Developing Architectural Documentation for the Hadoop Distributed File System

Len Bass, Rick Kazman, Ipek Ozkaya

Software Engineering Institute, Carnegie Mellon University
Pittsburgh, Pa 15213 USA
lenbass@cmu.edu,{kazman,ozkaya}@sei.cmu.edu

Abstract. Many open source projects are lacking architectural documentation that describes the major pieces of the system, how they are structured, and how they interact. We have produced architectural documentation for the Hadoop Distributed File System (HDFS), a major open source project. This paper describes our process and experiences in developing this documentation. We illustrate the documentation we have produced and how it differs from existing documentation by describing the redundancy mechanisms used in HDFS for reliability.

1 Introduction

The Hadoop project is one of the Apache Foundation's projects. Hadoop is widely used by many major companies such as Yahoo!, E-Bay, Facebook, and others. (See http://wiki.apache.org/hadoop/PoweredBy for a list of Hadoop users.) The lowest level of the Hadoop stack is the Hadoop Distributed File System 2. This is a file system modeled on the Google File System 1 that is designed for high volume and highly reliable storage. Clusters of 3000 servers and over 4 petabytes of storage are not uncommon with the HDFS user community.

The amount and extent of documentation of the architecture 3 that should be produced for any given project is a matter of contention. There are undeniable costs associated with the production of architectural documentations and undeniable benefits. The open source community tends to emphasize the costs and downplay the benefits. As evidence of this claim, there is no substantive architectural documentation for a the vast majority of open source projects, even the very largest ones. The existing description of the architecture of most widely used open source systems tend to be general descriptions rather than systematic architectural documentation targeted for the system's stakeholders 4 .

This paper describes the process we used to produce architectural documentation with emphasis on what is different about producing documentation for open source projects. This production was the first step in a more ambitious project that will analyze the community for evidence as to the value of the documentation but we have nothing to report on that front as yet.

HDFS makes two assumptions that take it out of the realm of a standard file system: it assumes high volumes of data in primarily a write-once, read-many-times environment. The only block size that HDFS supports is 64Mbytes. There is very little synchronization supported since the kinds of applications for which it is designed are primarily batch – collect data and process it later. The second assumption that HDFS makes is that it will run primarily on commodity hardware. With 3000 servers, hardware failures, even with all RAID devices, become a normal occurrence. As a consequence the software was designed to handle failure smoothly. Since the software must handle failure in any case, use of commodity hardware makes the use of a multi-thousand server cluster much more economical.

The structure of this paper is that we will first describe our idealized process for producing more detailed architectural documentation. We then discuss what we actually did and how it differed from the idealized process. Throughout the paper we use the description of the HDFS availability strategy as illustrative of both the existing documentation and our additions to it.

2 Our Process for Developing the Documentation

When writing architectural documentation it is necessary to have an overview of what the system components are and how they interact. When there is a single architect for the system, the easiest route is to simply talk to this person. Most open source projects, however, do not have a single identifiable architect—the architecture is typically the shared responsibility of the group of committers.

The first step of our documentation process is to gain this overview. Subsequent steps include elaborating the documentation and validating and refining it. To do this we needed to turn first to published sources.

2.1 Gaining the Overview

HDFS is based on the Google File System and there are papers describing each of these systems 1, 2. Both of these papers cover more or less the same territory. They describe the main run-time components and the algorithms used to manage the availability functions. The main components in HDFS are the NameNode that manages the HDFS namespace and a collection of DataNodes that store the actual data in HDFS files. Availability is managed by maintaining multiple replicas of each block in an HDFS file, recognizing failure in a DataNode or corruption of a block, and having mechanisms to replace a failed DataNode or a corrupt block.

In addition to these two papers, there is an eight page "Architectural Documentation" segment on the Apache Hadoop web site 5. This segment provides somewhat more detail than the two academic papers about the concepts used in HDFS and provides an architectural diagram, as shown in Figure 1.

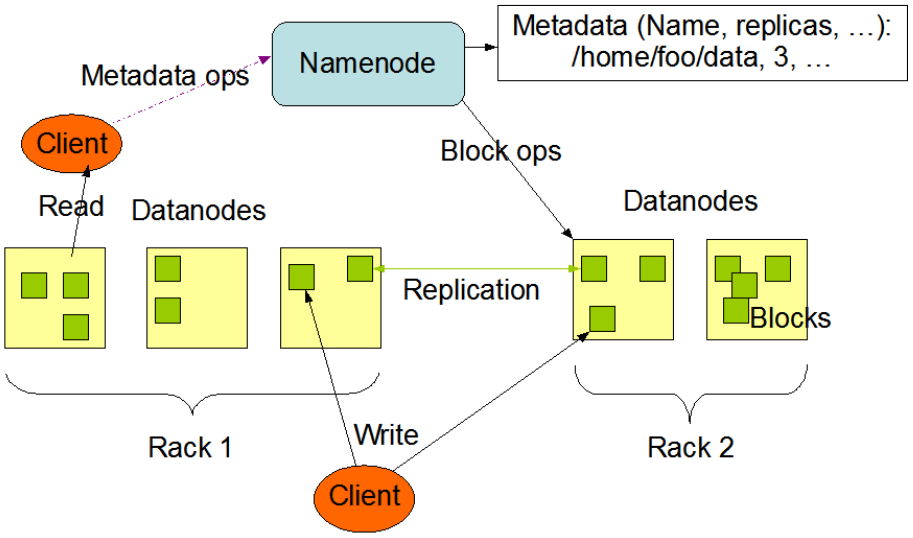

Figure 1. HDFS Architecture Diagram from 5

Code level documentation (JavaDoc) is also available on the HDFS web site. What currently exists, then, are descriptions of the major concepts and algorithms used in HDFS as well as code-level JavaDoc API documentation.

What is missing from the existing documentation can be seen by considering how architectural documentation is used. Architectural documentation serves three purposes: 1) a means of introducing new project members to the system, 2) a vehicle for communication among stakeholders, and 3) the basis for system analysis and construction 3 6. These uses of architectural documentation include descriptions of the concepts and, where important, the algorithms. But architectural documentation, to be truly useful for those who wish to modify the system, must also connect the concepts to the code. This connection is currently missing in the HDFS documentation. A person who desires to become a contributor or committer needs to know which modules to modify and which are affected by a modification. Communication among stakeholders over a particular contribution or restructuring is also going to be couched in terms of the relation of the proposed contributions to various code units. Finally, for system construction, maintenance, and evolution to proceed, the code units and their responsibilities must be unambiguously identified. Lack of such focused architecture documentation can assist contributors become committers faster. It could also assist addressing many current open major issues. As of April 12, 2011 out of the 834 total issues in HDFS Jira 628 of the issues are major issues.

Architectural documentation occupies the middle ground between concepts and code and it connects the two. Creating this explicit connection is what we saw as our most important task in producing the architectural documentation for HDFS.

2.2 Expert Interview

Early in the process of gaining an overall understanding of HDFS, we interviewed Dhruba Borthakur of Facebook, a committer of the HDFS project and also the author of the existing architectural documentation posted on the HDFS web site 5. He was also one of the people who suggested that we develop more detailed architectural documentation for HDFS. We conducted a three hour face to face interview where we explored the technical, historical, and political aspects of HDFS. Understanding the history and politics of a project is important because when writing any document you need to know who your intended audience is to describe views that are most relevant to their purposes 3.

In the interview, we elicited and documented a module description of HDFS as well as a description of the interactions among the main modules. The discussion helped us to link the pre-existing architectural concepts—exemplified by Figure 1—to the various code modules. The interview also gave us an overview of the evolutionary path that HDFS is following. This was useful to us since determining the anticipated difficulty of projected changes provides a good test of the utility, and driver for the focus, of the documentation. Figure 2 shows a snippet from our interview and board discussions where Dhruba Borthakur described to us the three DataNode replicas in relationship to the NameNode.

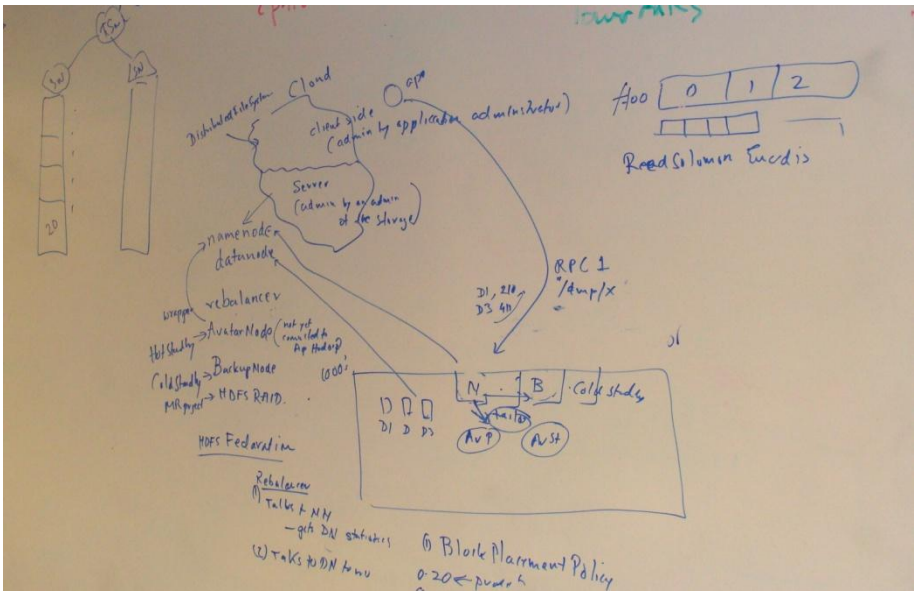

Figure 2. Elicitation of Architectural Information

2.3 Directory Structure

A final item that proved very helpful is the directory structure of HDFS. The code is divided cleanly into the following pieces:

- The library used by the client to communicate with the NameNode and the DataNodes.
- The protocols used for the client communication
- The NameNode code
- The DataNode code
- The protocols used for communication between the NameNode and the DataNodes.

In addition, there are a few other important directories containing functionality that the HDFS code uses, such as Hadoop Common.

2.4 Tool Support

An obvious first step in attempting to create the architectural documentation was to apply automated reverse engineering tools. We employed SonarJ 7 and Lattix 8, both of which purport to automatically create architectural representations of a software product by relying on static analysis of the code dependencies. However, neither of these tools provided useful representations although they did reveal the complexity of the dependencies between Hadoop elements. For example, Figure 3 shows an extracted view of the most important code modules of HDFS, along with their relationships, produced by SonarJ.

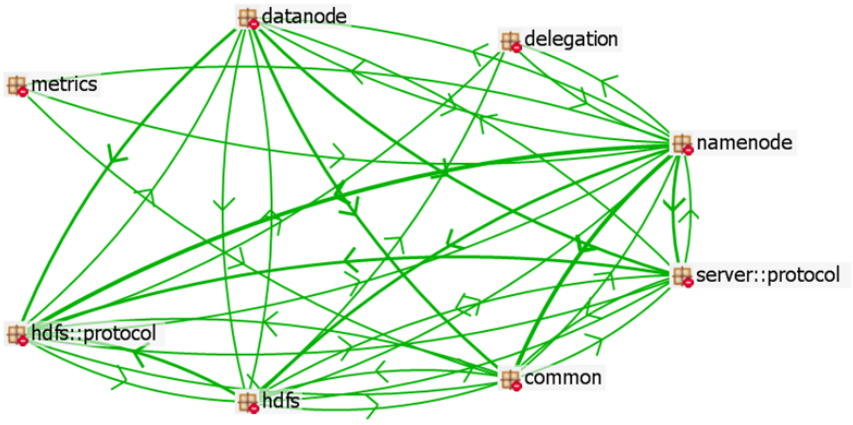

Figure 3. Module Relationships in HDFS

What are we to make of this representation? It appears to be close to a fully connected graph. Is the code a “big ball of mud” 9? The answer lies in the purpose

and goals of the architecture. The main quality attribute foci of HDFS are performance and availability. These concerns dominate the architectural decisions and the discussions amongst the project's committers. Of significant, but decidedly lesser concerns, are qualities such as modifiability and portability. The process Hadoop follows in handling modification is a planned evolutionary processes where a committer suggests alternative design, it is vetted among the key committers, and then planned for an agreed upon future release cycle. The goals of the project should be aligned with the focus of the architecture. Since performance and availability were the top goals of HDFS, it is not surprising that these concerns shaped the architectural decisions. Since modifiability and portability were of lesser concern, it is also not surprising that these qualities were not strongly reflected in the architectural structures chosen.

The reverse engineering tools SonarJ and Lattix are primarily focused on these latter concerns—modifiability and portability. They aid the reverse engineer in determining the modular and layered structures in the architecture by allowing the definition of design rules to detect violations for such architectural structures. We thus see a mismatch between the goals of the tools and the goals of HDFS. For this reason, the structures that these tools were able to automatically extract were not particularly interesting ones, since they did not match the goals of the project and the important structures in the architecture. HDFS does not have any interesting layering, for example, since its portability concerns are, by and large, addressed by the technique of "implement in Java". The governing architectural pattern in HDFS is a master-slave style, which is a run-time structure. And modifiability, while important, has been addressed simply by keeping the code base at a relatively modest size and by having a significant number of committers spending considerable time learning and mastering this code base.

The modest code size, along with the existing publications on the availability and performance strategies of HDFS allows us to document the architecture by tracing the key use cases through the code. While this is not an easily repeatable process for larger open source projects, it proved to be the most accurate and fit for purpose strategy for creating the architecture documentation of HDFS.

This lack of attention to specific architectural structures aimed at managing modifiability is a potential risk for the project as it grows, since it makes it difficult to add new committers—the learning curve is currently quite steep. Our architectural documentation is one step in addressing this risk. Another step that the HDFS committers could take is to simplify the "big ball of mud".

3 Elaboration

This project began Nov. 1, 2010 and the interview with Dhruba Borthakur took place on Nov. 22, 2010. Thus, it took a month to gain an overview of HDFS. December was devoted to exploring the architecture using the reverse engineering tools and the month of January was devoted to writing a first version of the architectural

documentation. It is in the elaboration phase that value is added to the existing materials.

The elaboration phase of the architectural documentation is when the connections between the concepts and the algorithms are made manifest. Consider the following section from the existing architectural documentation (found at http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.html) about one of the key mechanisms for maintaining high data availability: the heartbeat.

Data Disk Failure, Heartbeats and Re-Replication

Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS any more. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

Our elaboration of this concept, as we have produced it in the architectural documentation, is:

Heartbeats are the mechanism by which the NameNode determines which DataNodes are currently active. There is a heartbeat monitor thread in NameNode that controls the management.

NameNode maintains information about the DataNodes in DataNodeDescriptor.java. DataNodeDescriptor tracks statistics on a given DataNode, such as available storage capacity, last update time, etc., and maintains a set of blocks stored on the DataNode. This data structure is internal to the NameNode. It is not sent over-the-wire to the Client or the DataNodes. Neither is it stored persistently in the fsImage (namespace image) file.

A DataNode communicates with the NameNode in response to one of four events.

1. *Initial registration. When a DataNode is started or restarted it registers with the NameNode. It also registers with the NameNode if NameNode is restarted. In response to a registration, NameNode creates a DataNodeDescriptor for the DataNode. The list of the DataNodeDescriptors is checkpointed in fsimage (the namespace image*

file). Only the DataNodeInfo part is persistent, the list of blocks is restored from the DataNode block reports.

2. *In response to a heartbeat request from the NameNode. If NameNode has not heard from a DataNode for some period of time, it sends a request for a Heartbeat. If this request does not generate a response, the DataNode is considered to have failed and each of the replicas it maintains must be created on a different DataNode. When the DataNode has reported, NameNode:*
    - *Records the heartbeat, so the DataNode isn't timed out*
    - *Adjusts usage stats for future block allocation*

    If a substantial amount of time passed since the last DataNode heartbeat then NameNode requests an immediate block report.

3. *In response to a blockReport() request from the NameNode. NameNode may request the DataNode to report all of the replicas that it is currently maintaining. NameNode does this when it has reason to believe that its list of blocks in the DataNode is not up to date. i.e., on start up or if it has not heard from the DataNode for some period of time.*

4. *Completion of a replica write. When the DataNode has successfully written a replica, it reports this event through a blockReport().*

The differences between the original documentation 5 and the new version that we have produced are as follows:

- *Much more detail*. Rather than giving a general description of the concepts, the specific interactions between a DataNode and the NameNode are described.
- *Code is explicitly named*. The classes that contain the code that provides the heartbeat responsibility are identified.

Subtle optimizations are identified. For example, a blockReport() sent by the DataNode to the NameNode indicates that the DataNode is alive and there is no need for a heartbeat query to that DataNode. At the time of writing of this paper we are also working on adding sequence diagrams of major use cases to further highlight the architectural details mapping the architecture documentation more explicitly to the code.

4. Validation and Refinement

The final phase of the production of the architectural documentation is to validate it. We have now received comments on our draft from two committers of HDFS - Dhruba Borthakur of Facebook and Sanjay Radia of Yahoo!. Based on their comments we have modified the draft architectural documentation. At the time of

writing this paper, we are preparing the documentation for publication on the HDFS web-site and appropriate blogs. Before doing so, we are creating baseline metrics on the existing state of the basic metrics that we can track to architecture documentation such as number of committers, contributors, and surveys that we will conduct with them to establish a baseline impression of the state of the architecture for HDFS (like actual versus perceived architecture).

5. Structure of the Documentation

The documentation that we have produced has 6 major sections. These are

1. *Introduction*
2. *HDFS Assumptions and Goals*. This section talks about the design rationale and major quality attribute concerns for HDFS.
3. *Overview of HDFS Architecture*. This section introduces the three types of processes within HDFS – the application, the NameNode, and the DataNode.
4. *Communication among HDFS elements*. This section describes the four canonical runtime interactions between the three types of HDFS processes. These interactions, as shown in Figure 1, are: Application code <-> Client, Client <-> NameNode, and Client <-> DataNode, NameNode <-> DataNode.
5. *Decomposition and Basic Concepts of HDFS elements*. This section describes how each of the basic elements reacts to client requests to create, write, read, and close files. It also describes the modes and thread structure within NameNode and how these modes and threads are used to manage the file systems and provide high availability.
6. *Use Cases*. The basic use cases of create, write, read, and close are described in terms of sequence diagrams.

6. Discussion

We will now discuss three aspects of creating architectural documentation that are lessons learned from this process: where to start, how to evolve the documentation, and the use of tools. We will also discuss how the production of architectural documentation for an open source project differs from the production of architectural documentation for a closed project.

6.1 Where to start

There were two documents that helped us get started in documenting HDFS: the Google File System paper (the Google file system was the original model for HDFS)

and the existing architectural documentation on the web-site. These two documents provided a good start on our gaining an early understanding of HDFS. What would we have done if this level of documentation had not existed?

Whether or not there is existing documentation, our process calls for interviewing experts. The documentation that exists is invariably out of date (if it were not, we would not be doing this job) and much of the information that we require typically resides in the heads of just a few individuals. These individuals are usually very busy and without sufficient time or interest to produce the architectural documentation. For HDFS, one interview was sufficient. If the pre-existing level of documentation is not sufficient to gain an overview level of understanding, then more interviews may have been necessary. The interview was quite lengthy, involved multiple drawings on a whiteboard and, although it did not work out, we had hopes of arranging another interview in the same trip. Although face to face interviews are difficult and expensive to arrange, it is hard for us to image the same results from a video conferencing meeting.

Our notes from the interview contain several photos of drawings from a white board. It is possible that additional information could have been gained from either e-mail or telephone conferences, but we did not feel the need for that for this case, In an open source project, the committers are usually easy to find, although possibly difficult to arrange time with.

To summarize, the techniques for gaining an understanding of the architecture from dealing with the committers include face to face meetings, off line communication, and telephone conferences.

6.2 Evolution

Systems evolve and (hopefully) more slowly, architectures evolve. This means that the architecture documentation in an evolving system may quickly become out of date. Using HDFS as an example, this fear seems to be overblown. The fundamental structures of HDFS—for example, the separation and relationships between client, DataNode, and NameNode—have not changed since its inception. Of course, the details of each of these elements and their interactions have evolved, but at the architectural level there was considerable stability.

A recent major change is the addition of the ability to append data to an existing file. In architectural terms, this involved multiple classes and the addition of a major new functionality. Yet in terms of the documentation, a search of the documentation we produced finds several one-word references to *append* plus an 11 line paragraph describing how *append* is different than *write*. Generating the additional architectural documentation associated with *append* would have been the work of just a few hours.

Major changes currently being considered for HDFS are a refactoring of the code and several different proposals being considered to break the current design of an HDFS deployment being limited to one cluster. These two types of changes raise different issues in terms of the evolution of the architectural documentation.

- *Refactoring to simplify the code structure*. Refactoring the code would not change the concepts or the algorithms used, but it would have an impact on the mapping of the important concepts to classes. Yet the changes to the architectural documentation can be kept to a minimum. A refactoring will add new classes, modify existing classes, or deleting classes. Any major new class will be constructed from portions of existing classes. We can match a list of classes being modified or deleted with the classes mentioned in the documentation. For each match, the changes to the documentation will consist of adding a class name or removing a class name. These are minimally intrusive changes.
- *Breaking the current limitation of a single cluster per deployment*. This type of change is more far reaching and will have more impact on the documentation. Yet this type of change is not made radically or quickly. In fact, the discussion of the proposed changes can be found in the Jira bug-tracking system prior to the change actually taking place.

The discussion in Jira provides exactly the type of information that needs to be captured in the architectural documentation. Consider the following example, open issue from HDFS Jira, HDFS-1783 created March 24, 2011 by Dhruba Borthakur:

> *The current implementation of HDFS pipelines the writes to the three replicas. This introduces some latency for realtime latency sensitive applications. An alternate implementation that allows the client to write all replicas in parallel gives much better response times to these applications.*

Although architectural documentation is created once and then needs to be maintained and evolved, we argued here that if one considers the type of evolution that a system like HDFS undergoes once it has become successful, the concomitant evolution to the architectural documentation is relatively minor and painless.

6.3 The Use of Tools

Tools are most useful in the elaboration stage of the documentation. As discussed above, tools are not much help in gaining an initial understanding of the concepts and algorithms, and may be of limited use in understanding the module structure. But tools can be very useful in tracking the effects of a method call or the use of a particular class.

One tool that is particularly useful is the call graph. A call graph enables tracking how a call to “write” by the client goes through NameNode. It is not the best way to understand that NameNode is not involved with the data transfer, per se but it will provide a track through the classes that allocate blocks.

As discussed above, we originally tried to create a module view with tool support but that effort was unsuccessful. The tools that we used that support the construction of a module view require some initial guesses as to the decomposition of the modules. Beyond the decomposition of HDFS into the client, NameNode, and DataNode, finding further decompositions proved unsuccessful for us. As a result,

the architectural documentation that we produced only has those three major components identified.

6.4 Open Source Specifics

One distinction between producing architectural documentation for an open source project and a proprietary project comes from the openness and availability of discussions about issues. In an open source project, Jira, mailing lists and bulletin boards become the repository of these discussions and they can be mined for rationale information. In a closed source project we must rely upon the availability, good will, and good memory of individuals. Although in principle there is nothing to stop closed source projects from adopting these practices, in our experience, we have rarely seen evidence of their existence.

There are other factors that are traditionally cited as distinctive to open source activities—multiple eyes, requirements arising from the contributors rather than from an explicit elicitation process, and so forth. But none of these other distinctive characteristics of open source appears to substantially affect the process of creating architectural documentation.

7. Next Steps

The production of the architectural documentation is the first step in a more ambitious research project to measure the *impact* of architectural documentation. The current group of committers of HDFS is stressed because of their HDFS-related workload and would like to grow the HDFS committer community. Currently there are 221 contributors as opposed to 28 committers. Our conjecture is that the existence of architectural documentation will shorten the learning curve for potential contributors and committers, thus lowering the bar to entry.

To test this conjecture we have created improved architectural documentation for HDFS, and begin disseminating it on May 19, 2011 via http://kazman.shidler.hawaii.edu/ArchDoc.html. We announced the availability of the documentation to the Hadoop community through HDFS Jira (issue number HDFS-1961)

In addition, we are collecting a number of project measures. We will measure the usefulness of the documentation by tracking how often it is downloaded and how often it is mentioned in discussion groups. We are tracking project health measures, such as the growth of the committer group, and the time lag between someone's appearance as a contributor and their acceptance as a committer and other measures. And we are tracking product health measures, such as the number of bugs per unit time and bug resolution time.

This study is much more long-term than the production of the architectural documentation, although it crucially depends on the documentation as a first step. We will report on our results in due course.

8. Acknowledgements

The work was supported by the U.S. Department of Defense. We would also like to thank Dhruba Borthakur and Sanjay Radia for their assistance.

9. References

1. Ghemawat, S., Gobioff, H., Leung, S-T.: The Google file system. ACM SIGOPS Operating Systems Review - SOSP '03, Volume 37 Issue 5, pp. 23-43 (2003)
2. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System, IEEE 26th Symposium on Mass Storage Systems and Technologies, Incline Village, NV, 1-10 (2010)
3. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. 2nd ed., Addison-Wesley (2010)
4. Brown, A., Wilson, G.: The Architecture of Open Source Applications (2010), http://www.aosabook.org/en/index.html (accessed June 6, 2011)
5. Apache Hadoop, HDFS Architecture, http://hadoop.apache.org/common/docs/r0.19.2/hdfs_design.html (accessed Apr. 7, 2011)
6. ISO/IEC 42010:2007 – Recommended Practice for Architectural Description of Software-intensive Systems (2007), http://www.iso-architecture.org/ieee-1471/. (accessed June 6, 2011)
7. SonarJ, http://www.hello2morrow.com/products/sonarj, (accessed Apr. 9, 2011)
8. Lattix, http://www.lattix.com, (accessed Apr. 9, 2011)
9. Foote, B., Yoder, J.: Big Ball of Mud. Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97), Monticello, Illinois (1997)