Distributed Continuous Quality Assurance: The Skoll Project

Adam Porter, Cemal Yilmaz
Computer Science Department
University of Maryland, College Park
{aporter,cyilmaz}@cs.umd.edu

Douglas C. Schmidt
Electrical Engineering
& Computer Science Department
Vanderbilt University
{schmidt, bala}@dre.vanderbilt.edu

Abstract

Quality assurance (QA) tasks, such as testing, profiling, and performance evaluation, have historically been done in-house on developer-generated workloads and regression suites. Since this approach is inadequate for many systems, tools and processes are being developed to improve software quality by increasing user participation in the QA process. A limitation of these approaches is that they focus on isolated mechanisms, but not on the coordination and control policies and tools needed to make the global QA process efficient, effective, and scalable. To address these issues, we have initiated the Skoll project, which is developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality.

1. INTRODUCTION

Software testing and profiling plays a key role in software quality assurance (QA). These tasks have often been performed in-house by developers, on developer platforms, using developer-generated input workloads. One benefit of in-house QA is that programs can be analyzed at a fine level of detail since QA teams have extensive knowledge of, and unrestricted access to, the software. The shortcomings of in-house QA efforts, however, are well-known and severe, including (1) increased QA cost and schedule and (2) misleading results when the test-cases and input workload differs from actual test-cases and workloads or when the developer systems and execution environments differ from fielded systems.

In-house QA processes are particularly ineffective for performance-intensive software, such as that found in (1) high-performance computing systems (*e.g.*, those that support scientific visualization, distributed database servers, and financial transaction processing), (2) distributed real-time and embedded systems that monitor and control real-world artifacts (*e.g.*, avionics mission- and flight-control software, supervisory control and data acquisition (SCADA) systems, and automotive braking systems), and (3) the operating systems, middleware, and language processing tools that support high-performance computing systems and distributed real-time and embedded systems. Software for these types of performance-intensive systems is increasingly subject to the following trends:

- **Demand for user-specific customization.** Since performance-intensive software pushes the limits of technology, it must be optimized for particular run-time contexts and application requirements. General-purpose, one-size-fits-all software solutions often have unacceptable performance.
- **Severe cost and time-to-market pressures.** Global competition and market deregulation are shrinking budgets for the development and QA of software in-house, particularly the operating system and middleware infrastructure. Moreover, performance-intensive users are often unable or less willing to pay for specialized proprietary infrastructure software. The net effect is that fewer resources are available to devote to infrastructure software development and QA activities.
- **Distributed and evolution-oriented development processes.** Today's global IT economy and n-tier architectures often involve developers distributed across geographical locations, time zones, and even business organizations. The goal of distributed development is to reduce cycle time by having developers work simultaneously, with minimal direct inter-developer coordination. Such development processes can increase churn rates in the software base, which in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same situation occurs in evolution-oriented processes, where many small increments are routinely added to the base system.

As these trends accelerate, they present many challenges to developers of performance-intensive systems. A particularly vexing trend is the explosion of the software configuration space. To support customizations demanded by users, performance-intensive software must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or run-time. For example, performance-intensive middleware, such as web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) have dozen or hundreds of options. While this flexibility promotes customization, it creates many potential system configurations, each of which may need extensive QA to validate.

When increasing configuration space is coupled with shrinking software development resources, it becomes infeasible to handle all QA in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their software will run. Due to time-to-market driven environments, therefore, developers must often release their software in configurations that have not been subjected to extensive QA. Moreover, the combination of an enormous configuration space and severe development constraints mean that developers must make design and optimization decisions without precise knowledge of their consequences in fielded systems.

Solution approach: distributed continuous QA. The trends and associated challenges discussed above have yielded an environment in which the software systems tested and profiled by in-house developers and QA teams often differ substantially from the systems run by users. To address these challenges, we have begun a long-term, multi-site collaborative research project called Skoll.[1] This

[1]Skoll is a Scandinavian myth that explains the sunrise and sunset

paper describes

- Skoll's *distributed continuous QA process* that leverages the extensive computing resources of worldwide user communities in order to improve software qualities and provide to greater insight into the behavior and performance of fielded systems, and
- Skoll's tools and services, including its *model-driven intelligent steering agent* that controls and automates the QA process across large configuration spaces on a wide range of platforms.

2. RELATED WORK

QA tasks have traditionally been performed in-house. For the reasons described in Section 1, however, in-house QA is increasingly being augmented with in-the-field techniques [3, 1, 4, 2]. Examples range from manual and reactive techniques (such as distributing software with prepackaged installation tests and encouraging end-users to report errors when they run into problems) to automated and proactive techniques (such as online crash reporting and auto-build scoreboard systems used in many open-source projects).

Although the existing distributed QA efforts and tools help to improve the quality and performance of software, they have significant limitations. For example, since users decide (often by default) what features they will test, some configurations are tested multiple times, whereas others are never tested at all. Moreover, these approaches do not automatically adapt to or learn from the test results obtained by other users. The result is an opaque, inefficient, and *ad hoc* QA process.

To address these shortcomings, the Skoll project is developing and empirically evaluating a process, methods, and tools for around-the-world, around-the-clock QA that (1) works with highly configurable software systems, (2) uses intelligent steering mechanisms to efficiently leverage end-users resources in a QA process that adapts based on the analysis of previous results received from other sites, and (3) minimizes user effort through the judicious use of automated tools. We discuss these capabilities in the next section.

3. THE STRUCTURE AND FUNCTIONALITY OF SKOLL

As outlined in Section 1, the Skoll project is a long-term, multi-site collaborative research effort that is developing processes, methods, and tools to enable:

- **Substantial amounts of QA to be performed at fielded sites using fielded resources**, *i.e.*, rather than performing QA tasks solely in-house, Skoll pushes many of them to user sites. This approach provides developers and testers more effective access to user computing resources and provides visibility into the actual usage patterns and environment in which the fielded systems run.
- **Iterative improvement of the quality of performance-intensive software.** Skoll provides a control layer over the distributed QA process using models of the configuration space, the results of previous QA tasks, and navigation strategies that combine the two. This control layer determines which QA task to run next and on which part of the configuration space to run it. As the process executes, problems may be uncovered (and fixed) in the software, the models, and the navigation strategy.

cycles around the world.

- **Reduced human effort via judicious application of automation.** Skoll also uses the models developed in the previous step to help automate the role of human release managers, who monitor the stability of software repositories manually to ensure problems are fixed rapidly. In addition, Skoll uses automated web tools to minimize user effort and resource commitments, as well as ensure the security of user computing sites.

The approach we are taking to achieve these goals is based on a *distributed continuous QA process*, in which software quality and performance is iteratively and opportunistically improved around-the-clock in multiple geographically distributed locations. The Skoll project envisions distributed continuous QA via a geographically decentralized computing pool made up of thousands of machines provided by users, developers, and companies around the world.

The resources in the Skoll computing pool are scheduled and coordinated carefully via *data-driven feedback*. This adaptation is based on analysis of QA results from earlier testing tasks carried out in other locations, *i.e.*, Skoll follows the sun around the world and adapts its QA process continuously.

3.1 The Skoll Client/Server Architecture

To perform the distributed continuous QA process, Skoll uses a client/server architecture. Figure 1 illustrates the roles and components in this architecture, focusing primarily on the Skoll server and its interactions with various types of users. Figure 2 then shows the components in Skoll clients.

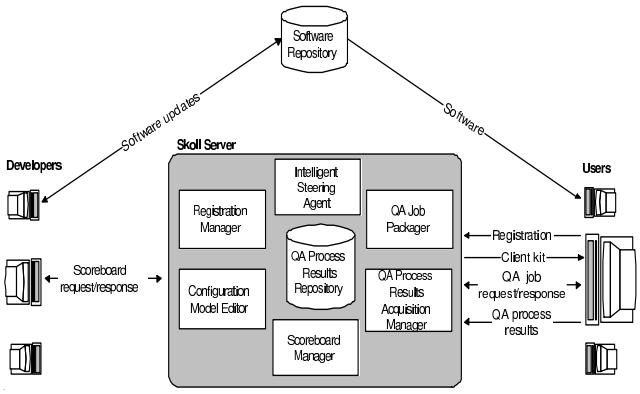

Figure 1: Components in Skoll Client/Server Architecture

User clients register with the Skoll *server registration manager* via a web-based registration form. Users characterize their client platforms (*e.g.*, the operating system, compiler, and hardware platform) from lists provided by the registration form. This information is stored in a database on the server and used by the Skoll *intelligent steering agent* (ISA). As described further in Section 3.2, the ISA automatically selects and then generates valid *job configurations*, which consists of the code artifacts, configuration parameters, build instructions, and QA tasks (e.g., regression/performance tests) associated with a software project. A job configuration also contains registration-specific information tailored for a particular client platform, along with the locations of the *CVS server* where the code artifacts actually reside.

After a registration form has been submitted and stored by the Skoll server, the *server registration manager* returns a unique ID and configuration template to the Skoll client. The template can be modified by end users who wish to restrict or specify what job configurations they will accept from the Skoll server. The Skoll

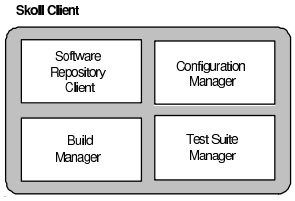

Figure 2: Skoll Client Architecture

client's architecture is shown in Figure 2. The Skoll client periodically requests job configurations from the server via HTTP POST requests. The server responds with a job configuration that has been customized by the server's *intelligent steering agent* in accordance with (1) the characteristics of the client platform, and (2) its knowledge of the valid configurations space and the results of previous QA tasks. The server maintains this information using the techniques described in Section 3.2.

The *CVS client* component is responsible for downloading software from the CVS repository. The information required to perform this task, such as the version and module name (CVS terminology) of the software, are sent in the job configuration. The *client configuration manager* component prepares the software by creating and/or customizing the appropriate header files. The instructions from the server provide mapping information between each parameter and the header file in which the parameter must be defined. The *client build manager* component builds the software by using the compiler specified at the registration time. The *client test suite manager component* is responsible for locating and executing the tests in the test suite.

For each job configuration, the Skoll client records all of its activities into a log file accessible from the Skoll server. Each log file consists of multiple sections, where each section corresponds to an operation performed by the client, such as CVS check out, build, and execute QA tasks. As the builds and tests complete, the client log files are sent to the Skoll server, which uses the *server QA process results acquisition manager* shown in Figure 1 to parse the log files and store them into a database.

Since the Skoll architecture is designed to support a user community with heterogeneous software infrastructures, developers must be able to examine the results without concern for platform compatibility and local software installation. We therefore employ web-based scoreboards that use XML to display the build and test results for job configurations. The *server scoreboard manager* provides a web-based scoreboard retrieval form to developers through which they can browse a scoreboard for a particular job configuration.

3.2 The Intelligent Steering Agent

Portions of the Skoll architecture described above are similar to those used by other distributed QA systems described in Section 2. A distinguishing feature of Skoll, however, is its use of an *intelligent steering agent* (ISA) to control the process. The ISA controls the process by deciding which configurations, in which order, to give to each incoming Skoll client request.

As QA tasks are carried out, their results are returned to the Skoll server and made available to the ISA. The ISA can therefore learn from past results, using that knowledge when generating new configurations.

The configuration model The most basic element of ISA approach is a formal model of the system's configuration space. Each software system controlled by Skoll has a set of configurable options, each with a small, discrete number of settings. Each option value must be set before the system executes. Creating a job configuration, therefore, involves mapping each option to one of its allowable settings.

Since the configuration options may take many values, the *configuration space* can be quite large. Not all possible configurations are valid, however. We define which configurations are valid by imposing *inter-option constraints* on values of options.

Planning internals. Given Skoll's formal configuration model, we can cast the configuration generation problem as a planning problem. Given an *initial state*, a *goal state*, a set of *operators*, and a set of *objects*, the ISA planner returns a set of actions (or commands) with ordering constraints to achieve the goal. In Skoll, the initial state is the default job configuration of the software. The goal state is a description of the desired configuration, partly specified by the end user. The operators encode all the constraints, including knowledge of past test executions. The resulting plan is the configuration (*i.e.*, the mapping of options to their settings).

We modified the Skoll planner so that, it can iteratively generate all acceptable plans, unlike typical planning systems that usually generate a single plan for a given set of constraints. Since the ISA generates multiple plans, we also added "navigation strategies," which are algorithms that allow us to schedule or prioritze among a set of multiple acceptable plans. This capability is useful in Skoll to cover a set of configurations, yet only proceed one step at a time in response to Skoll client requests. These algorithms also allow Skoll to add new information derived from previous QA task results to the planning process.

Planner output. The Skoll client requests a job configuration from a Skoll server. The Skoll server then queries its databases and (if provided by the user) a configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. This information is packaged as a planning goal and sent to the ISA to be solved. Using this goal, the ISA planner generates a plan that is processed by the Skoll server, which ultimately returns all instructions necessary for running the QA task on the user's platform. These instructions are called the *job configuration*.

3.3 Skoll in Action

At a high level, the Skoll process is carried out as shown in Figure 3 and described below:

1. Developers create the configuration model and navigation strategies. The ISA configuration model editor then automatically translates the model into planning operators and stores them in an ISA database. Developers also create the client kit.
2. A *user* submits a request to download the software via the registration process described earlier. The user then receives the Skoll client software and a configuration template. If users wish to temporarily change configuration settings or constrain specific options they do so by modifying the configuration template.
3. The Skoll client periodically requests a job configuration from a Skoll server.
4. In response to a client request, the Skoll server queries its databases and the user-provided configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. It then packages this information as a planning goal and queries the ISA. The ISA generates a plan and returns it to the Skoll server. Finally, the Skoll server creates the job configuration and returns it to the Skoll client.
5. The Skoll client invokes the job configuration and returns the

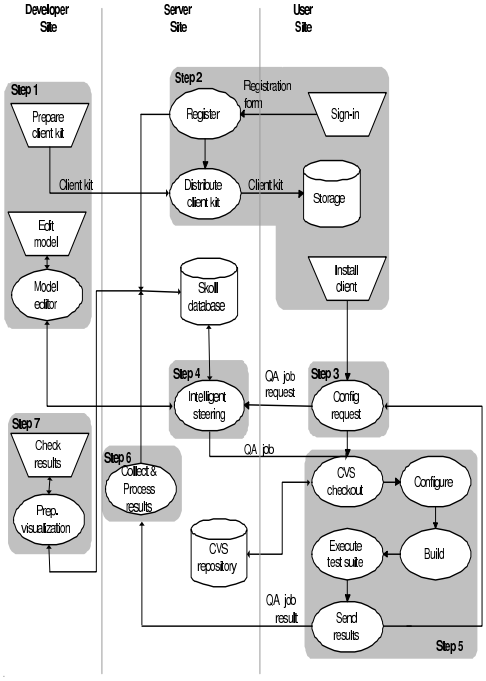

Figure 3: Process View

results to the Skoll server.

6. The Skoll server examines these results and updates the ISA operators to reflect them.
7. Periodically and when prompted by developers the Skoll server prepares a *virtual scoreboard*, which depicts all known test failures and their details. It also performs statistical analysis of the failing options and prepares visualizations that help developers quickly identify large subspaces in which tests have failed.

4. FEASIBILITY STUDIES AND FUTURE WORK

To demonstrate the benefits of Skoll, we are evaluating its impact via experiments on the ACE [5, 6] and TAO [7] open-source projects. ACE+TAO are production quality performance-intensive middleware consisting of well over one million lines of C++ code and regression tests contained in ∼4,500 files. Hundreds of developed around the world have worked on ACE+TAO for more than a decade, providing us with an ideal test-bed for our distributed continuous QA tools and processes.

The results are providing valuable insight into the benefits and limitations of the current Skoll processes. Skoll can iteratively model complex configuration spaces and use this information to perform complex testing processes, find test failures corresponding to real bugs and help developers localize the root causes of certain test failures.

Our future work is focusing on refining our hypotheses, study designs, analysis methods, and tools – repeating and enhancing experiments as necessary. We ultimately plan to involve a broad segment of the ACE+TAO open-source user community in over fifty countries worldwide to establish a large-scale distributed continuous QA test-bed.

5. REFERENCES

[1] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9. ACM Press, 2002.

[2] B. Liblit, A. Aiken, and A. X. Zheng. Distributed program sampling. In *Proceedings of PLDI'03*, San Diego, California, June 2003.

[3] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the international symposium on Software testing and analysis*, pages 65–69. ACM Press, 2002.

[4] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st international conference on Software engineering*, pages 277–284. IEEE Computer Society Press, 1999.

[5] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.

[6] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[7] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.