Autotest — Testing the Untestable

John Admanski
Google Inc.
jadmanski@google.com

Steve Howard
Google Inc.
showard@google.com

Abstract

Increased automated testing has been one of the most popular and beneficial trends in software engineering. Yet low-level systems such as the kernel and hardware have proven extremely difficult to test effectively, and as a result much kernel testing has taken place in a manual and relatively ad-hoc manner. Most existing test frameworks are designed to test higher-level software isolated from the underlying platform, which is assumed to be stable and reliable. Testing the underlying platform itself requires a completely new set of assumptions and these must be reflected in the framework's design from the ground up. The design must incorporate the machine under test as an important component of the system and must anticipate failures at any level within the kernel and hardware. Furthermore, the system must be capable of scaling to hundreds or even thousands of machines under test, enabling the simultaneous testing of many different development kernels each on a variety of hardware platforms. The system must therefore facilitate efficient sharing of machine resources among developers and handle automatic upkeep of the fleet. Finally, the system must achieve end-to-end automation to make it simple for developers to perform basic testing and incorporate their own tests with minimal effort and no knowledge of the framework's internals. At the same time, it must accommodate complex cluster-level tests and diverse, specialized testing environments within the same scheduling, execution and reporting framework.

Autotest is an open-source project that overcomes these challenges to enable large-scale, fully automated testing of low-level systems and detection of rare bugs and subtle performance regressions. Using Autotest at Google, kernel developers get per-checkin testing on a pool of hundreds of machines, and hardware test engineers can qualify thousands of new machines in a short time frame. This paper will cover the above challenges and present some of the solutions successfully employed in Autotest. It will focus on the layered system architecture and how that enables the distribution of not only the test execution environment but the entire test control system, as well as the leveraging of Python to provide simple but infinitely extensible job control and test harnesses, and the automatic system health monitoring and machine repairs used to isolate users from the management of the test bed.

1 Introduction

Autotest is a framework for fully automated testing of low-level systems, including kernels and hardware. It is designed to provide end-to-end automation for functional and performance tests against running kernels or hardware with as little manual setup as possible. This automation allows testing to be performed with less wasted effort, greater frequency, and higher consistency. It also allows tests to be easily pushed upstream to various developers, moving testing earlier into the development cycle.

Using Autotest, kernel and hardware engineers can achieve much greater test coverage than such components usually receive. This typical lack of effective low-level systems testing comes with good reason: automated testing of such systems is a difficult task and presents many challenges distinct from userspace software testing. This paper introduces the requirements Autotest aims to meet and some of the unique challenges that arise from these requirements, including robust testing in the face of system instability, scaling to thousands of test machines, and minimizing complexity of test execution and test development. The paper will discuss solutions for each of these challenges that have been employed in Autotest to achieve effective, fully automated low-level systems testing.

2 Background

High-quality automated testing is a necessity for any large, long-lived software project to maintain stability

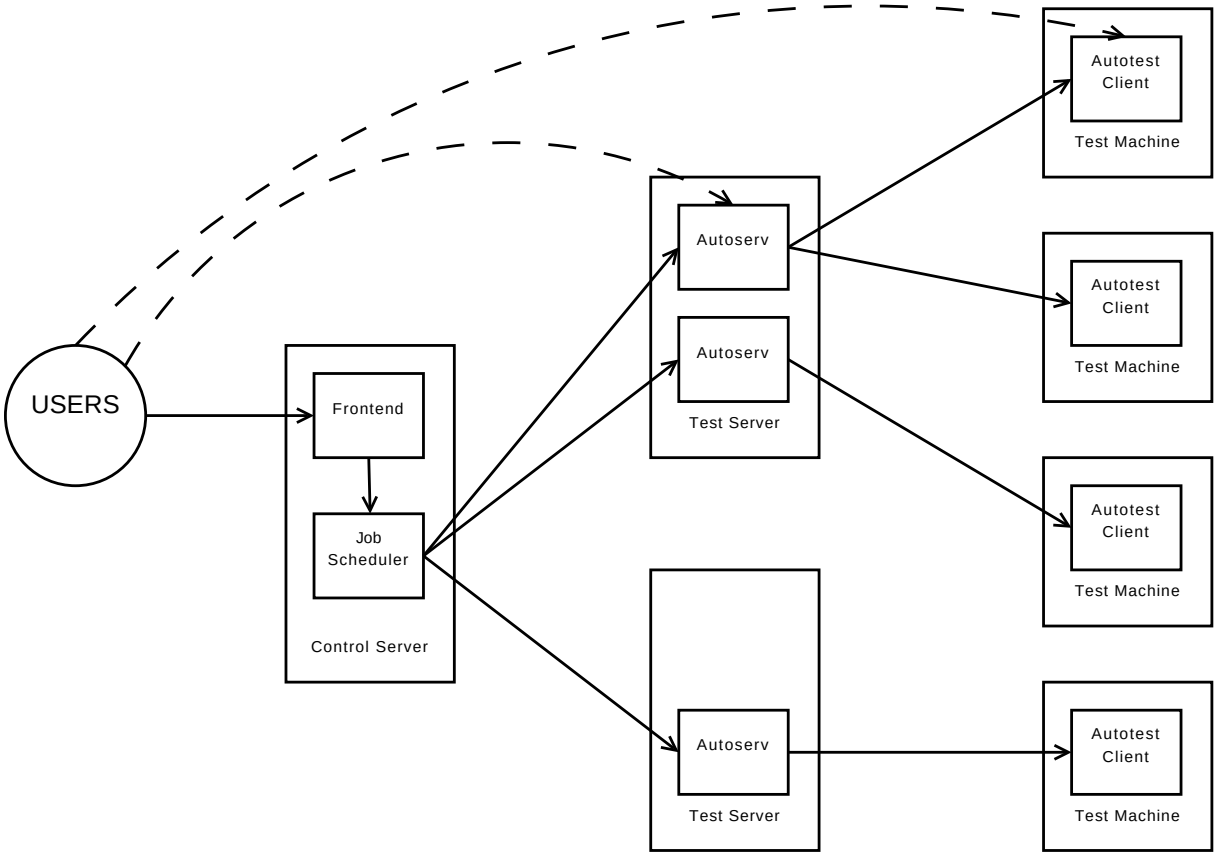

Figure 1: High level operation of a complete Autotest system

while permitting rapid development. This is as true for the Linux kernel and other system software as it is for user-space software. However, so far the benefits of automated testing have been most successfully realized within user-space applications.

Most existing test automation frameworks are targeted at software running on top of the platform provided by the hardware and operating system, the realm in which nearly all software operates. By taking advantage of the assumption that an application is running in a reliable standardized environment provided by the platform, a framework can abstract away and simplify most of the underlying system. When attempting to provide the same services for kernel (and hardware) testing, this assumption is no longer reasonable since the underlying system is an integral component of what is being tested. This was part of the original motivation for the development of the first versions of Autotest and its predecessor, IBM Autobench[5][4].

Autotest begins with the goal of testing the underlying platform itself, and this goal engenders a unique set of requirements. Firstly, because the platform on which Autotest runs is itself under test, Autotest must be built from the ground up to assume system instability. This requires graceful handling of kernel panics, hardware lockups, network failures, and other unexpected failures. In addition, tasks such as kernel installation and hardware configuration must be simple, commonplace activities in Autotest.

Secondly, because the platform under test cannot be easily virtualized, every running test requires a physical machine. Hardware virtualization may be used for basic kernel testing, but as it fails to produce accurate performance results and can mask platform-specific functional issues it is useful only for the most basic kernel functional verification. Autotest is therefore built to run every test on a physical machine, both for kernel and hardware testing. This makes coordination among multiple machines a core necessity in Autotest and furthermore implies that scaling requires distribution of testing among hundreds or even thousands of machines. This additionally creates a need for a system of efficient sharing of test machines between users to maximize utilization over such a large test fleet.

Finally, Autotest must fulfill the generic requirements of any testing framework. In particular, Autotest must minimize the overhead imposed on test developers. It must be trivial to incorporate existing tests, easy to write simple new tests, and possible to write complex multi-process or multimachine tests, all within the same basic framework. Furthermore, developing tests should be a simple, familiar process, requiring interaction with only a small subset of the available infrastructure. Tests must therefore be easily executable by hand and simultaneously pluggable into a large-scale scheduling system. These levels of abstraction are broken down into distinct modules discussed in more detail throughout this paper.

As illustrated in Figure 1, the lowest layer of the system is the Autotest client, a simple test framework that runs on individual machines. The next layer, Autoserv, is designed to run on centralized test servers to automatically install and execute clients and to coordinate multi-machine tests. The outermost layer consists of a single frontend and job scheduler to allow multiple users to share a single test fleet and results repository. Note that the dependencies go in only one direction making the design more modular and allowing users to interact with the system on multiple levels. On a large scale users can push a button on a web interface to launch a complete test suite on a large cluster of machines while on a small scale users can run a single test on a local workstation by executing a shell command.

2.1 Related work

The Linux Test Project "has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux"[1]. It is a collection of functional and stress tests for the Linux kernel and related features as well as a client infrastructure for test execution. The client infrastructure eases the execution of a many tests (there are over 3,000 tests included), supports running tests in parallel, can generate background stress during test execution, and generates a report of test results at the end of a run. LTP is not, however, intended to be a general-purpose, fully-automated kernel testing framework. There are a number of Autotest goals that are specifically non-goals of LTP[8]. It is essentially a collection of tests and is therefore suitable for inclusion into Autotest as a test, and indeed such inclusion has been easily done.

An automation framework called Xentest was developed for testing the Xen virtualization project. David Barrera *et al.* note that "testing Linux under Xen and testing Linux itself are very much alike" and perform part of their testing by "running standard test suites under Linux running on top of Xen", including LTP[3]. Since testing Xen is much like testing the underlying hardware itself the goals of Autotest share much in common with those of Xentest, both from a kernel testing and a hardware testing point of view. Xentest is a collection of scripts with support for building and booting Xen, running tests under it, and gathering results logs together. It does not support any automated analysis of test results to determine pass/fail conditions. Test runs are configurable by a control file using the Python ConfigParser module. This provides simple configuration but lacks any programmatic power within control files. Finally, Xentest is built closely around Xen and does not aim to be generic framework for kernel or hardware testing. On the other hand, Autotest could be used to perform Xen testing much like Xentest does and some work has been done on this in the past.

Crackerjack is another test automation system, one designed specifically for regression testing[10]. It focuses on finding incompatible API changes between kernel versions. This is valuable testing but is a narrower focus from that of Autotest.

Two frameworks that address the problem of distributed kernel testing are PyReT[6] and ANTS[2]. The former depends on a shared file system for all communications while the latter uses a serial console. Both of these requirements on test machines were deemed too restrictive for Autotest, which relies solely on an SSH connection for communications. ANTS is quite robust to test machine failures, as it configures all test machines from scratch using network booting and is capable of using remote power control to reset and recover machines that have become unresponsive. The system additionally includes a machine reservation tool so that machines can be shared between developers and the automated system without conflict. These are all important features that have found their way into Autotest. However, the system is built strictly for nightly testing and does not support a general queue of user-customizable jobs. It includes very limited results analysis in the form of an email report upon completion of the night's tests. It runs a number of open-source tests (including LTP) but does not support more complex, multimachine tests. Finally, the system is proprietary and therefore of little direct

utility to the community.

For distributed performance testing of the kernel there exist systems presented by Alexander Ufimtsev[9] and Tim Chen[7]. In both systems, test machines operate autonomously, running a client harness which monitors the kernel repository, building and testing new releases as they appear. In this sense, the systems are built around the specific purpose of per-release testing, although the latter system includes support for testing arbitrary patches on any kernel. Both systems' clients transmit results to a central repository, a remote server in the former case and a shared database in the latter. The former system includes some automated analysis for regression detection based on differences from previous averages, a task not yet implemented in Autotest. The latter system includes a web frontend displaying graphs of each benchmark over kernel versions, with support for displaying profiler information, rerunning tests or bisecting to find the patch responsible for a regression. Autotest includes partial support for these features but could benefit from improvements in this area.

3 Autotest Client

The most basic requirement that Autotest is intended to fulfill is to provide an environment for running tests on a machine in a way that meets the following criteria:

1. The lowest, most bare-metal access must be available.
2. Test results are available in a standard machine-parseable way.
3. Standard tests developed outside of the framework can be easily run within it.

The first of the criteria, low-level system access, seems fairly self-evident when writing tests which are aimed at the kernel and the hardware itself. To test a particular component of a system, the test must be written using tools that have access the standard API for that component. Since C is the lingua franca of the systems world, a C API can generally be counted on as being available, but even that isn't always the case. When creating a file system during a test, `mkfs` is going to be the easiest and most readily available mechanism; so as well as being able to easily incorporate custom C the framework must also make it easy to work with external tools.

This initial requirement could have been satisfied by writing the framework itself in C, but that would ultimately have conflicted with the other requirements that Autotest was expected to meet. First, this would've made calling out to external applications ultimately more difficult; while functions like `fork`, `exec`, `popen` and `system` provide all the basic mechanisms needed to launch an external process and collect results from it, working with them in C requires a relatively large amount of boilerplate compared to a higher-level scripting language such as Perl or Python. This only becomes more true if the output of the executed process needs to be manipulated and/or parsed in any way. The second requirement that test results be logged in a standard way almost guarantees that the test will need to do string manipulation, another task simplified by using a scripting language.

To meet these somewhat conflicting requirements, the Autotest framework itself was written in Python, with utilities provided to simplify the compilation and execution of C code. Tests themselves are implemented by creating a Python module defining a test subclass, satisfying a standardized, pre-defined interface. Individual tests are packaged up in a directory and can be bundled along with whatever additional resources are needed, such as data files, C code to be compiled and executed or even pre-compiled binaries if necessary.

This also satisfies the third of the three requirements, the ability to run standard tests written independently of Autotest. All that is required is to bundle the components necessary for the test with a simple Python wrapper. The wrapper is responsible for setting up any necessary environment, executing the underlying test, and translating the results from the form produced by the test into Autotest standard logging calls. The wrappers are generally quite simple; the median size of a test wrapper in the current Autotest distribution is only 38 lines.

Using Python for implementing tests also provides an easy mechanism for bundling up suites of tests or customizing the execution of specific tests. Tests themselves are executed by writing a "control file" which is simply a Python script executing in a predefined environment. It can be a single line saying "execute this test", a more complex script that executes a whole sequence of tests, or even a script that conditionally executes tests depending on what hardware and kernel are running on the machine. The environment provided by Autotest contains additional utilities that allow control

files to put the machine into any state necessary for executing tests, even if it requires installing a kernel and rebooting the machine. Having the full power of Python available allows test runners to perform limitless customization without having to learn a custom job control language.

This power does come with one major drawback, though. Due to the dynamic nature of Python and the power available to control files, it is impossible to statically determine much information about a job. For example, it is impossible to know in advance what tests a job will run, and indeed the set of tests run may potentially be nondeterministic. This limitation has not been severe enough to outweigh the benefits of this approach.

3.1 Installation Problems

As this system was put into use at Google, the installation of Autotest onto test machines quickly became a serious performance issue. Allowing test developers to bundle data, source code and even binaries with their tests made it easy to write tests but allowed the installation size to grow dramatically. The situation could be somewhat alleviated by minimizing how often an install was necessary, but in practice this only helps if the test framework can be pre-installed on the systems.

The solution to this problem is a fairly standard one: rather than treating Autotest and its test suite as a single, monolithic package, break it up into a set of packages:

- a core package containing the framework itself
- packages for the various utilities and dependencies such as profilers, compilers and any non-standard system utilities that would need to be installed
- packages for the individual tests

Each package is able to declare other packages as dependencies. The core package can be installed everywhere and is fairly lightweight, consisting only of a set of Python source files without any of the more heavyweight data and binaries required by some tests. When executing a job, the framework is then able to dynamically download and install any packages needed to execute a specific test.

4 Autotest Server

4.1 Distributing test runs across machines

The Autotest client provides sufficient infrastructure for running low-level tests but it only executes tests and collects results on a single machine. To test a kernel on multiple hardware configurations, a tester would need to install the test client on multiple machines, manually run jobs on each of these machines, and examine the results scattered across these systems.

This deficiency led to the development of Autoserv, an Autotest Server, a separate layer designed around the client. It allows a user to run a test by executing a server process on a machine other than the test machine. The server process will connect to the remote test machine via SSH, install an Autotest client, run a job on the client, and then pull the results back from the test machine. Localizing these server runs to a single machine allows users to run test jobs on arbitrary sets of machines while collecting all the results into a central location for analysis.

4.2 Recovering failed test systems

Once users start running tests on larger sets of machines, dealing with crashed systems becomes a much more common occurrence. As the number of test machines increases, bad kernels (and random chance) are going to result in more failed systems. When testing on a single machine, manual intervention is the simplest method of dealing with failure, but this does not scale to hundreds or thousands of machines. Automation becomes necessary with two major requirements:

- Automatically detect and report on test machine failures
- Provide a mechanism for repairing broken systems

Handling these requirements entirely within the client running on the test machine is impractical; detecting and reporting a kernel panic or hardware failure will not even be possible when the crash kills the test processes on the machine. Similarly, repair may require re-imaging a machine which will wipe out the client itself.

With job execution controlled from a remote machine, handling these requirements becomes feasible. Autoserv implements support for monitoring serial console output, network console output and general syslog output in `/var/log`. It can also interact with external services that collect crash dumps and even power cycle the machine if that capability is available. In the very worst case the server process can at least clearly log the failure of the job (and any tests it was running) along with the last known state of the failed test machine.

Automated repair can also be performed. This is implemented in Autoserv in an escalating fashion, first by making several attempts to put the machine back into a known good state, then by optionally calling out to any local infrastructure in place to carry out a complete reinstallation of the machine, and finally, if necessary, by escalating the repair process to a human. Testing on large numbers of machines now becomes much more practical when systems broken by bad kernels (or bad tests) can be put back into a working state with a minimum of human intervention.

4.3 Multi-machine tests

Remote control of test execution also introduces the opportunity to run single tests that span multiple machines. While this could be done with the Autotest client alone by running the client on a master test system and having it drive other slave test systems, this would require duplicating most of the "remote control" infrastructure from the server directly into the client. This could also be problematic from a security point of view since, rather than routing control through a single server, the test machines would require much more liberal access to one another.

Since Autotest already established the need for a separate server mechanism, it was natural to extend it to support "server-side" testing. Instead of only providing a fixed set of server operations (install client and run job, repair, etc.), Autoserv allows testers to supply a Python control file for execution on the server, just like on the client. This can be used to implement, for example, a network test with the following flow:

- Install Autotest client on two machines
- Launch "network server" job on one machine
- Launch "network client" job on one machine
- Wait for both jobs to complete and collect results

No single-machine networking test can duplicate the same results, particularly when attempting to quantify networking performance and not just test the stability of the network stack.

This also allows for execution of larger-scale cluster testing. Although this begins to creep beyond the scope of systems testing it still has significant value, not as a way to test the cluster applications but rather as a way of testing the impact of kernel and hardware changes on larger-scale applications. A smaller-scale cluster test can follow a workflow similar to that for network testing. Alternatively, a server job can make use of pre-existing cluster setup and management tools, simply driving the external services and collecting results afterwards.

4.4 Mitigating Network Unreliability

While one of the primary goals of Autoserv is to increase reliability, it also introduces new unreliabilities as an unfortunate side effect. The primary issue is that it introduces a new point of failure, the connection between the server and the client machines. Working directly with the client, a user can launch a job on a machine and return after expected completion, and any transient network issues will not affect the test result. This is no longer the case when the job is being controlled by a remote server that continuously monitors the test machine. The problem can be alleviated somewhat by periodically polling the remote machine rather than continually monitoring it, but ultimately this only reduces susceptibility to the problem.

Implementing more reliable communications over OpenSSH ultimately proved too difficult, primarily due to the lack of control over and visibility into network failure modes. One alternative considered was to use a completely separate communication mechanism, but this was rejected as impractical. Using SSH provides Autotest with a robust and secure mechanism for communication and remote execution, without requiring the large investment of time and labor required to invent a custom protocol that would then need to be installed on every test machine.

Instead the solution was to add an alternative SSH implementation that uses a Python package (paramiko[1])

[1] `http://www.lag.net/paramiko/`

instead of launching an external OpenSSH process. Using an in-process library allowed tighter integration and communication between Autoserv and the SSH implementation, allowing the use of long-lived SSH connections with automatic recovery from network failure. At the same time modifications were made to the Autotest client to allow it to be run as a detachable daemon so that the automatic connection recovery could re-attach to clients with no impact on the local testing.

Adding paramiko support had the additional benefit of reducing the overhead of executing SSH operations from Autoserv by performing them in-process, as well as simplifying the use of multi-channel SSH sessions to avoid the cost of continually creating and terminating new sessions. Within Autoserv this is implemented in such a way that the paramiko-based implementation can be used as a drop-in replacement for the OpenSSH-based one, allowing testers to make use of whichever is better suited to their needs. OpenSSH works better "out of the box" with most Linux configurations, while paramiko, which requires more setup and configuration, ultimately allows for more reliable, lightweight connections.

5 Scheduler and Frontend

5.1 Shared machine pool

Autoserv provides a convenient and reliable way for individual users to test small numbers of platforms. As a standalone application, however, it cannot possibly fulfill the requirement of scaling to thousands of machine and achieving efficient utilization of a shared machine pool. To address these needs the Autotest service architecture provides a layer on top of Autoserv that allows Autotest to operate as a shared service rather than a standalone application. Rather than execute the Autotest client or server directly, users interact with a central service instance through a web- or command-line-based interface. The service maintains a shared machine pool and a global queue of test jobs requested by users. There are three major components that make this usage model possible. The Autotest Frontend is an interface for users to schedule and monitor test jobs and manage the machine pool. The Autotest Scheduler is responsible for executing and monitoring Autoserv to run tests on machines in the pool in response to user requests. Finally, the results analysis interface, not discussed in this paper, provides a common interface to view, aggregate and analyze test results.

The Autotest Frontend is a web application for scheduling tests, monitoring ongoing testing, and managing test machines. It operates on a database which takes the available tests, the machines in the shared test bed, and the global queue of test jobs that have been scheduled by users. The scheduler interacts with the frontend through this database, executing test jobs that have been scheduled and updating the statuses of jobs and machines based on execution progress.

The frontend supports a number of features to help users organize the machine pool. First, the system supports access control lists to restrict the set of users that can run tests on certain machines. Some machines may be open for general testing, but some users, particularly hardware testers, will have dedicated machines that cannot be used by others. Second, the system supports tagging of machines with arbitrary labels. The most common usage of this feature is to mark the platform of a machine, which is often important for both job scheduling and results analysis. Labels can additionally be used to declare machine capabilities, such as remote power control, or to group together large numbers of machines for easier scheduling.

The scheduler is a daemon running on the server whose primary purpose is to execute and monitor Autoserv processes. The scheduler continuously matches up scheduled test jobs with available machines, launches Autoserv processes to execute these jobs, and monitors these processes to completion. It updates the database with the status of each job throughout execution, allowing the user to track job progress. Upon completion, the scheduler executes a parser to read Autoserv's structured results logs into a database of test results. The user can then perform powerful analysis of these results through a special results analysis interface.

An important feature of the scheduler is its statelessness. While it maintains plenty of in-memory state, all important state can be reconstructed from the database. This is exactly what happens upon scheduler startup, ensuring that when the scheduler needs to restart, all tests will continue running uninterrupted and machine time won't be wasted. This is critical for minimizing user impact during deployments of new Autotest versions or after a scheduler crash.

In addition, as the test fleet scales to thousands of machines, automated fleet health management becomes critical. To this end, the scheduler takes advantage of Autoserv's machine diagnosis and repair functionality. The scheduler launches special Autoserv processes to verify machine health before each job and perform repairs as necessary. Machines that cannot be repaired are marked as such in the database, from which a machine health dashboard can read and summarize machine health data. Additionally, the scheduler performs periodic reverification of known dead machines to catch any manual repairs that may have occurred.

5.2 Distributed execution for scalability

When all Autoserv processes are running on a single server, serious performance degradation tends to set in around 1,000 simultaneous machines under test. The scheduler supports global throttling of running processes to avoid bringing the system to a halt, but this still leaves a scalability limit imposed by the hardware itself. To alleviate the problem and allow for further scaling, the scheduler supports distributing Autoserv processes among a pool of servers.

A single scheduler coordinates execution among multiple servers and all results are centralized on a single archive server after execution completes. Each server can support roughly 1,000 machines under test, and to date no Autotest installation has reached a limit on the number of servers that can be utilized in the system. In addition to increasing scalability, distributed execution increases system reliability. Since execution servers are completely independent of each other, each can fail completely without bringing the entire service to a halt. With this distributed execution model, the Autotest service at Google has scaled to approximately 5,000 simultaneous machines under test.

5.3 Automatic generation of control files

To run a single test, users of Autoserv can run one of the existing control files written for each test. However, in order to run multiple tests within a single execution the user must write a custom control file. While control files have been kept as simple as possible, writing a custom control file still presents a major barrier to entry for new users. To this end, the Autotest Frontend simplifies the process of running multiple tests by support automatic generation of control files.

Creating a job through the frontend consists of selecting a number of tests, a number of machines, and a variety of job options. The user can select tests from a list, which includes a description of each test, and the frontend will automatically generate a control file to run the selected tests. Users may also specify a kernel to install and select profilers to enable during testing and the generated control file will incorporate all of these options. This allows users to run moderately complex jobs through Autotest with ease, without requiring any knowledge of control files. Machines can be similarly selected from a list, either one-by-one or in bulk based on filtering by hostname or platform (or any other machine label). Furthermore, users may request that the job run on any machine of a particular platform and allow the scheduler to select one at run time. This feature helps increase utilization of shared test machines and makes it particularly easy to run automated jobs without a static, dedicated set of machines.

5.4 Support for high-level automation

The bulk of the work for the web frontend is performed on the web server, which operates primarily as an RPC server. It is written in Python using the Django[2] web framework and communicates with a MySQL[3] database. The web interface is a fully-fledged application running in the browser implemented using Google Web Toolkit[4]. It communicates with the server solely through the RPC interface. There is also a command-line interface, implemented in Python, which communicates with the server through the same RPC interface. This is made possible by the use of the lightweight JSON[5] data-interchange format which is easily implemented in either language. Furthermore, custom scripts can be written that access the RPC interface directly, providing the full capabilities of the web frontend through a simple interface. This supports powerful and easy high-level automation, allowing users to extend the functionality of Autotest with external scripts layered on top of the frontend.

[2] `http://www.djangoproject.com/`
[3] `http://www.mysql.com/`
[4] `http://code.google.com/webtoolkit/`
[5] `http://www.json.org`

6 Future Directions

Autotest has made great strides in automating the execution of kernel and hardware tests. But test execution usually occurs in the context of a qualification process, and the full qualification process remains a tedious and rather mechanical ordeal. Qualifying a new kernel generally involves running a collection of functional and performances tests over a large population of machines representing a range of hardware platforms. The choice of tests to execute may be dependent on the outcome of earlier tests. The results must then be compared to those for a known stable kernel to find statistically significant deviations. Furthermore, a continuous testing system would like to execute this entire process in a fully-automated fashion, reporting deviations on a per-change basis. Qualifying a collection of new machines involves a similar, but not identical, process. In particular, individual machines will must be tracked through a cycle of testing, triaging, and repairing by either updating system software or manipulating hardware components. At the same time, this individualized tracking must scale to hundreds or thousands of machines, and the process must culminate in a report of significant deviations from a known stable platform.

While Autotest abstracts away many of the low-level issues involved in these processes, it does little to automate these higher-level processes. Successful automation of such processes is one of the major unsolved problems for the Autotest project. Fortunately, the high-level automation support provided by the frontend makes it possible to prototype solutions to these problems. Such solutions can be built on top of the Autotest architecture without requiring modifications to Autotest itself, and indeed a number of such solutions have been built to satisfy needs of particular Autotest users. These prototypes provide a useful path forward to incorporate such automation into the Autotest system.

In addition, improved reporting remains an area of great opportunity for Autotest. Autotest's current reporting interface can generate a variety of reports, potentially spanning multiple jobs, but it still requires a significant manual effort to draw useful high-level conclusions and it still makes triage of failures a difficult task. To aid the former task, Autotest needs to support better automated folding of larger amounts of data into smaller, more concise reports which highlight significant quality deviations and hide the rest of the data. For easier triaging of failures, Autotest needs to better categorize and organize test output and more efficiently guide users to the places where failure details are most likely to be found.

7 Conclusion

A significant amount of developer time has been invested in Autotest to enable the continuous execution of small- and large-scale tests on thousands of machines. This effort has successfully overcome numerous problems with reliability and scalability inherent in testing low-level systems such as the kernel and hardware components. While further work remains to be done to improve and automate the high-level testing workflow, the fundamental components are in place and already usable for large-scale testing today.

Acknowledgements

We would like to thank Martin Bligh for his input to and his reviews of drafts of this paper.

Legal Statement

References

[1] Linux Test Project. `http://ltp.sourceforge.net`.

[2] Jason Baietto. Linux Quality Assurance Utilizing An Automated Nightly Test System. `http://www.ccur.com/isddocs/ANTS.pdf`.

[3] David Berrera, Li Ge, Stephanie Glass, and Paul Larson. Testing the Xen Hypervisor and Linux Virtual Machines. In *Linux Symposium*, volume 1, pages 271–288, 2005.

[4] Kamalesh Bibulal and Balbir Singh. Keeping the Linux Kernel Honest. In *Linux Symposium*, volume 1, pages 19–29, 2008.

[5] Martin Bligh and Andy P. Whitcroft. Fully Automated Testing of the Linux Kernel. In *Linux Symposium*, volume 1, pages 113–125, 2006.

[6] Aaron Bowen, Paul Fox, James M. Kenefick Jr., Ashton Romney, Jason Ruesch, Jeremy Wilde, and Justin Wilson. Automated Regression Hunting. In *Linux Symposium*, volume 2, pages 27–35, 2006.

[7] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. Keeping Kernel Performance from Regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[8] Subrata Modak and Balbir Singh. Building a Robust Linux kernel piggybacking The Linux Test Project. In *Linux Symposium*, volume 2, pages 91–100, 2008.

[9] Alexander Ufimtsev and Liam Murphy. Automatic System for Linux Kernel Performance Testing. In *Linux Symposium*, volume 2, pages 403–408, 2006.

[10] Hiro Yoshioka. Regression Test Framework and Kernel Execution Coverage. In *Linux Symposium*, volume 2, pages 285–296, 2007.