. Since many developers contributing to the server were based in a variety of product groups, their expectations and habits were often misaligned with the newly formed CSS group.

The process adopted by the CSS group was relatively stringent, especially in the area of inspections, including mandatory review, tracking of all comments in reviews, and re-review of modified code. Particularly for developers from product groups with more lightweight processes, following this process for changes to the server code was unattractive, and seemed to them unnecessary. After all, they were generally just building functionality that they needed, and did not see working on the server code as all that different from working on code for their own products.

It proved difficult organizationally to make this process work. The benevolent dictator did not have sufficient time available to personally review all code contributions, so a pool of knowledgeable contributors was established to assist with reviews. But since members of this pool had their primary responsibilities in product groups, they were reluctant to set aside time for the reviews. At least one interviewee thought that participation in these reviews should be made mandatory.

Build environments were sometimes different across sites, and caused some problems. According to one interviewee, one site had problems with particular template libraries that proved fairly difficult to resolve.

5.3 Managing the contributions

The evolution of the code base was outlined in Section 4.4. To summarize, the code base evolved from being maintained in a single CVS main trunk to an additional CVS branch as well as replicated instances of the CVS branch in at least two different source code control systems used by other projects.

Each project was adding to the code, as was the benevolent dictator. The benevolent dictator would periodically provide drops of his code base to the CSS group. Contributions of the other groups would then be merged with the drop provided by the benevolent dictator. Each unit was adding to the code, some in a major effort while others in incremental minor efforts. What's more, this addition was often done in separate repositories. Thus, a "buy back" process was defined as the movement of code from one repository to another. Contributors in non-CVS repositories often attempted to tag changes that needed to be bought back to make the job of merging easier. Periodically, the benevolent dictator would inspect these changes and "buy back" those that were sufficiently general into the main CVS archive. Similarly, other organizations would take deliveries from the CVS main

archive (for content that others had contributed) and integrate the release into their private repositories.

The "buy back" step, in principle, is a valid step in an open source model. However, the fact that different repositories and change management tools were used by the main actors resulted in this task being more difficult than it should be. The problem is further exacerbated if the work across the repositories is not synchronized on a constant basis. If the "buy back" step is delayed, the code across the repositories starts to diverge, thus making this step a non-trivial rework of features.

5.4 Coordinating open source and product group development

Contributions to the server came from many sources, and were made by many people, as discussed above. One of the most basic problems that many interviewees experienced was that developers were unaccustomed to thinking and designing solutions that were more general than their own product line. They typically did not make changes to the SIP server in a way that would support all users, but rather worked in their customary way, unconcerned about building in dependencies that limited the generality of their work.

The tendency to work in a limited, product-specific way was also described in many cases as a response to management pressure to get the changes in so they could have a release of their product ready on time. Interviewees mentioned the difficulty of the conflicting pressures between the product release cycle, where time-to-market was often critical for success of a product release, and developing functionality with full generality, which is critical to the success of a common resource.

The interviewees mentioned several important different types of changes: 1) those that were very general and should go into all releases, 2) those that were specific to a platform and should go into all releases targeted at that platform, and 3) those that were specific to a given product. It was important to keep these three types of changes separate, but it was often difficult for developers to understand which sort of change they were making. Complicating all of this was the fact that the code was portable across four operating systems: Solaris, Linux, pSOS (a real-time operating system) and Windows. Some developers were well versed with only a subset of the operating systems, thus writing code that could be guaranteed to be portable across the other operating systems proved to be quite challenging.

In some cases, for example when they used functions available only in the code for their product, it was easy to determine that they should use "#ifdef" or similar instructions to isolate the change. But for many changes, when developers did not know in any detail the assumptions that might be embedded in the way other products used SIP functions or data, it seemed risky to add the changes directly to the code base. One interviewee commented that developers "should take the time to understand if it will [affect other applications], but usually they don't." For this reason, very large amounts of code were "#ifdef'ed," often unnecessarily. If then, the benevolent dictator "bought back" the new functions, the "#ifdefs" were no longer valid. Eventually, several interviewees reported that the number of these compiler instructions in the code made it very hard to read.

Additional problems were encountered in coordinating the timing of product-specific and transaction library-specific releases. As products near release, projects impose a code freeze after which changes are tightly controlled, and no new functionality is added. The transaction library, on the other hand, driven by the benevolent dictator's schedule, has its own release cycle, with fixes and new functionality being added without any particular regard to the product release cycle. The issue arose when the product developers wanted to include all fixes to the server, but would have liked to exclude new functionality. This is difficult to manage. As one developer said, "every project has to be in charge of its own base; [you] can't be a slave to someone else's base."

One additional complication was the product groups often had to deliver custom versions of products specific to a single customer, since they required a small change that would not make sense for other customers. This meant that they also had to maintain multiple versions of the SIP server, and merge bug fixes as required across these versions. This situation of maintaining customer-specific releases was complicated and messy in any case, and maintaining multiple versions of an unfamiliar technology such as the SIP server made the situation worse.

5.5 Publicity and communication

Several interviewees mentioned that when they found they needed a SIP stack for their product, they stumbled on the internal resource accidentally as they were beginning to research SIP stacks available on the market. The strategic technology evolution department also found it difficult at first to get sufficient information to evaluate the internal SIP stack, although they quickly moved to correct the situation.

In addition to the basic problem of raising awareness across the company that the technology exists, there is the problem of bringing developers up to speed on what it does and how it was designed. Since the internal open source strategy relies on users of the technology, often spread widely across the company, to improve and extend it, they need ways to come up to speed. The benevolent dictator delivered talks around the company describing it in as much technical detail as he could given time constraints, and developers could review the code, some documentation, and could read the standards to which it conformed. Nevertheless, several reported that they thought more resources were necessary to help them understand this particular implementation.

Another significant coordination problem was knowing what kind of work was going on for the server. There were many cases where developers in different product groups duplicated each other's efforts because they were unaware of each other's work. Several developers suggested that either some sort of process be put in place that required developers to register before beginning work on the server, or a web site or newsletter be established that would distribute updates information about the various SIP-related development efforts.

The SIP COE presented in Section 4.4 mitigates some of the challenges presented in this section.

5.6 Additional Benefits

In addition to creating, evolving, and maintaining a common resource for the company, the internal open source SIP project had other benefits that seem unique to this approach.

Compared to alternative ways of creating SIP capabilities, this approach appeared to aid in disseminating knowledge of SIP technology through the product groups. Had all SIP development been carried out by a single group that had full and exclusive ownership of the server, technical staff from product groups would not have been able to acquire the level of "hands-on" experience that the open source approach gave them. If, on the other hand, there had been a fully decentralized approach where each product group developed its own server, it would have been difficult for these groups to learn from each other and the benevolent dictator, since there versions would differ substantially and there would be little need to interact.

Another advantage was reported by individuals, who saw this participation as an opportunity to develop professionally valuable technical skills. One developer, in particular, began contributing in his spare time, evenings and weekends, much as "hobbyist" developers in other open source projects do. In order to acquire new skills – and because he enjoyed the challenge, he began, completely on his own, to study the code and contribute. Eventually, his contributions were recognized, and he was officially assigned to do substantial SIP development, which was an assignment he welcomed. This is a potentially important way for the company to groom highly motivated staff to handle new technologies.

The "many eyeballs" effect of open source development is well known (i.e., the code benefits from being scrutinized by a wider audience with different interests and capabilities [16]). This effect exhibited itself in this specific project in many interesting ways:

- By studying the code, the performance experts suggested a list of changes that would optimize the implementation [5];
- API experts suggested a layer of API that would lead to a more programmable framework;
- Others who were working on a 3G IMS project suggested (and contributed) modifications that made the code compliant to that architecture;
- Others still ported the code to other operating systems such as Windows and pSOS.

There are three reasons why these groups contributed the changes. The first is that having a stable, standards-compliant implementation provided motivated individuals a test-bed to try out new ideas (for instance, a major contribution to the code was a technique to optimize the parsing step). Another very important reason was saving time by making the contribution part of the base software. Unless this was done the group may have to manage their contribution separately. This may involve merging their contribution to the base code each time a new release arrived. To avoid this, it was better to contribute the change. A third reason is that certain groups, having used the software, *wanted* to contribute something back.

One big advantage of using open source techniques is to allow other groups to examine the existing code and bring their unique expertise directly to bear. For instance, while the original author of the code (vkg) was well versed with the IETF standards, he found it too time consuming to keep up with the 3G standard as well. Thus contributions coming in from the 3G team reflected their expertise, and were a welcome addition to the code.

Finally, there was at least one indication that the open source approach improved in some ways on earlier efforts in the company to encourage and support reuse. In particular, for the SIP stack, one could just get the code and modify it as needed. As one interviewee pointed out, there was no need to "fight anybody" to make the changes, or to wait for approval. Nor was there a need to take "headcount" from other groups to make the changes, as product groups made the changes that they needed. This optimistic picture needs to be modified somewhat, as we discussed above when we talked about merging changes back into the main branch.

6. LESSONS LEARNED

We enumerate the lessons learned in the context of the open source project characteristics outlined in Section 2.

6.1 Requirements and Software Processes

Project management in corporate open source is a complex and challenging phenomenon. The tools and processes for front end planning are optimized within that project to effectively subdivide the customer deliverable into components matching the business structure. Subsequent processes within each organization further divide the deliverable until it reaches a size for which a feature development team can assume full responsibility. Precise work assignments and delivery timelines are established. However, this framework alone is insufficient for an open source project.

Our experience has shown that the corporate open source model requires the existing feature commitment and project management processes to be augmented with a functionally based system. This system must address the unique needs to prioritize across disjoint projects, to identify common work, to facilitate the resolution of architectural or scheduling outages, to track effort spent by the virtual team, and to ensure that the overall product meets the needs of all the customers. Functionally based quality assurance must also be supported.

6.2 Work Assignment and Incentive Structure

It is essential to recognize and accommodate the tension between cultivating a general, common resource on the one hand, and the pressure to get specific releases of specific products out on time. To accomplish the first objective it is critical to have management support for the "benevolent dictator". Keeping up with the changes being made to the code as new features are added and accepting contributions from the set of interested users is a time consuming task. The benevolent dictator should be the final arbiter on what goes into the code while preserving the architecture.

However, unlike traditional open source, the benevolent dictator cannot be concerned solely with a personal vision when making decisions about what features go in and how the software evolves. In a corporate setting, those features that attract the most paying customers must percolate to the top of the priority list. The benevolent dictator can still remain a powerful force for maintaining the conceptual and architectural integrity of the software, but business necessities must be respected as well.

Some developers will naturally gravitate towards understanding sizeable portions of the code and contributing in a similar manner, often on their own time. Such developers should be recognized by making them the owner of particular subsystems or complex

areas of the code (the "trusted lieutenant" phenomenon). More specifically, product groups could have designated experts in the open source project, and undergo more extensive training, perhaps in the form of multi-day workshops. This could perhaps be reinforced by providing incentives for project developers to have their code approved by the benevolent dictator for inclusion in the open source base code.

On a related note, another lesson learned is that there should be a larger core code review team. It is very unlikely for a developer in the CSS group to be cognizant of a feature being put into the code by another organization. Yet it is the duty of the CSS developers to ensure that the fidelity of the code is preserved with the addition of this new feature. Thus, it is necessary that a larger core review team be established. This team should consist of the benevolent dictator at the very least and those trusted lieutenants that are responsible for the subsystem which is impacted the most on the addition of the new code.

6.3 Software Architecture

Owning the source code and having many eyeballs contributing to it has made it easier to keep up with the numerous extensions to SIP. It is beyond the capability of one team to be knowledgeable in all aspects (for instance, the team that knows about performance optimization may not know much about security). Having access to the source code is invaluable since different individuals contribute in different ways to the cohesive whole.

One of the most important lessons is that independent strains of the software should be at best discouraged, or at worst, tracked carefully with an eye towards an eventual merge into a single branch or trunk. One of the biggest challenges we faced was how to merge independent changes done across two development lines. Each line had features and bug fixes that the other one wanted.

Since the standards and the technology were rapidly evolving, owning the source code allowed the company to respond quickly to customer needs. The authors of the paper have witnessed many commercial companies who have purchased SIP stacks from third party vendors; in such cases, these companies have to depend on the release schedules of the stack vendors. In developing solutions in the Internet timeline, this delay can provide extremely costly. Identifying states of flux such as this should be a valuable guide to finding opportunities for internal open source projects.

The interviewees strongly suggested that had the server not already existed by the time they experienced customer demand, they would have been forced by release schedules to purchase an existing stack rather than build their own. This supports the notion that a research or advanced technology group is a good location for starting this effort, since it requires the ability to anticipate needed technology before the market demands it, and to have funding not directly tied to particular product groups.

Once the decision has been made to foster a project as open source, disseminating information for it as widely as possible is a good strategy. Developers need to know that that common resource is important to the company, and is part of the company strategy. Tied to the information campaign at the grass roots level is the need to evangelize the open source effort internally at the corporate level. This allows a larger group of developers, managers, and project leaders to be aware of such an asset, use it, and if the need be, contribute to it. With the establishment of the SIP COE, the CSS group has a one-stop shop for downloading white papers on the technology to downloading the source code itself.

6.4 Tool Compatibility

And finally, it is important to move toward a common set of development tools, particularly version control and change management systems. Unlike traditional open source, the broader community of developers is constrained by the tool environments of their project work. Moving code among different version control systems in order to build a variety of products is a difficult problem, and introduces the temptation of maintaining separate forks for each project. Establishing a common repository from the onset and ensuring that it is not diluted appears to be a pre-requisite. However, this step is increasingly difficult in a corporate environment where each project has its own competing needs. We have been successful in establishing a common repository, but the success has been borne out of countless challenges in keeping different repositories synchronized.

A well thought out code distribution strategy is also important. In traditional open source, the recipient receives a tar file (or downloads the source tree) and proceeds from there. However, in a corporate setting, the distributed code has to fit in the load building strategy of a particular group. In some cases, the CSS group has had to accommodate the peculiarities of how a certain group builds the product.

7. OPEN QUESTIONS AND CONCLUSION

The experience we have gained leads to yet more open questions. As more projects are using the software, each one wants to customize it in its own manner. It is a challenge to allow such customizations while still preserving the core architecture. It would be extremely valuable to improve our understanding of how to design architectures to support open source style development. Clearly, the software architecture plays a major role in dictating the kinds of coordination that are required in doing the technical work, but we do not yet understand very much about how to architect software in ways appropriate for different development styles and organizational settings.

Another question concerns limitations of the open source development methodology. Can what we did at Lucent Technologies be replicated with any random project across all industries? We succeeded due to the convergence of many external forces and ideas. The manner of protocol development in the IETF was a big impetus to our project since we essentially tracked the earlier drafts; i.e., our implementation matured with the standard. When we started our work, Internet telephony was not viewed as the mainstream technology that it has now become. While we like to think it was clear foresight, we acknowledge as well the role of luck that we were correctly positioned when the company was looking for a SIP implementation that was standards compliant and that it owned. It is not clear, in general, how and when to initiate a project that can serve as a shared resource. It seems likely that a portfolio of technology investments would be required since prediction of future directions is so uncertain.

We also had a significant pool of users who were interested and capable developers, which seems to be a precondition for a successful open source project. If SIP servers were simply a well-understood and stable commodity technology, product groups could simply use it out of the box.

7.1 Success criteria

Based on this case study, we speculate that internal open source projects will have the best chance to succeed where

- a technology is needed by several product groups (hence there is reason to pool resources),
- the technology is relatively immature so that requirements and features are not fully known at the outset (so there is a need to evolve continuously),
- product groups have different needs and specific expertise in customizing the software for their needs (so everyone benefits from the contributions of each group), and
- the initial product has a sound, modular architecture (so that it is feasible to merge all the diverse changes into a single development branch).

We expect that future research will shed light on whether these speculations are correct.

7.2 Conclusion

We conclude by observing that this project has established a wider SIP community at Lucent Technologies. This has resulted in a shared technology asset that is highly competitive, is of higher quality, has decreased product generation costs, and has engaged the larger research, development and product management community within the company towards understanding how to build products that use this very important signaling technology.

8. ACKNOWLEDGMENTS

The third author was supported by a grant from an IBM Faculty Award, and National Science Foundation research grant IIS-0414698.

9. REFERENCES

[1] Arlein, R. and Gurbani, V., An Extensible Framework for Constructing Session Initiation Protocol (SIP) User Agents. *Bell Labs Technical Journal*, *9,* 3 (November 2004), p. 87-100.

[2] Broy, M., et al., Workshop on Open Source in an Industrial Context, http://osic.in.tum.de/

[3] Colbert, R.O., Compton, D.S., Hackbarth, R.S., Herbsleb, J.D., Hoadley, L.A., and Wills, G.J., Advanced Services: Changing How We Communicate, *Bell Labs Technical Journal, 6, 3,* (June 2001), pp. 211-228.

[4] Conway, M.E., How Do Committees Invent? *Datamation*, *14,* 4 (1968), p. 28-31.

[5] Cortes, M., Ensor, J.R., and Esteban, J.O., On SIP Performance. *Bell Labs Technical Journal*, *9, 3* (November 2004), p. 155-172.

[6] Gurbani, V., Garvert, A., and Herbsleb, J., A Case Study of Open Source Tools and Practices in a Commercial Setting, *Proceedings of the 5th ACM Workshop on Open Source Software Engineering (WOSSE)*, (May 2005), pp. 24-29.

[7] Dinkelacker, J., Garg, P., Miller, R., and Nelson, D., Progressive Open Source. In *Proceedings of the 2002 ACM International Conference on Software Engineering (ICSE '02)*, pp. 177-184, May 2002.

[8] Halloran, T.J. and Scherlis, W.L. *High Quality and Open Source Practices*. in *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*. 2002. Orlando, FL.

[9] Herbsleb, J.D. and Grinter, R.E., Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, *Sept./Oct.,* (1999), p. 63-70.

[10] Herbsleb, J.D. and Mockus, A., An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, *29,* 3 (2003), p. 1-14.

[11] MacCormack, A., Rusnak, J., and Baldwin, C., *Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code*, in *Harvard Business School Working Paper*. 2004: Boston, MA 02163.

[12] Mockus, A., Fielding, R., and Herbsleb, J.D., Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, *11,* 3 (2002), p. 309-346.

[13] Rosenberg, J., et al., SIP: Session Initiation Protocol, IETF RFC 3261, July 2002, http://www.ietf.org/rfc/rfc3261.txt

[14] Scacchi, W., Understanding the requirements for developing open source software systems. *IEE Proceedings on Software*, *149,* 1 (February 2002), p. 24-39.

[15] Shah, S. Understanding the Nature of Participation and Coordination in Open and Gated Source Software Development Communities. In *Annual Meeting of the Academy of Management*. 2004.

[16] Raymond, E., The Cathedral and the Bazaar. *O'Reilly Publishing Company, First Edition*, (February 2001).