Software Development by Ubiquitous Crowdsourcing

Michael N. Huhns
University of South Carolina
huhns@sc.edu

ABSTRACT
The benefits of the open-source approach to software development have not been fully realized, because the number of software developers is still relatively small and orders of magnitude smaller than the number of users. Developers typically are experts in computing, whereas users typically have domain expertise: this produces a disparity in viewpoints, causing a mismatch between the developed software and its desired use. Moreover, the proposition, "given enough eyeballs, all bugs are shallow," would take on much greater significance, if a larger fraction of the users could also be developers. This paper describes how to take advantage of code, components, and designs contributed by crowds. It uses agent-based wrappers to manage the necessary collaboration and competition, allowing the contributions to be used alongside their existing counterparts until their behavior and features can be assessed.

Author Keywords crowdsourcing software, multiagent systems, software robustness

ACM Classification Keywords D.2 [Software Engineering]: Coding tools and techniques

General Terms design**,** human factors, reliability

INTRODUCTION

An interviewer asked Linus Torvalds if Linux's user base or developer base was more important [5]. Torvalds answered that he did not see the need to distinguish between them, because in Linux users can also be developers. In reality, there is a large distinction, because the number of developers (there were approximately 800 contributors to the latest release of the kernel) ranks in the hundreds and the number of users ranks in the millions.

Similarly, the number of open source developers for Tomcat, the open-source middleware from Apache, is ~25 and has been that number for four or five years, while the growth of Tomcat has been astronomical [3].

The distinction is not restricted to open-source software: Microsoft Windows has approximately the same number of developers as Linux, but the resultant operating system is used by more than 100 million people.

In another facet of the distinction, [2] observed that software tends to resemble the organization that built it: this is often vastly different than the community that uses it, making it more difficult for users to understand it.

Based on this, would it be beneficial for the development of Linux, or any open-source software system, for the distinction between users and developers to be reduced and a much larger fraction of the users also to be developers? If the answer is yes, then how might this be made to occur?

Superficially, it would require (1) an easy way for users to contribute to the software development process and (2) a way to accommodate and manage the contributions. This paper describes both, with an emphasis on the latter. Our research is based on the following premises:

Premise 1. Users and customers will be more satisfied with a software system if they have a stake and involvement in its development.

Premise 2. Software will be more robust if more "eyeballs" are looking for and correcting bugs.

Premise 3. Software will be more understandable, and thus usable, if end-users participate more in its design.

Given these premises, we are investigating how multiple versions of components can be managed to take advantage of their individual strengths, while collectively being unaffected by the individual weaknesses. Moreover, there are classes of software systems for which a crowdsourcing approach is beneficial and classes for which it is not. We can determine these classes and identify the software development models most appropriate for each class.

BACKGROUND AND ANALYSIS

The influential essay, "The Cathedral and the Bazaar," promulgated a view of open-source software development that contrasts two different development models [10]:

1. The *Cathedral* model, in which source code is available with each software release, but code developed between releases is restricted to an exclusive group of software developers.
2. The *Bazaar* model, in which the code is developed over the Internet in view of the public. Linux development has been following this model.

It posits that *"given enough eyeballs, all bugs are shallow,"* which is termed Linus' law: the more widely available the source code is for public scrutiny and testing, the more rapidly all forms of bugs will be discovered. It claims that much more time and energy must be spent looking for bugs in the Cathedral model, since the code is available only to a few developers. The essay helped convince many open-

UbiComp'10, September 26–29, 2010, Copenhagen, Denmark.
ACM 978-1-60558-843-8/10/09.

source and free software projects to adopt Bazaar-style models, including the Mozilla and Firefox projects.

However, the superiority of a Bazaar-style model of open-source software development has not been evident, partly because the size of the developer community for both proprietary and open-source software is roughly the same (there are not "more eyeballs" in open-source projects), and partly because it is still orders of magnitude smaller than the size of the user community.

To meet these challenges, a crowdsourcing approach exploiting concepts from N-version programming, multiagent systems, and social consensus appears promising. We address each of these concepts next.

N-Version Programming

N-version programming [6,9], also called dissimilar software and design diversity, is a technique for achieving software robustness. First considered in the 1970's, it consists of *N* disparate and separately developed implementations of the same functionality. It has long been recognized that the use of multiple versions of software is a potential solution to the reliability problem, so why has it been employed only for a few applications, such as critical satellite systems? The reasons for its lack of use, and our approach for addressing them, are:

- *N* versions require *N* times as much memory. *Due to storage advances memory is not typically a limitation.*
- Executing *N* versions requires *N* times as many CPU cycles. *Because software is not easily parallelized and compilers cannot always distribute machine code uniformly, CPUs on multicore chips are often idle—they are thus available to execute multiple versions.*
- It is not clear where the *N* versions come from. *Algorithm versions can be solicited from the developer and end-user communities and Web services themselves are a source of diverse algorithms.*
- *N* implementations based on the same flawed specification might still result in a flawed system. *Our approach cannot account for specification flaws.*
- Even *N* versions developed independently might fail dependently. *We will provide measures of dependence.*
- Combining the results of *N* versions is unspecified, seemingly different for each application of *N* versions, and left to a custom module that might not be reliable. *Our research has shown that a generic wrapper agent can be used to combine a variety of algorithms.*

Agent-Oriented Software System Development

The most common technique for hardware, redundant components, is inappropriate for software, because having identical copies of a module provides no benefit. Software reliability is thus a more difficult and still unresolved problem [1,9]. Multiagent systems have been investigated to increase reliability, and this has led to an interest in combining them with software engineering methodologies.

The focus of this paper is on extending traditional software development methodologies to widespread system development by crowds other than expert developers. The behavior of the resultant systems will depend on their construction and execution environment.

When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics relevant to our needs [4]:

- Active agent-based versions can benevolently compensate for the limitations of other modules.
- Agents can represent multiple viewpoints and can use different decision procedures, therefore increasing diversity and reliability.

We build upon these benefits to increase the reliability of software systems.

Software Reliability and Redundancy

Hardware robustness is typically characterized in terms of faults and failures; equivalently, software robustness is typically characterized in terms of bugs and errors. The general aspects of dealing with faults and bugs are: (1) predict their occurrence, (2) prevent their occurrence, (3) estimate their severity, (4) discover them, (5) repair or remove them, and (6) mitigate or exploit them.

Software failure estimation uses statistical techniques [8,9]. Reducing failure rate is dependent on good software engineering techniques and processes. Good development and run-time tools can aid error discovery and repair. Mitigation techniques mainly depend on redundancy. However, achieving an appropriate level of redundancy in software systems is difficult. If a hardware system fails, an identical replacement can provide continuity, but identical software systems would fail in identical ways under the same demand. Moreover, code cannot be added arbitrarily to a software system. The challenge is to design the software system so that it can accommodate the additional components and take advantage of the redundant functionality.

We hypothesize that agents are an appropriate abstraction for adding redundancy and that the software environment that takes advantage of them is akin to a society of such agents, where there can be multiple agents filling each societal role. Agents by design know how to deal with other agents, so they can accommodate additional agents naturally. They also are able to negotiate over and reconcile different viewpoints.

Social Consensus

The Social Web (essentially Web 2.0 technologies) derives content and information organization from large-scale collaboration. Folksonomies have the ability to form stable structures by forming a consensus over large sets of tags. A

similar collective categorization scheme could be used as an initial organization for contributed software. However, it does not provide a solution for the crowdsourcing of software development. For example, Wikis have the problem that the last one to edit an entry "wins." The design and development of software is not a democratic process and voting is often inappropriate for deciding which module might be best [11].

DEVELOPMENT BY CROWDSOURCING

Software robustness can be increased in an efficient manner based on 1) support for contributions from a large and widespread developer community at multiple levels of a software architecture, and 2) an adaptive, multiagent execution environment for managing collaborative and competing versions.

There are different types of software. Analogous to n-tier architectures, we have identified as distinct types interface software, application software, middleware, and backend software: (1) each might or might not be suitable for crowdsourcing development, (2) each might require a different development methodology, and (3) each might require a different way of combining the contributions.

Contributions would be initialized with a neutral rating and subsequently would increase or decrease in importance and preferential use over time. For example, if a module always fails, then eventually it will be removed. If a module often finishes first with a correct or common answer, then it will in the future be invoked more preferentially.

Research and empirical results, such as [6,7,8,9], show that multiversion software increases reliability, albeit not at the expected rate. It has been shown that even independently developed versions fail dependently. One explanation for this is that people tend to make the same mistakes in similar circumstances [6]. This indicates that changing the circumstances may reduce failure dependencies.

We are investigating the feasibility of extending established software design methods to enable a *comparison of dependency* between different versions based on their *specifications* only. We have had success in *predicting* dependency and, thus, diversity, by using standard software metrics. The availability of such metrics and supporting tools, along with the development of the multiagent infrastructure for melding and executing versions, allows redundant algorithms to be obtained from people outside of the traditional development community.

Fundamentally, the amount of redundancy required is well specified by information and coding theory. Assume each software module in a system can behave either correctly or incorrectly (the basis for unit testing as used by most software development organizations) and is independent of the other modules (so they do not suffer from the same faults). Then two modules with the same intended functionality are sufficient to detect an error in one of them, and three modules are sufficient to correct the incorrect behavior (by choosing the best two-out-of-three). More generally, based on a notion of Hamming distance for error-correcting codes, $4m$ independent agents can detect $m-1$ errors in their behavior and can correct $(m-1)/2$ errors.

Redundancy must be balanced with complexity, which is determined by the number and size of the components chosen for building a system. That is, increasing the number of independent versions increases redundancy, but also increases the complexity of the system. Further, choosing the proper *size of the modules* is crucial, because smaller modules are simpler to handle but their interactions are more complicated because there are more modules.

A Multiagent Execution Environment

Our second thrust addresses the need for developing an adaptive infrastructure to handle multiversion software efficiently and correctly. For this, we use a multiagent system where agents encapsulate the different software versions. The agents are produced by wrapping a contributed module or algorithm with a minimal set of agent capabilities. To specify these capabilities, we will evaluate the needs of multiversion programming systems. In particular, we are planning to investigate the problem of *version granularity* (i.e., the optimal size of a version to increase reliability at minimal cost), *decision making strategies* (i.e., voting protocols and group negotiation), and *adaptive behavior* (i.e., accommodating changes in the environment and learning about which versions to trust).

A high-level view of our execution environment is shown in Figure 1. The agent-based framework can support competition among versions, flexible granularity (e.g., entire software system vs. software components), and a variety of execution models (e.g., all versions execute in parallel vs. a new version replaces a failed version).

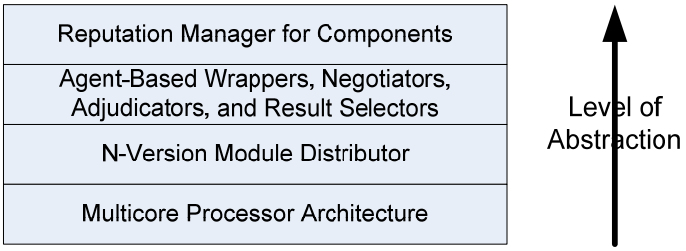

Figure 1. Layered architecture for software execution environment

In our distributed approach the algorithms jointly decide which one(s) should perform the processing. Conventional algorithms do not typically have such a distributed decision-making ability, so the agent-based wrapper enables an algorithm to participate in distributed decision-making. An agent in this system would have to know about itself: what it needs, what it can accomplish, and how.

Version Granularity

Software systems are usually constructed from several components. Each component performs specific tasks and interacts with the others. Increasing the reliability of large,

complex systems via an N-version execution environment raises the question of what should constitute a "version." Considering the entire system as a single version has the advantage of a small population of versions and fewer control operations needed for the execution. However, achieving independence among the versions becomes complex and their execution rigid. Smaller version size, such as replicating each procedure, clearly increases flexibility of the execution. Further, developing and evaluating independent versions will be simpler than for large, complex programs. However, the smaller granularity will make it necessary for the system to be controlled carefully to compose the necessary modules correctly.

PRELIMINARY RESULTS

We collected a number of algorithms in four domains—geographic location control, sorting, list-reversing, and evaluation of postfix arithmetic expressions—each written by a different person and therefore having different input and output signatures and performance characteristics. The programmers were undergraduate computer science majors and the work was done as standard homework assignments. The students were unaware that their algorithms would be used in our tests for robustness, so the algorithms did not have any special features that would bias our results. We converted each algorithm into an agent composed of the algorithm without any modifications and a wrapper for that algorithm. The wrapper knows nothing about the inner workings of its associated algorithm. It has knowledge only about the external characteristics of its algorithm, such as the data type(s) it requires and produces, its time complexity, and its space complexity.

First, 30 students each implemented an agent for a control-system application as a concurrently executing Java thread and interacting through a base class environment. The goal of the agents was to form themselves into a geometric circle in a plane. The agents each understand what a circle is, what it means to be part of a circle, where the nearest agents are located, and an estimate of how close the group is to being in a circle. The agents can reason about where they should be and the direction they should move to get there. We introduced a few agents that do not have the ability to move properly. The group overcomes this by helping to move the misbehaving agents and produces an acceptable circle. We have anecdotal evidence, via one comparison, that such an implementation can be constructed more rapidly and robustly than conventionally.

We have compared several adjudication approaches, including majority voting, consensus voting, maximum likelihood voting, recovery blocks, consensus recovery blocks, and combinations. These assume the reliabilities of individual versions are known and the versions are independent. Unfortunately, we have found that independently developed versions tend to fail dependently. Fortunately, we have also found that code complexity measures (e.g., source lines of code, percent branch statements, complexity, number of statements per method, and average block depth) are an indirect means of representing dependencies among versions and, thus, estimates of proneness to coincident failures [11].

CONCLUSIONS

Producing robust software has never been easy, and the crowdsourcing approach recommended here would have major effects on the way that software systems are constructed. We plan to explore answers to the following questions: What types of software are amenable to the crowdsourcing development approach? Is there an optimal granularity for the size of the agent-based components? How many versions are needed for a desired correctness? How can independently constructed components reconcile their behaviors? The result will be improved software that more closely behaves as users and stakeholders desire.

REFERENCES

1. Algirdas Avizienis, "Toward Systematic Design of Fault-Tolerant Systems," *IEEE Computer*, Vol. 30, No. 4, 1997, pp. 51-58.
2. Fred Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, MA, 1995.
3. Bob Brown, "Open source's future: More Microsoft, bigger talent shortages," *Network World*, 11/27/2007.
4. Helder Coelho, Luis Antunes, and Luis Moniz, "On Agent Design Rationale," in *Proceedings of the XI Simpósio Brasileiro de Inteligência Artificial (SBIA)*, Fortaleza (Brasil), October 17-21, 1994, pp. 43-58.
5. *ComputerWorld*, Vol. 41, No. 43, October 22, 2007.
6. D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, 1985, pp. 1511-1517.
7. J.C. Knight and N.G. Leveson, "A Reply to the Criticism of the Knight&Leveson Experiment,"ACM SIGSOFT Softw. Engr. Notes, 15, 1, pp. 24-35.
8. B. Littlewood and D.R. Miller, "Conceptual Modeling of Coincidental Failures in Multiversion Software," *IEEE Transactions on Software Engineering*, SE-15, 12, pp. 1596-1614, 1989.
9. Bev Littlewood, Peter Popov, and Lorenzo Strigini, "Modelling software design diversity—a review," *ACM Computing Surveys*, Vol. 33, No. 2, 2001, pp. 177-208.
10. Eric S. Raymond, "The Cathedral and the Bazaar," 1998 www.firstmonday.org/issues/issue3_3/raymond/.
11. Rosa Laura Zavala Gutierrez and Michael N. Huhns, "Multiagent-based Fault Tolerance Management for Robustness," in *Robust Intelligent Systems*, Alfons Schuster, editor, Springer, London, 2008, pp. 23-42.