Automated Deploy for Chrome OS Testing

Michael Checca, Bohao Li, Yilan Liu

March 2014

A Major Qualifying Project Report:
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Michael Checca

Bohao Li

Yilan Liu

Date: March 2014

Approved:

Professor David Finkel, Advisor

This report represents the work of one or more WPI undergraduate students.
Submitted to the faculty as evidence of completion of a degree requirement.
WPI routinely publishes these reports on its web site without editorial or peer review

Abstract

The current NVIDIA Chrome OS team is rather small, and they simply do not have that many man-hours to devote to testing Chromebooks. To alleviate this pressure, our MQP group has developed a system to automatically deploy and test the newest builds of NVIDIA's Chrome OS software on their new Tegra-based Chromebooks. Our software system allows the NVIDIA Chrome OS team to quickly test new nightly builds, eliminate broken builds, and ensure their development Chromebooks are always in a bootable state.

Acknowledgments

We would like to thank Professor David Finkel for his hard work in establishing the Silicon Valley project center, allowing us to work with NVIDIA in Santa Clara. He has worked with us through our PQP to ensure that we would know what to expect in California and at NVIDIA. He guided us through our research with the information he was given about the details of our project. While at the project center, he met with us to make sure we were providing NVIDIA with the quality of work that is to be expected from WPI students.

We would also like to thank NVIDIA for sponsoring our project. They proposed a project that fit a genuine need in their system, and provided us with the opportunity to build a deployable system to solve their problem. A special thank you goes to Andrew Chew, our mentor at NVIDIA. He helped focus our project to capture all of the requirements needed by NVIDIA. Andrew is very knowledgeable about Chrome OS and has been tremendously helpful in getting us acclimated with the Chrome OS and NVIDIA development environments, which allowed us more time to work on the project. Allen Martin has been the liaison connecting Professor Finkel and WPI with NVIDIA. We would like to thank him for encouraging NVIDIA to provide WPI students with projects.

We would be remiss if we did not acknowledge the WPI IGSD (Interdisciplinary and Global Studies Division). Without them, we would not have a project center to come to. They did a great job at preparing us for what to expect in an off-campus MQP. Their desire in helping students succeed in off-campus projects is much appreciated.

List of Figures

1 Chrome OS Booting and Recovery 6
2 Deploy Flow 7
3 bashcov Code Coverage Output 19
4 shUnit2 Test 20
5 Update Script Runtimes 21
6 Load Balancing 23

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures iv

Table of Contents v

1 Background 1

1.1 NVIDIA . . . 1
1.2 Tegra . . . 1
1.3 History of Chrome OS . . . 1
1.4 Chrome vs Chromium . . . 2
1.5 Chromium OS Development . . . 2
1.6 Chromium OS Image Deployment . . . 3

2 Problem Statement 4

3 Procedure 6

3.1 Hardware Entities . . . 7
3.1.1 Build Repository . . . 8
3.1.2 Build Server . . . 8
3.1.3 Test Host . . . 8
3.1.4 Debug Board . . . 8
3.1.5 Target . . . 8
3.1.6 Recovery Media . . . 9
3.2 Software Tools . . . 9
3.2.1 Bash . . . 9
3.2.2 Python . . . 9
3.2.3 Minicom . . . 10
3.2.4 Expect . . . 10
3.2.5 Dev Server . . . 10
3.2.6 pm342 Control Tool . . . 10
3.2.7 cron . . . 10
3.2.8 Run Remote Tests . . . 11
3.2.9 sSMTP . . . 11
3.3 Workflow . . . 11
3.3.1 Pull Build . . . 11
3.3.2 Start Dev Server . . . 12
3.3.3 Flash Bootloader . . . 12

3.3.4 Load Latest Image . . . 13
3.3.5 Run Tests . . . 13
3.3.6 Reboot into Recovery . . . 13

4 Design Considerations **15**
4.1 Speed . . . 15
4.2 Reliability . . . 15
4.3 User-Friendly . . . 16
4.4 Accountability . . . 16
4.5 Correctness . . . 16
4.6 Lightweight . . . 17

5 Testing and Code Coverage **18**
5.1 Tools . . . 18
5.1.1 bashcov . . . 19
5.1.2 shUnit2 . . . 20

6 Results **21**
6.1 Time Benefits . . . 21
6.2 Cost Benefits . . . 22
6.3 Accuracy . . . 22

7 Future Work **23**
7.1 Scalability . . . 23
7.2 Other Projects . . . 24
7.3 Missing Features . . . 24
7.3.1 Web Access to Test Reports . . . 24
7.3.2 Automatically Update Recovery Image . . . 24
7.3.3 Inject L4T (Linux for Tegra) . . . 24

8 Glossary **26**

9 References **27**

1 Background

1.1 NVIDIA

NVIDIA is one of the world's most recognized names in visual computing. The company was co-founded in 1993 by Jen-Hsun Huang and Chris A. Malachowsky[1]. From its inception, NVIDIA has partnered with numerous hardware companies to develop the latest and greatest in graphics technology. Throughout its career, NVIDIA has developed some industry-moving technologies. In perhaps its most well-known and profitable market, NVIDIA commercialized the graphics processing unit (GPU) in 1999. For PCI graphics cards, NVIDIA introduced SLI (Scalable Link Interface) as a way to allow multiple graphics cards to be linked together to improve processing and rendering. With GPUs becoming the more powerful, NVIDIA released the CUDA™ parallel computing platform. CUDA™ allows developers to send C, C++, and/or Fortran code straight to the GPU, completely bypassing any assembly language[2]. This has allowed for a major increase in performance for applications that require large data processing. In the mobile market, NVIDIA is credited with releasing the first dual-core mobile processor, the Tegra 2, in 2011, and with the world's fastest quad-core mobile processor, the Tegra 4, in 2013. Today, NVIDIA employs 8,800 employees worldwide and holds around 7,000 patent assets[3]. In the 2013 fiscal year, NVIDIA brought in $4.3 billion in revenue under the public NASDAQ symbol `NVDA`, first registered in 1999 in the company's IPO (Initial Public Offering)[3].

1.2 Tegra

The Tegra chipset was released by NVIDIA on June 2, 2008[4], starting with the first products Tegra 600 and 650. It is a system on a chip series designed and developed by NVIDIA that especially targets mobile devices, such as smartphones and tablets. After Tegra 600 and 650, NVIDIA subsequently developed Tegra 2, Tegra 3, and Tegra 4, each with even more computing power than its predecessor. Tegra 3 is the world's first quad-core chip designed specifically for mobile devices[5]. With a GPU with up to 72 custom cores and ARM's most advanced CPU cores, Tegra 4 is undoubtedly one of the world's most powerful chipset for mobile devices[6]. Many mobile hardware developers have chosen NVIDIA's Tegra 4 as the chipset for their devices, for example Asus, Acer, HP, and HTC.

It was first confirmed in 2009 that NVIDIA was working on getting Google Chrome OS devices running on NVIDIA's Tegra platform[7]. In 2011, Google launched a Tegra 2 powered Chrome OS tablet named Seaboard[8]. Google's newest Chromebook project, with the code-name "puppy", is said to use Tegra 4 as its chipset[9].

1.3 History of Chrome OS

Google Chrome OS is a lightweight operating system designed to run applications on the web through the Google Chrome web browser. The exact history of Chrome OS is unknown and highly debated[10]. Jeff Nelson, a former Google engineer, claims to have created a new operating system codenamed "Google OS". He even blogged about it under the title "Inventing Chromebook"[11]. However, many Google employees disagree with this claim, saying Nelson had nothing to do with the development of Google Chrome OS.

Nelson's work was done prior to the release of Chrome as a web browser, and was a stripped-down version of Linux that included Mozilla Firefox as its browser. In Google's official unveiling of the project on July 7th, 2009, Sundar Pichai, VP Product Management, proclaimed that Chrome OS would be "for people who live on the web"[12].

Chrome OS was originally developed with Canonical Ltd., the company behind Ubuntu[13]. The project was open sourced as Chromium OS in November of 2009. Although originally having started out with a Ubuntu base, the team decided a switch to Gentoo Linux was a better choice. In 2010, the switch was finalized and Chrome OS was now running on top of Gentoo. The rationale for doing this was mainly rooted in the portage package manager, which automates much of the process of compiling from source code. The Gentoo operating system has long been known for its distribution as a mostly source code operating system. This means that anyone wanting to use it needs to compile the OS from source. A typical Gentoo compilation of the whole operating system can take about 24 hours. However, developers are generally the only ones who compile the entire OS from source. From a developer standpoint, the portage package manager and ebuild system makes compiling the operating system easy for developers.

1.4 Chrome vs Chromium

Chrome OS and Chromium OS are both similar projects that are supported by Google, but are intended for different audiences. Chromium OS is usually only used by developers of the operating system, while Chrome OS is intended to be used by the end-user. From a user standpoint, the experience you get with them is the same, aside from some color and logo changes. From a technical standpoint, Chromium OS is the base for Chrome OS. Chrome OS is built upon the source code of Chromium OS, but Chrome OS also includes proprietary software from manufacturers for specific Chromebooks. According to the Chromium FAQ, "Chromium OS is the open source project, used primarily by developers, with code that is available for anyone to checkout, modify, and build", while "Google Chrome OS is the Google product that OEMs ship on Chromebooks for general consumer use"[14]. In this document, we will use Chromium OS (Chromium) to refer to the source code and development practices. We will use Chrome OS (Chrome) to refer to the commercialized final product delivered to consumers. We will not be referring to the Chromium/Chrome browsers except where explicitly stated.

1.5 Chromium OS Development

Since Chromium is the open source version of Chrome, anyone is allowed to contribute to the project, whether it be documentation, testing, writing new code, or fixing bugs. Google has a set of their own development tools (called depot_tools[1]) to manage local code repositories and to aid in the code review process. These tools don't replace git or SVN for version control; they merely add extra functionality to help promote the Chromium development workflow. Google also uses a web-based code review tool called Gerrit[2] to approve/deny changes. Anyone can submit a change for review by pushing their code to a special branch where it is held until further review. Upon being pushed, Gerrit will triage the change in its own branch and create a review page for it. Although anyone can submit and view changes, only Chromium/Google

[1] `http://www.chromium.org/developers/how-tos/depottools`

[2] Gerrit is a Google developed fork of the Rietveld SVN review tool. The Rietveld author (Python creator Guido van Rossum) named it after Gerrit Rietveld, one of his favorite Dutch architects.

approved contributors can approve or deny these changes[15].

All of Chromium is built within a chrooted environment. A chroot is an entire Linux system that resides within an existing system. This is done so as to ensure that developer environments are similar, and to prevent interference from, or accidental destruction of, the host system. The build process is designed such that separate image must be compiled for each target device. This is necessary because different devices can be on different architectures. Most of the Chromebooks on the market today are running Intel Celeron processors running x86 architecture. Samsung has developed an ARM chip called "Exynos 5" which is currently in Samsung's own Chromebook and the HP 11 Chromebook. NVIDIA has plans to put the Tegra SoC into Chromebooks as well. To accommodate these different architectures, developers can select which image they build depending on the target they are installing on. Chromium images built within the chroot can be booted from via USB, transformed into virtual machine images, or run directly on non-Chromebook machines.

1.6 Chromium OS Image Deployment

Chromium OS, being based on Linux, makes it very easy to install an existing image by putting it on a USB drive and booting from it. Placing an operating system image on external media and booting the OS from it is called a "Live CD" or "Live USB". This allows developers to quickly run an operating system to test for bugs if they have no intention of actually installing the image, allowing them to bypass the normal installation process. Most live images, including the images Chromium produces, have the ability to install to disk once it has been loaded as a live image. This provides an alternative method for installing, as opposed to the traditional method of going through an install process, such as used by the Windows operating system.

In addition to booting from a USB or CD, many modern computers have the ability to boot over the network, using a technology called PXE boot (Preboot eXecution Environment). In this scenario, an image is downloaded from the local network, which is then used to boot the system. Typically, this is used to download a small program that installs an operating system by downloading the necessary requirements over the network. However, it can also be used to load an entire Linux image and boot to a live environment, as if a CD or USB was used. This approach, unlike the USB/CD approach, requires no physical access to the machine and can be completely automated through software. Unfortunately, Chromium OS only allows booting through the local storage and USB recovery media, not from the network. This has been done deliberately to prevent potential attacks due to the nature of this "always connected" device.

While Chromebooks cannot boot from the network, they can download updates from the network while the system is running. Google has developed the Dev Server, which is a Python program that serves updated images over the network to Chromebooks. Developers have high flexibility with this feature, and can select which pieces of software should be updated. This allows developers to update as much or as little as they want, depending on the changes they've made. Using these tools, a PXE network boot environment can be easily emulated by updating the image when the Chromebook is powered on. This is a valuable tool for beginning to automate the deployment process. Specifically, this is a way to do a "hands-off" update of the operating system where no physical interaction is necessary, except for some way to send the update command (`update_engine_client`). Once the update has been initiated, Chrome OS takes care of updating the file system and rebooting if necessary. No questions are asked during the process, and failures result in a "rolling back" of the update. These features drastically improve the reliability of Chrome OS updates.

2 Problem Statement

NVIDIA has established themselves as one one of the most recognized names in graphics processing and computing. They have been expanding their horizons by bringing their technology to the mobile community. Since NVIDIA is still fairly new to the mobile industry, their team sizes are far less impressive than that of the desktop and laptop graphics teams. With such smaller teams, it is urgent that each team makes efficient use of its resources. In the case of the Chrome OS integration team, this would include having an infrastructure to quickly run smoke tests (basic sanity checks) on every nightly build of the operating system. Having such infrastructure in place reduces the need for a dedicated QA engineer, allowing the developers to do their own testing. This allows the Chrome OS team to dedicate resources to the development of the software and hardware, as opposed to needing to dedicate engineers to finding bugs before they become bigger problems. To aid the developers in their quest to produce bug-free products, we were asked to develop an automated deployment process that will allow developers to update their Chromebooks and run basic sanity checks, called “smoke tests”. This will allow the testing process to be run quicker, as well as removing the need to have an engineer physically on the machine at all phases of the testing process.

NVIDIA has quite a bit of testing to do, but not enough time to do it. Every Chromebook they plan to sell needs to be tested with the latest version of the operating system. This problem is complicated by the fact that updating a Chromebook is a time-consuming and error-prone process. In order to reduce the cost and risk of error, a better method needs to be put in place for updating Chromebooks to the latest image.

Nightly builds are builds that occur once per day and include all the changes from that day. A nightly build is usually compiled by a machine that is as unbiased as possible, meaning no extra configuration of software has been done to the machine. Nightly builds are usually built at a time when developers are not actively working. This makes it clear what changes are included in the build. Although these builds occur automatically, there is currently no mechanism for testing them. An engineer can take the image, load it on an SD card, and run it on a Chromebook. This has obvious disadvantages, such as the dependency on an engineer to physically be there to start and observe the testing phase. With a limited number of very busy engineers, this is an unwanted and time-consuming process. The problem is that testing nightly builds is an essential part of the development process, but it is also a long, monotonous, and error-prone process. The solution is a system that can test the system automatically when the nightly builds completes, and have a list of tests that passed/failed waiting for the engineer to read when he/she comes in for work in the morning.

In order to ship any product, extensive testing must be done to ensure it behaves consistent with the expectations of the user. Within a computer software/hardware company, the team responsible for this is called the Quality Assurance (QA) team. In order to ensure the product runs as expected, the QA team needs to test various versions of software running on all different types of hardware. If problems are found, they need to be reported to the developers so they can be fixed before growing into bigger, and potentially unmanageable problems. This may not sound like a hard task, but consider the following example scenario:

A company plans to release a new line of laptops (5 in total) running an operating system they developed themselves. By the end of the day, it’s possible that as many as 100 changes have been submitted and are included in the nightly build of the OS. If a nightly build causes a laptop to fail some test, there can be as many as 100 different changes that broke it. If the QA team can only test one laptop per day due to the extensive manual labor involved, this means that each build on the laptop contains changes from the past

five days, or as many as 500 different changes! It is also possible that if a change that broke a laptop was just submitted that day, the laptop that was tested yesterday would not notice the failure until it was its turn to be tested in four days.

Although a somewhat crude example, this is certainly indicative of what can and does happen in some QA departments. There are just too many tests that need to be run, too many devices that need to be tested, and not enough QA engineers to accomplish all of the tasks. If they don't have the manpower or automation tools, the backlog becomes an unmanageable entity, growing longer while new bugs continue to pile up and go unnoticed. For a team as small as the NVIDIA Chrome OS team, automation is critical since manpower is very limited. If intervention is only needed during the case of a failure, an engineer can continue to work on other projects until a more pressing matter, such as a test failure, arises.

Over the past years, NVIDIA Chrome OS team used SD cards to manually copy Chrome OS images from build machine to target devices in order to test builds. This requires a lot of manual labor, which drives up the cost of development. Another downside to manual labor is that it is error-prone. However, with automation, the steps are executed exactly the same each time. This does not mean that it will never fail. However, it does mean that failures are not caused by incorrect commands or procedure, but rather the failures come from a problem with the hardware or software itself. To automate this process, the NVIDIA Chrome OS team would need a build server which pulls the latest build from the build repository and serves the image to all the target devices. They would also need Linux hosts that are each connected to a target Chromebook device to control and update the target device with the newest image and run tests on the device once the image is successfully installed.

3 Procedure

To accomplish a task that involves implementing automation, we must first understand how the underlying system works as a whole. This involves reading the technical documentation for the Chromium project to understand how the boot process works. The Chromium boot process is more complicated than most traditional systems, as it allows two Chromium images to be present at one time. They reside on separate partitions on disk and the bootloader is responsible for determining which one to load. There are two flags ("Successful Boot" and "Tries Remaining") that are used to determine if a currently loaded image has booted successfully or not. If it has not successfully booted and there are no more tries remaining, the bootloader will select the fallback root partition to boot. Once we understand how this process works as a whole, we can begin to automate it. The diagram below shows the flow of booting firmware A or B, and handling which one gets loaded. It also shows the recovery firmware flow. With a specialized debug board developed by NVIDIA, we can force the Chromebook to boot into the recovery firmware in order to reinstall a known good image. The Chromium OS design document[3] about firmware booting and recovery contains the following diagram showing the boot process.

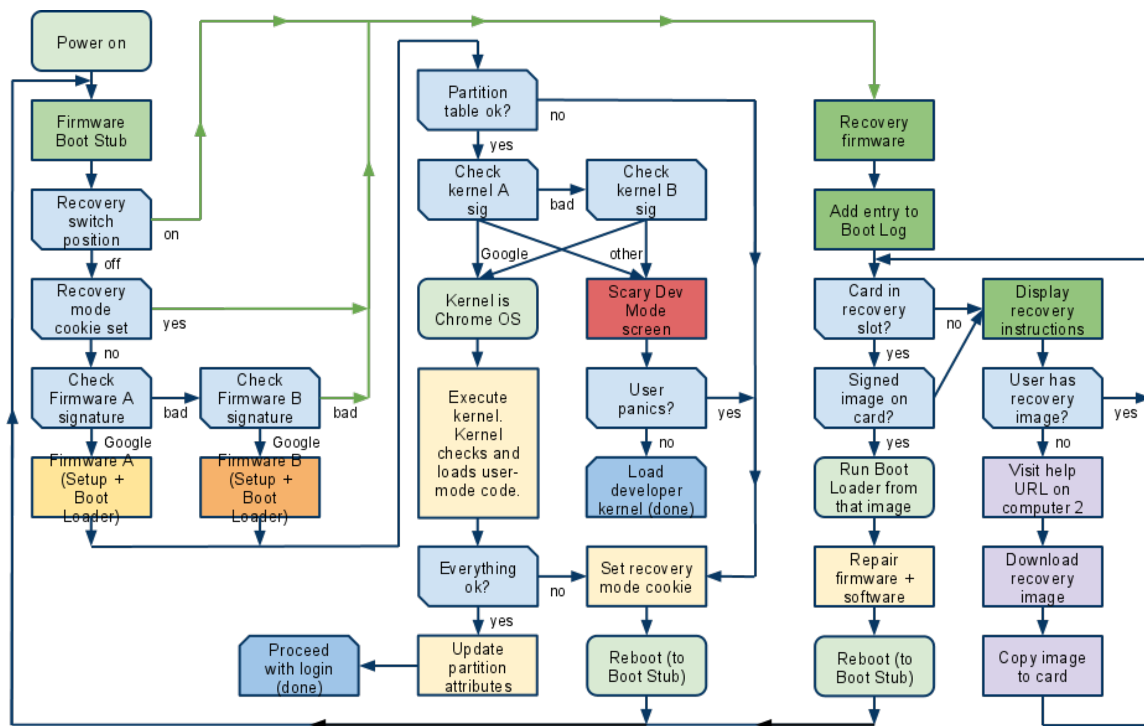

Figure 1: Chrome OS Booting and Recovery

We are paying close attention to the first and last columns of Figure 1. If the firmware A or B fails to boot, then we can set the recovery switch to the on position using the NVIDIA debug board. This forces the Chromebook to use the recovery firmware to install the image from the SD card. When the repair is finished, the Chromebook reboots into the new firmware which loads the recovery image that is now installed on the machine.

[3] `http://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery`

Although this is a tool that will be used primarily by NVIDIA employees for testing Tegra Chromebooks, much of the tools used were developed for the Chromium project and can be used with any target device configuration. Much of our testing was done on an Acer C720 shell that uses the Tegra124-Venice2 processor. There are a few NVIDIA specific pieces of software and hardware that were used that allows easier manipulation of the target device.

The procedure consists of six main tasks:

1. `Build Server` pulls latest build from `Build Repository`
2. `Build Server` starts Dev Server program to serve build to `Chromebook Target`
3. `Test Host` flashes bootloader to `target`
4. `Target` pulls build from `Build Server`
5. `Test Host` runs tests on `target`
6. `Target` is able to recover from corrupted builds, if necessary

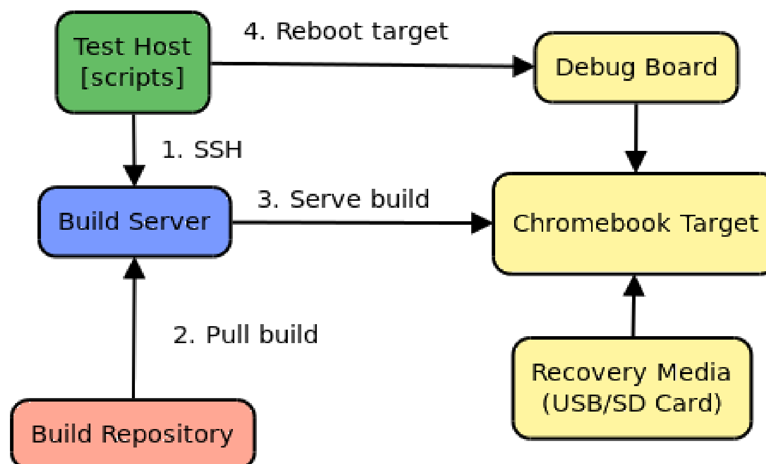

Figure 2: Deploy Flow

The flow diagram in Figure 2 shows the update process from beginning to end. The update starts out on the test host which contains all of the necessary scripts. The test host sends a script to the build server to obtain the latest nightly build, if a new one exists. Once the build has been downloaded, it is now ready to serve the build to the target. The test host tells the target to update itself using the build server. The host can send commands to the target through the debug board, which allows the test host to remain in complete control throughout the entire process. Once the target has been updated, the test host forces it to reboot. If all goes well, the tests are initiated once the target boots. If the update failed, the test host puts the target into recovery mode and Chrome takes care of restoring the image to the once found on the SD card.

3.1 Hardware Entities

As depicted in Figure 2 (Deploy Flow) at the beginning of this chapter, there are six major components in our procedure. These are described below in more detail.

3.1.1 Build Repository

The build repository, named "buildbrain", is the NVIDIA mobile team's internal storage for nightly builds. In buildbrain, the builds are organized by codeline, which is a combination of mobile and Tegra projects. The nightly builds remain in their entirety for a week or two until they are purged to free up space on the server. These builds provide the entire Chrome OS source, as well as the binary image. This binary image can be written to USB, converted to a virtual machine, or served as an update to an existing system.

3.1.2 Build Server

The build server is the machine responsible for downloading and extracting the latest nightly build, as well as serving the build to Chromebook target devices. The flow of the build server begins when the test host initiates the build server to obtain the latest build. Once the latest build is obtained from the build repository, the build server instructs the test host to tell the target that an update is available. The build server serves one image at a time, but can serve it to many different targets. This setup easily allows for scaling, since more build servers can be added if the load from multiple targets becomes overwhelming. If the build is already downloaded, no further action is taken, thus improving efficiency and saving resources such as bandwidth.

3.1.3 Test Host

The test host is the machine responsible for calling the update script to initiate the entire update process. Each Chromebook device is connected to a single test host. In a QA environment, each test host will be a dedicated Linux machine with one Chromebook connected to it. In a development environment, any developer machine can also function as a test host. The test host sends commands to the build server and target device while waiting for the commands to finish. The test host is responsible for instructing the build server to update to the latest image. Once the build server has been updated with the latest build from the build repository, the test host flashes the boot loader and initiates tests to be run. The test host is also responsible for recording all of the logging activity during all stages for the update process, as well as sending emails with the test results.

3.1.4 Debug Board

The connection between the test host and the target is established through the pm342 debug board, a piece of hardware capable of emulating several physical operations on the target, including resetting the target and putting it into recovery mode. The pm342 debug board connects itself to the test host through USB and to the target through a serial interface. Given this connection, the test host is able to open a terminal that speaks to the target using a terminal emulator. The pm342 debug board also supports a customizable serial number, allowing it to be identified among other USB devices connected to the test host.

3.1.5 Target

The Chromebook, often referred to as "target", is the device we are interested in automatically updating, for testing and quality assurance purposes. For the purpose of this project, no manual intervention, other than initially connecting the Chromebook to the test host, is needed. The Chromebook's main purpose in this

system is to listen to the test host and run commands. It also talks directly to the build server in order to obtain the latest build. Everything the target does is initiated and watched by the test host. If any failures occur, the test host ensures that a known good image is recovered to the device. This ensures that the target Chromebook is always bootable to a working state.

3.1.6 Recovery Media

In the context of this project and Chromebooks, recovery media is any type of removable media that can be used to restore a Chrome OS image to the Chromebook device. Since Chromebooks do not have optical drives, the recovery media is usually a USB drive or a SD card. Due to Chrome OS's heightened network security, booting over the network is not an option for recovery. Our automated deploy software can trigger recovery mode if a failure occurs. With a recovery media device inserted, the Chromebook can be forced to reboot into recovery mode to restore its image.

3.2 Software Tools

To accomplish a task that provides automation and requires cooperation from many different hardware entities, several software tools have been used and are explained below in more detail.

3.2.1 Bash

The Bourne Again SHell (bash)[4] is a shell command executor and language interpreter. The command executor is essential for running commands to automate the deployment process. For the purpose of our software, the language interpreter part of bash is helpful in allowing various functions to be defined and used across many source files. This allows our code to use the same functionality for tasks such as logging and error handling. Some existing NVIDIA scripts also use bash, which makes integration fairly simple.

3.2.2 Python

The Python programming language "is a great object-oriented, interpreted, and interactive programming language."[5] Google uses Python as its primary language for the developer tools in the Chromium project. We have been leveraging existing Python tools, such as the Dev Server, and also writing our own. In order to provide useful and meaningful logging from all of the hardware entities, we've developed a Python program acting as a server that accepts messages and logs them locally. This "remote logging server" runs on the test host and logs all messages from remote clients to the same file. This decreases the time and effort needed to track down the logs from any given update run. By using Python's flexibility, this program can be extended to include other commands, thus allowing all machines on the same network to contact each other through this hub residing on a single machine. By using the Python programming language, we are keeping consistent with the Chromium developer tools and allowing our tools to be easily run in the same environment without setup or modification.

[4] `https://www.gnu.org/software/bash/manual/bashref.html`
[5] `https://wiki.python.org/moin/`

3.2.3 Minicom

Minicom is a terminal and serial emulator. It allows communication between the target device and the test host. When the target is connected to the test host via the pm342 debug board, a `ttyUSB` device file is created automatically on the test host. By using minicom to open the `ttyUSB` file, we're able to create a terminal that speaks to the target. Through this terminal we can log into the target and execute Linux commands. In addition to allowing communication, minicom also logs the sessions to files. This allows engineers to quickly go through the logs to see the output of commands that were run on the target device. These log files can be very useful, since minicom is used to update the latest image, flash the boot loader, and initiate the tests. If any of these steps fail, the log file should be able to provide insight as to why.

3.2.4 Expect

Expect is an extension of the TCL scripting language, used mostly for automating interactions over applications such as `telnet`, `ftp`, `passwd`, `fsck`, `rlogin`, `tip`, and `ssh`[6]. Expect can send commands to the terminal, and wait for certain output from terminal. In our project, interacting with target devices through the minicom terminal emulator can be difficult to manipulate using scripts written in bash or shell, but it can be easily achieved by using Expect since it simply sends our commands to stdin and reads output from stdout, no matter what root environment we are in. It also deals with case handling very well: Expect can branch over all the cases of output, and we can define in each case, what will happen next. For example, while we are logging into our target devices through minicom, we can write two cases in Expect with the first one being waiting for the password prompt to come out and then send the password; and the second case can be exiting with an error code if it times out.

3.2.5 Dev Server

The Dev Server is a Python program developed by Google as a part of the Chromium OS developer tools[16]. It starts an update server at a specific port (usually 8080) and serves new images to target devices. The Dev Server runs as a HTTP server that accepts requests from target Chromebooks and returns payload responses that represent an updated image. Chromium OS comes with a command line tool (`update_engine_client`) to handle the client side of this process, which will obtain the update and install it.

3.2.6 pm342 Control Tool

The pm342 debug board has "phidget" capabilities, which means that it can be controlled by a personal computer through USB connection. The pm342 control tool is the software that manipulate the pm342 debug board. Having this software allows automation of hardware functions such as pressing the reset button and putting the target into recovery mode. The pm342 control tool allows easy manipulation of the debug board and provides a simple interface for shell scripts to use.

3.2.7 cron

Cron is a useful system daemon that allows users to execute commands or scripts automatically at a designated time/date. With the help of cron, we can run our update script at a specific time of the day or at every

[6] `http://www.nist.gov/el/msid/expect.cfm`

other hour without manual intervention. Crontab, stands for cron table, is a simple text file that contains a list of cron job entries. It is edited with a command-line utility. Each entry is composed of the command or script name and a specified time. These commands or scripts (and their run times) are then controlled by the cron daemon, which executes them in the system background.[7] Each user in the `UNIX` system has its own list of cron jobs. These jobs will run in the background as the user, regardless of whether the user is actually logged into the system.

3.2.8 Run Remote Tests

Google has provided us an important front end for running our tests on the target: `run_remote_tests.sh`. The script runs in the Chromium OS chrooted environment and takes in several flags to specify the target's IP and the tests to be run on the target. Upon completion, the script will generate detailed test reports that can be useful for debugging purposes.

3.2.9 sSMTP

sSMTP is a simple MTA (mail transfer agent) program to relay e-mail messages from a local computer to a mail hub. It has advantages such as being simple and lightweight. It simply relays e-mails to a SMTP (Simple Mail Transfer Protocol) server, so it is not a mail server like sendmail or postfix, and it doesn't receive mail, expand aliases, or manage a queue.[8] A major benefit of using sSMTP is that it can be quickly set up on a build server or test host to be able to send emails to NVIDIA engineers. Using sSMTP, we can generate e-mails which contain useful information (such as test results) immediately after the deployment cycle, let sSMTP deliver them to a NVIDIA mail hub, then dispatch them to the designated recipients.

3.3 Workflow

Our task is easily divisible into six smaller tasks: pull build, start dev server, flash bootloader, load latest image, run tests, and optionally reboot into recovery. Each stage can be run individually, assuming that the stage before it has successfully completed. For example, the bootloader can be flashed without first pulling the build if a build has already been pulled, or you can jump straight to running tests if you don't want to flash a new build but want to re-run tests. Each of these steps are explained below in more detail.

3.3.1 Pull Build

In order to test the stability of the current state of Chrome OS development, NVIDIA's Quality Assurance team has the Chrome OS source code built nightly and stored in NVIDIA's build repository called buildbrain. For the purpose of this project, the newest image of Chrome OS needs to be made available by a server to all Chrome OS target devices. To accomplish this, the server first needs to pull the latest nightly Chrome OS build image. We query buildbrain to get the location of the most recent successful nightly build, then SSH into the build machine from the test host to download the latest build as a compressed `tar` file from buildbrain. After we extract the compressed build, we will get a file called `chromiumos.run`, which is an

[7] `https://help.ubuntu.com/community/CronHowto`
[8] `https://wiki.debian.org/sSMTP`

executable that can generate all of the Chrome OS source, along with built images which were produced by the source. Executing `chromiumos.run` will get us the latest built test image.

3.3.2 Start Dev Server

Google has developed what it calls the Dev Server, a Python program to help with updating Chrome devices with customized versions of Chrome OS over the network. It allows developers to update their Chromium OS devices with new builds without having to copy them via a USB drive. There are mainly three things that Dev Server can be used for: updating target device with a specific image on the development machine, customizing target device with your own packages, and building specific packages on the dev server and installing them on the target device. For the purpose of this project, we used the Dev Server for installing the latest built image on the target device by passing in the `--image` parameter to direct it to the location of the latest built image. We ran the Dev Server program on our build machine, so the build machine can serve the test image to all the target devices.

3.3.3 Flash Bootloader

The bootloader is a piece of firmware used to boot up the Chrome OS upon powering up the target (in our case, the NVIDIA Tegra-124 Venice 2 Chromebook). The bootloader will initialize the bare hardware, then load the kernel. NVIDIA utilizes two kinds of bootloaders for the target:

- Core-boot - Initialize hardware then load the kernel directly from internal storage
- U-boot - Initialize hardware then try to load the kernel from an external storage device (USB, SD card), but falling back to internal storage if none are available

There are more details concerning Core-boot and U-boot, but for the purpose of our project, we utilize Core-boot to test out the Chrome OS image stored internally and if the internal image is corrupted, we flash U-boot to the target and boot from a bootable flash drive or SD card already connected to the target to begin the recovery procedure.

With NVIDIA Chromebooks still under development, a secondary boot device, the SPI flash memory is used in addition to the primary boot device, the iROM. The target is currently set to boot from the SPI flash memory, allowing us to update the target's bootloader.

Flashing the bootloader to the target requires a NVIDIA pm342 debug board and a USB A-A cable. The pm342 debug board is an internal NVIDIA board originally designed for testing the Tegra 114 platform. pm342 is controllable through software, through which we're able reset the target and force it into recovery mode. When in recovery mode, the target treats itself as an USB device and the test host, through the USB A-A cable, uploads the bootloader to the target's SPI flash memory. After flashing the bootloader, we use the pm342 debug board to reset the target. The new bootloader will be used when the target reboots.

Should the booting go wrong, boot logs can be used to help determine the cause. To capture the target's boot log, we rely on a piece of software called minicom. Minicom is a text-based terminal emulation program for Unix-like systems. On the test host, we invoke minicom to listen to the pm342 board, which is listed as a device under the `/dev` file system. While connected to the test host using a USB A-B cable, the pm342 board is also connected to the target through a serial interface. Through the pm342 board, the test host is able to establish connection with the target without using Ethernet. Using minicom's capture mode, most

activities on the target are captured, including the boot log. The captured information is stored in log files that are available for later inspection. These log files contribute significantly to the debugging process.

3.3.4 Load Latest Image

Once we have booted the target from internal storage using core-boot, we can run `update_engine_client --update` on the target to download and install the latest nightly build image served by the Dev Server. `update_engine_client` is the client side implementation of Google's Dev Server, which allows the client to force an update using a specified server.

In Chromium OS, a hard drive contains at least three partitions: one for state resident on the drive (user's home directory/Chromium programs, logs, etc.) and two for the root file system[17]. Only one of the two partitions for the root file system is used for booting up the system at a given time. The other one is used for updating, or falling back to, if the current partition is corrupted[18]. When downloading the new image from the Dev Server, `update_engine_client` stores it in the unused root file system partition. Using this setup, `update_engine_client` is able to install the downloaded image onto the target with the target's OS still up and running.

If booted from a corrupted image, the target might fail to download the latest update. In this scenario, we put the target into recovery mode, install the healthy image onto its internal storage then retry the update.

3.3.5 Run Tests

The NVIDIA Chrome OS development team currently do not have their own test cases. However, Google has developed a comprehensive collection of test cases for Chrome OS and, because Chromium OS is an open source project, we have access to those test cases. At the time we were developing our solution, the NVIDIA team was relying on manual testing to validate the newly built images. The Google Chromium OS test suite provide an excellent option for smoke testing - preliminary testing that validates the basic functionalities of the system. By utilizing an existing Chromium OS script, we're able to run the Google Chromium OS test cases against the target.

We've implemented log files to log our entire testing process. Throughout our log files, the users can identify the failed test cases. In addition to our logs, the Chromium OS test suite also generates its own test details. These details are saved to our log directory with their specific locations written to our log files. With both testing and logging in place, the users can easily identify the potential problems with the new images by running our script.

3.3.6 Reboot into Recovery

There are several cases where the target needs to be recovered to the last healthy Chrome OS image:

1. The target failed to boot after the new bootloader is uploaded to the targets SPI flash memory
2. The target failed to pull the latest update from the dev server, provided that the dev server is working properly

3. The new image loaded to the target failed to pass all test cases

In any of the above cases, we immediately start the recovery procedure. On the test host, we have set up a recovery directory which holds a known good image. In addition, a bootable external device (USB flash drive or SD card) is connected to the target. The following steps are necessary to install the recovery image onto the target:

1. On the test host, change to the directory holding a healthy image
2. Flash U-boot to the target
3. Reboot the target (since U-boot is already flashed to the target, the target will boot from the external device, which stores the healthy image)
4. Install the healthy image onto the target
5. Reboot the target again to finish the recovery

4 Design Considerations

The automated deploy system that we are designing is to be used by NVIDIA QA engineers, as well as NVIDIA developers to test their code on their own Chromebooks. The obvious main requirement is that it has to work, but it also needs to be beneficial to use, meaning that using the tool saves time, effort, and/or resources. The system requirements have manifested themselves into our design considerations, each of which is listed below in more detail.

4.1 Speed

Since deploying a Chrome OS build is a time-consuming process, our system needs to streamline it as much as possible. One of the major goals of this project is to save time for NVIDIA engineers so they are free to work on other more pressing matters. Our system keeps running commands after the previous one finishes, so there is no wasted time waiting for input. We have identified two major bottlenecks due to the size of the Chrome OS build. The first bottleneck comes from downloading the build, and the second one comes from extracting it. Attempting to remove one bottleneck, however, could result in the other bottleneck becoming far too damaging. The bottlenecks target two hardware limitations: network and CPU/RAM usage. If the network is fast but the host is a weak machine, it would be possible to do the CPU/RAM intensive computations on the server which produced more data, thus sending more data through the network. While this allows less work to be done on the host, this places more computational stress on the build server. If more compression is done on the build server so the network doesn't need to send as much data, the build server still suffers from extra computational stress. Also, the host will need to undo all of the work previously done by the build server. While this saves bandwidth, the time saved through the network will be lost during the compression and extraction steps. Our current balance has the build server download the image as it appears on buildbrain, then performs the extraction. The current balance will benefit machines with faster networking and/or more CPU/RAM power, but will not penalize machines that do not have the extra ability.

4.2 Reliability

Our automated deployment system needs to work with constantly changing versions of software and various configurations of hardware. Our system needs to take many possible scenarios into account and accommodate them accordingly. Potential scenarios we may encounter could include intermittent network failure, target device connections unplugged, and/or missing images on buildbrain. Since our system uses multiple machines, network connection can be a point of failure. If network failures occur, we can repeatedly try to connect, but will give up after enough tries, much like an engineer would, and report a failure. The engineer who receives the error will know it was caused from the network and can take appropriate actions to fix it. Since we are writing software to interface with hardware, the timing is not always consistent. This means that it sometimes it takes longer for a machine to do certain tasks like boot, respond to commands, or communicate over the network. We have accommodated this variation in time to a degree. As such, we will not be waiting around for hours for a machine to boot, since no engineer would either. A Chromebook is supposed to boot in seconds, so any boot that takes minutes should be considered a failure. Once again, the engineer will know that the fail was caused by slow boot times, and he/she can investigate the boot times

more in depth by looking at the specific build.

4.3 User-Friendly

Another goal of this project is to have a system that will be deployed to many different QA facilities and developers. This means that the system will be set up frequently and used by many different people in differing environments. To help the initial setup process, a setup script will do all of the configuration needed to set up a machine as a server or a host. Once all of the machines are set up using the script, the updating takes place through a single entry point: the `update.sh` script. Having a single entry point helps to reduce the learning curve and knowledge needed to get an automated deploy up and running. From the perspective of a developer, the single entry point makes it easy to trace through the code and see what it is actually doing. Our logging features will also help to guide and inform the user and/or developer about what is going on at every state of the update process. To make it easy for the user to add/remove test cases to run on the target, a test case configuration file is implemented holding the names of all intended test cases.

4.4 Accountability

Any action the system performs must be documented and accounted for. For example, if the new updated system fails to boot, the QA engineer should be able to see that an update was attempted but failed at a specific step. The engineer can then look closer at the step that made it fail so he/she can figure out exactly what went wrong. The logging features will aid in showing what steps have been executed and what the output was from commands that failed. Since our system consists of shell scripts, they can be run in debug mode to show exactly what commands are being executed. This makes debugging very easy, should the need for it arise. With the way our code is organized into modules of functions, changes to code usually only affect the local scope. This means that changes to one function will not cause another function to randomly fail.

4.5 Correctness

Each part of the system should provide the same output if given the same input multiple times. For example, if a new build is available, the portion of the system that checks for updates should always report that one is available. When we put these parts together, we get a system that contains only known working parts that can provide reliable and correct results every time. In order to ensure our system consists of only quality code, we've implemented a code coverage and test suite environment. The test suite tests individual functions of our system and makes sure they function as expected given various ranges of inputs. We are testing normal expected inputs, inputs that can cause errors, and inputs that may never happen but are possible. The code coverage tool ensures that the code we are running is actually being used and executed. Any code that is not being used will be reevaluated to see if it is necessary, and possibly staged for removal or modification. Error detection code is necessary, but may not often (sometimes even ever) be run. This will unfortunately bring our numbers down, but we take this into account when we look at the evaluation of our code and consider it to be a required component of our system.

4.6 Lightweight

One of our major goals is to make our script as lightweight as possible. In this way, we can not only ensure low system requirements, but also make our script easy to setup. A low system requirement makes our software able to run on machines that have less expensive hardware so we can reduce the total cost. A small distribution size and an easy setup process save resources such as time and space, while improving the efficiency of the Chrome OS team's deployment cycle. Therefore, we always make ensuring that our script are as lightweight as possible a top design consideration. This impacts how we chose our software. First of all, we picked sSMTP as our Mail Transfer Agent (MTA) because it's small, and it's easy to configure and deploy. We didn't have to setup a full MTA server. On the other hand, we used as many of Linux's built-in utilities as possible so we can avoid installing too many packages as our software dependencies.

5 Testing and Code Coverage

One of the design considerations previously outlined is “correctness”. In order to prove to others that the code is in fact doing what we expect, we have set up a testing framework. This testing framework allows tests to be run on the various common functions. With the way the code is organized, most functions can be tested in isolation with no prior set up required. Not only does this make writing test cases easier, it also reduces the risk that the environment or previous runs of the function will have any effect on the results. It is important that the code maintains this property, since it is what makes the test results useful, as well as providing a simple environment for running tests in. Having this testing environment also encourages newly written code to have the property of being easily testable, increasing the overall quality of the code as a whole.

Testing code has many benefits, the most seemingly obvious one being that it proves code to be functionally correct. In this case, functionally correct means that the function provides the expected output, given a known input. It does not mean that the intention of the code is correct. For example, I could write a function called `is_odd` and return false if the number is odd, or true if it is even. Basically, the `is_odd` function is actually an `is_even` function. If my test cases check that false is returned for odd numbers, then my code is functionally correct, but has a very misleading name, and may be considered wrong. In this case, the issue is that the test cases are wrong. The function `is_odd` on an odd number should return true. This problem is easy to spot by looking at the test cases and expected output.

In addition to testing functionality, code testing allows future developers to make changes without worrying about breaking previous compatibility. If a developer makes changes and then run tests that fail, the developer can revert the change without wasting any more time. Without test cases, a developer would have to manually go through and make sure that inputs that previously worked with the old code still work with the new code. This is obviously undesirable, since it involves manually testing the code each time a change is made. Eventually, everyone who makes changes in that area has essentially done the job of test cases, but at a much longer duration of time.

In software engineering, the terms “coupling” and “cohesion” are often introduced in discussions about software system design. The term coupling refers the amount each module/file is interconnected with the other modules in the system. The term cohesion refers to how closely related each function of a file are to each other. It is the generally accepted opinion in the software development community that loose coupling and high cohesion are best[19]. This essentially means that modules/files do not depend on each other, and the functions within files all work to accomplish the same goal. From a testing standpoint, this makes each pass/failure more meaningful. In a tightly coupled environment where all the files depend on each other, a single failure in one file could cause failures in every other file. Test cases and code coverage can help find areas where tight coupling can be a problem.

5.1 Tools

In order to focus on the main task at hand but still provide useful code coverage and testing feedback, we have leveraged two open-source projects and wrapped our own code around it. These tools are for internal use by developers of the software and are not necessary to run the automated deploy. They simply provide

feedback to developers about the quality and correctness of the code.

5.1.1 bashcov

The bashcov project[9] allows shell scripts to be run in their normal environment, then produces a HTML file that shows the code coverage. Code coverage is a measure of the number of lines of code that have been executed in relation to the total number of lines of code in the system. While attaining high levels of code coverage is often desirable, it is important to note a few issues:

- Code coverage with shell scripts is difficult
- Results sometimes take more in-depth investigation
- With much error checking in place, quite a bit of code may rarely be run

These issues, however, can be eased with the coupling of multiple unit tests. Running code coverage in conjunction with unit tests can greatly enhance the ability to see and interpret the results in a more meaningful manner. The figure contains an example of bashcov output from running unit tests on the logging portion of the code.

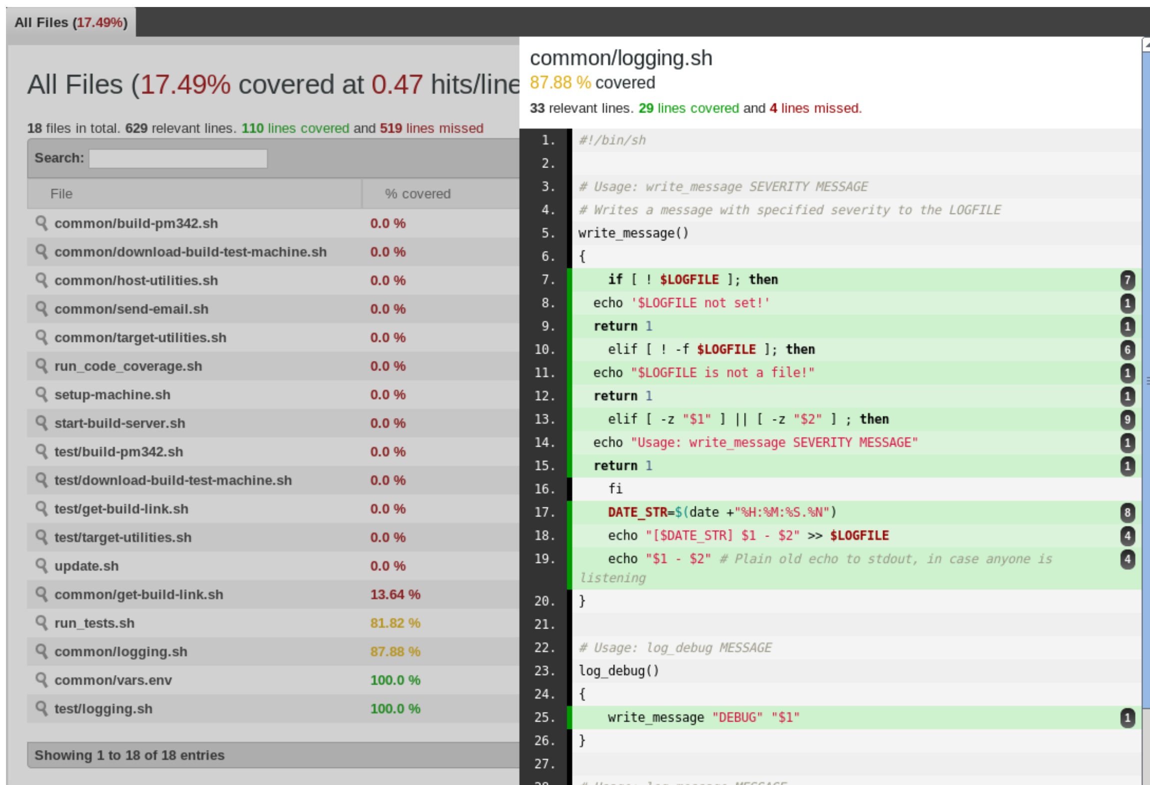

Figure 3: bashcov Code Coverage Output

As visible in Figure 3 (bashcov Code Coverage Output), there are many files that have not even been touched, but the logging file has been thoroughly executed and tested by the unit tests. By running tests on only

[9] `https://github.com/infertux/bashcov`

the logging portion, it is apparent that the logging features do not depend on other files, which is good since it promotes loose coupling. Although it reads 87.88% covered, 100% has actually been covered. The reason is that the lines that were considered not covered were part of a multi-line command that echoes text. Unfortunately, this is one example of a limitation of the software trying to do code coverage from within bash. An engineer, however, can quickly look at the uncovered lines and know that it is nothing to be concerned about, and that the coverage is in fact 100%.

5.1.2 shUnit2

The shUnit2 project[10] is a bash implementation of the xUnit family of unit testing frameworks. With most of the code organized into convenient functions, writing unit tests for the Chrome OS automated deployment project is very simple. Tests on the individual functionality ensures that these functions are suitable to be called from other functions. This allows a streamlined process when debugging the code, since test results can either confirm or deny a specific function as being problematic. One issue with unit testing in the context of the Chrome OS project is the hardware component. It is very difficult to test hardware-specific functionality, such as flashing the bootloader, without having the hardware physically hooked up to the machine. These issues will show up in the code coverage tools, and engineers can determine where to proceed in the debugging process. In our code, we do our best to isolate such functions, allowing test cases to cover more of the code. In cases where this is unavoidable, "mock objects" can be used to trick the scripts into thinking they're talking to a real machine, when it is actually just talking to a customized script to produce desired cases.

Writing test cases in shUnit2 is just like in any other xUnit framework implementation. The test itself is a function that contains various function calls or shell commands, followed by a series of "asserts" in which expected results are tested against the actual results. This allows for complex commands to be run first, but then followed by simple assertions that can be easily read in everyday language. A simple test is shown below in Figure 4.

```
test_build_pm342()
{
  rm -f /tmp/pm342
  build_pm342 /tmp/
  assert '[ -x /tmp/pm342 ]'
}
```

Figure 4: shUnit2 Test

The test in Figure 4 calls the `build_pm342` function with the arguments "/tmp". Once that function completes, it asserts that an executable file has been placed at "/tmp/pm342". If that is not the case, something went wrong in the `build_pm342` function and the test case will fail. The engineer will know to look at the `build_pm342` function for bugs. Combined with the code coverage, engineers can use the information to find out which areas of the code were called and what failed. They can then make improvements to the function, optionally adding more test cases, and re-run the tests.

[10] https://code.google.com/p/shunit2/

6 Results

This section will attempt to quantify the effects of the update script. These results are measurements gathered from running multiple iterations of the software in varying environments.

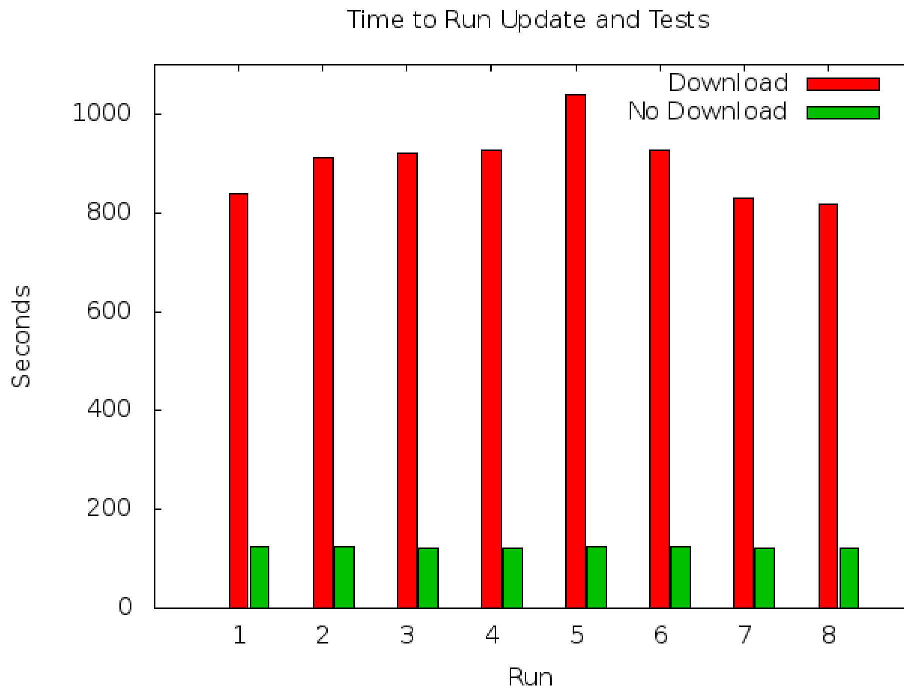

Figure 5: Update Script Runtimes

The graph in Figure 5 shows the two types of updates that can occur: an update where the build server needs to download a new image, and one where it does not, since the previous image is cached. All runs in the graph are successful runs from start to finish. An update and test without needing to download a new image usually takes about two minutes. In the worst case scenario where the build needs to be downloaded, it can take around 850 to 900 seconds, which is about 13 to 15 minutes. Not only does the graph show the average run time, it also shows that there is not much variation between runs, even when networking is a major component of the run. This is valuable information for developers, as it allows them to plan adequate time to automatically update their machine.

6.1 Time Benefits

With developers not needing to guide their Chromebook through an update, they are free to spend their time elsewhere. This time can be spent on anything from developing new features to fixing bugs. With such a small team, developers are often working on multiple features or bugs at the same time. This can cause unnecessary stress on developers who have too much to work on, but not enough time to finish it all. By automatically deploying the latest Chrome OS image, developers have one less thing to worry about. They also never have to worry about needing to restore their Chromebook due to it being in an unusable state. This helps to keep their list of activities smaller and more manageable.

While our system does not reduce the time needed to perform a deploy, it does execute the steps as soon as possible, which can cut down on idle time. The system is also completely autonomous and requires no manual intervention. This allows the time saved to be quantified as the time needed to execute each deploy. Since they system is doing the deploy, not the developer, the developer recoups that time to work on other projects.

6.2 Cost Benefits

When developers spend their time developing with reducing deployment time, they are able to put in more combined effort hours than before. The extra effort hours saved will go a long way for the small development team. Not only will the Chrome OS team feel the benefits, NVIDIA as a company will see benefit from the extra work the Chrome OS team will now be able to produce. Our system also has the ability to be run with the test host and build server as the same machine. This would allow NVIDIA to deploy the system using existing hardware, saving them the cost of purchasing extra machines. The deployment system is also fairly light on system requirements and resources, making it possible to use an existing server instead of a dedicated machine.

6.3 Accuracy

The automated nature of the system ensures that commands will not be mistyped or run out of order. This ensures that any failures that occur are due to a bad image, not errors in the deployment process. Having this accuracy in place helps to maximize the time and cost benefits by preventing costly and hard to debug errors. The automated deploy scripts also ensures that each individual run of the deploy produces output that is consistent. This means that running the script repeatedly with the same parameters will produce the same output. For developers and QA engineers, this is a valuable resource as it reduces the areas to investigate during errors.

7 Future Work

While the work we've done is already of benefit to NVIDIA, there is room for potential extensions of this project. In this section, we will discuss potential future ideas for the software system, as well as features that could not be implemented due to time constraints.

7.1 Scalability

Our testing has been done in a small environment with only one test host and one target Chromebook. The plan is to deploy this in what NVIDIA calls a mobile sanity environment where multiple Chromebooks will be tested. The automated deployment system is only for use by NVIDIA on their internal network, where bandwidth is usually very fast. There is a one-to-one relationship between the Linux test hosts and Chromebook targets. However, there is a one-to-many relationship between the build server and test hosts. If the many portion of this one-to-many relationship gets too much to handle, multiple build servers can be put in place to ease this load.

Most high-volume websites deal with load-balancing in a similar fashion. To prevent a single machine from getting overwhelmed with requests, multiple machines are added to handle the load. A single machine, commonly called the "load balancer", functions as a traffic cop to direct each request to a suitable server. Since the build server runs a small HTTP server, this type of scalability is very easy to implement.

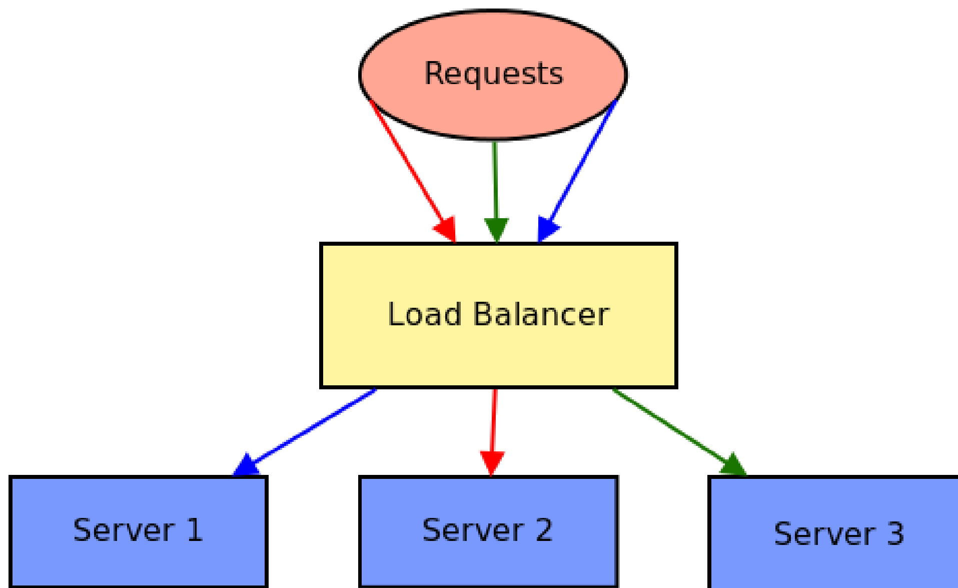

Figure 6: Load Balancing

Figure 6 shows the basic flow of a load balancer. Multiple requests come in to a single machine (the "load balancer") and they get distributed to available machines based on the load the machines already have. This is a simple and effective way of distributing HTTP traffic to maintain high download speeds and ensure availability.

7.2 Other Projects

Although our Chrome OS deploy project is specific to NVIDIA's Chrome OS project, the system has generic components that can be modified to work with existing projects. For example, Android devices could use this same setup, following a few minor changes. Some steps that would need to change would be the place where the build gets downloaded from, device specific flashing procedure, and running tests. NVIDIA already has a testing harness that encompasses flashing, booting, and running tests. By extracting the Chrome OS specific items from our system and writing them as "plugins" for the testing harness, the automated deploy system could potentially use the testing harness to automate testing for all NVIDIA products.

7.3 Missing Features

Due to the limited timeframe to implement our project, there are some features that were skipped over. However, all of the major features required to run our system have been implemented. Below, we will briefly discuss some features we looked over, why we chose to do so, and how they may be implemented in the future.

7.3.1 Web Access to Test Reports

Since NVIDIA does not have their own Chromebook tests at the moment, we are running official Google Chrome OS tests on the target Chromebook. These results are stored on the test host in the same folder that holds the update log files. Our system does nothing more than store the log results for an engineer to view later at a convenient time. A nice supplement to the current system would be to process the results and provide an HTML page with easy to read results. In addition to serving them statically, some type of intelligent aggregation of previous run statistics would also be useful. NVIDIA's current product-agnostic testing harness provides many of these features, so implementing this would be as easy as writing a plugin that allows the test harness to talk with our system.

7.3.2 Automatically Update Recovery Image

The current recovery image (described in section 3.3.6) is simply an image of Chrome OS that is known to boot properly and provide USB and Ethernet access. In order to prevent builds from going "stale" and becoming out of date, it would be beneficial to be able to "tag" a particular build as the recovery image. The build server could be responsible for obtaining and holding on to this build before the build repository deletes it. In order to get it on the USB device connected to the Chromebook, the test host can instruct the Chromebook to download the image from the build server and run the `dd` command to write the image to the USB drive. This allows each target to automatically keep their recovery image up to date. This feature would become very useful in the event that a Chromebook needs a driver that is not available in older builds.

7.3.3 Inject L4T (Linux for Tegra)

Linux for Tegra (L4T) is a software board support package (BSP) for specific NVIDIA reference platforms. It provides:

- Bootloader and Boot Configuration Tables (BCT)

- Linux kernel binary and source code
- Reference file system
- Binary userspace drivers for multimedia, graphics
- Demonstration applications
- Host utilities for flashing

NVIDIA's plans for a Tegra powered Chromebook involve providing the end-user with a device that will outperform other ARM Chromebooks, and also last longer. In order to provide this experience, NVIDIA needs to test the L4T library on their Chromebooks. Currently, there is no method for building Chrome OS with L4T enabled. It is up to the developer to copy the necessary libraries to the target device for testing. To expedite testing and prevent errors, automatic injection of the L4T libraries would be a desirable feature. In our current setup, there are two main places where the libraries could be injected:

- On the build server before the image is served
- Downloaded and installed by the target on a running system after being updated

These two scenarios would provide the same end result, but require different procedures. If the L4T libraries are to be injected into the image that is being served, the image would need to be unpacked so the libraries could be added. The unpacked image, which now contains the L4T libraries, would need to be repacked before it could be served to the target. Once the target downloads the update, it is now ready for any and all Tegra tests. If the other approach is used, the L4T libraries would not be injected until the target has successfully updated, but before the tests are run. During this in-between time, the test host can instruct the target where and how to download the L4T libraries.

8 Glossary

- bootloader - firmware responsible for determining how to load the operating system
- Chromebook - consumer available device running Google Chrome OS
- Chromium OS - open source project used primarily by developers, with code that is available for anyone to checkout, modify, and build
- chrooted environment - an entire Linux system residing within another system. A user may chroot (change root) into the directory and is presented with a new root system.
- Flash (flashing the bootloader) - act of telling Chromium OS which of two partitions it should boot from
- Google Chrome OS (also Chrome OS) - Google product that hardware manufacturers ship on Chromebooks for general consumer use
- Image - binary components making up a complete Chromium system
- Linux test host - machine connected directly to a Chromebook that is responsible for starting the automated update process
- Live CD/USB - method of booting an entire operating system from removable media, such as a CD or DVD
- minicom - serial and modem control emulation program
- MTA (Mail Transfer Agent) - program responsible for sending and receiving mail at the system level
- Nightly build - build of software that includes changes from that day
- partition - logically separated unit of a disk drive that can be treated as a separate disk
- pm342 - hardware and software component of the NVIDIA debug board
- PXE (Preboot eXecution Environment) - method of booting/installing an operating system by downloading it from the local network
- Quality Assurance (QA) Team - group of engineers responsible for testing for and reporting bugs on software/hardware in development
- root partition - partition where the essential operating system files reside
- SoC (System on a Chip) - single chip that integrates all components of a computer, including: microprocessor, memory (ROM, RAM, etc), timers, external interfaces (USB, Ethernet, VGA, etc)
- Target - device running Chromium/Chrome OS, primarily used in a development context
- `update_engine_client` - client side implementation of Googles Dev Server, which allows the client to update using a specified server
- Version Control - software system that allows changes to source code to be tracked and pushed out to multiple users

9 References

[1] NVIDIA. *NVIDIA®: The Visual Computing Company*. URL: http://www.nvidia.com/content/company-info/introduction-to-nvidia.pdf.

[2] *Parallel Programming and Computing Platform — CUDA*. URL: http://www.nvidia.com/object/cuda_home_new.html.

[3] NVIDIA. *NVIDIA History*. URL: http://www.nvidia.com/page/corporate_timeline.html.

[4] *NVIDIA Rolls out "Tegra" Chips*. 2June 2008. URL: http://archive.techtree.com/techtree/jsp/article.jsp?article_id=89833%3E.

[5] Mike Issac. *NVIDIA Unveils Tegra 3, World's First Quad-Core Mobile Processor*. 7Nov 2011. URL: http://www.wired.com/gadgetlab/2011/11/nvidia-tegra-3-processor/.

[6] NVIDIA Corporation. *Introducing NVIDIA® Tegra® 4, The World's Fastest Mobile Processor*. URL: http://www.nvidia.com/object/tegra-4-processor.html.

[7] Kevin C. Tofel. *Confirmed - Nvidia Working on Chrome OS Tegra Devices Tech News and Analysis*. 23 Sept. 2009. URL: http://gigaom.com/2009/09/23/google-chrome-os-tegra/.

[8] Dinsan Francis. *Google's Own "Seaboard" - A Tegra 2 Powered Chrome OS Touch Device*. 28 Apr. 2011. URL: http://www.chromestory.com/2011/04/googles-own-seaboard-a-tegra-2-powered-chrome-os-touch-device/.

[9] Kevin Parrish. *"Puppy" Chromebook May Have Nvidia's Tegra 4*. 8Feb. 2013. URL: http://www.tomshardware.com/news/Chromium-Puppy-Tegra-4-T114-Dalmore,20975.html.

[10] Steven J. Vaughan-Nichols. *The Secret Origins of Google's Chrome OS*. URL: http://www.zdnet.com/the-secret-origins-of-googles-chrome-os-7000012215/.

[11] Jeff Nelson. *Inventing Chromebook*. 2012. URL: http://blog.jeff-nelson.com/2012/11/on-inventing-chromebook.html.

[12] Sundar Pichai. *Official Blog: Introducing the Google Chrome OS*. 7 July 2009. URL: http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html.

[13] Chris Kenyon. *Google Chrome OS and Canonical*. 19 Nov. 2009. URL: http://blog.canonical.com/2009/11/19/google-chrome-os-and-canonical/.

[14] The Chromium Projects. *Chromium OS FAQ*. URL: http://www.chromium.org/chromium-os/chromium-os-faq.

[15] *Gerrit Code Review - Uploading Changes*. URL: https://gerrit-review.googlesource.com/Documentation/user-upload.html.

[16] The Chromium Projects. *Dev Server*. URL: http://www.chromium.org/chromium-os/how-tos-and-troubleshooting/using-the-dev-server.

[17] The Chromium Projects. *Disk Format - The Chromium Projects*. URL: http://www.chromium.org/chromium-os/chromiumos-design-docs/disk-format.

[18] The Chromium Projects. *File System/Autoupdate*. URL: http://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate.

[19] Jeremy Miller. *Patterns in Practice: Cohesion And Coupling*. URL: http://msdn.microsoft.com/en-us/magazine/cc947917.aspx.