Live upgrading thousands of servers from an ancient Red Hat distribution to 10 year newer Debian based one.

Marc MERLIN
Google, Inc.

Abstract

Google maintains many servers and employs a file level sync method with applications running in a different partition than the base Linux distribution that boots the machine and interacts with hardware. This experience report first gives insights on how the distribution is setup, and then tackles the problem of doing a difficult upgrade from a Red Hat 7.1 image snapshot with layers of patches to a Debian Testing based distribution built from source. We will look at how this can actually be achieved as a live upgrade and without ending up with a long "flag day" where many machines are running totally different distributions, which would have made testing and debugging of applications disastrous during a long switchover period.

Like a coworker of mine put it, "It was basically akin to upgrading Red Hat 7.1 to Fedora Core 16, a totally unsupported and guaranteed to break upgrade, but also switching from rpm to dpkg in the process, and on live machines."

The end of the paper summarizes how we designed our packaging system for the new distribution, as well as how we build each new full distribution image from scratch in a few minutes.

Tags: infrastructure, Linux, distribution, live upgrade

Introduction

The Linux operating system that Google uses in our service "production" environment has a strange history which will be described before explaining how we upgraded it.

Google's production Linux OS is managed in three layers. The kernel and device drivers, user-space, and the running applications.

The kernel and device drivers are updated frequently and separately from the operating system. These files are maintained, fleet-wide, by a different team. Aside from obvious touch points, this maintenance is unrelated to the work described in this paper.

Each application runs in a chroot-ed jail. This jail includes all the programs, shared libraries, and data files required for the application to run. Therefore they are not entangled with the rest of the operating system. This independence from the underlying operating system is fairly extreme: even external libraries are statically linked. We provide multiple hermetic versions of python, the C++ libraries, the C runtime loader and libraries that applications can choose from. These are all decoupled from the booted operating system.

The remaining part is the user-space files - the init scripts, the /usr/bin binaries, and so on. The OS's native package system is only used for this part, which is the focus of this paper.

Because of this decoupling the user-space portion could go a long time without upgrades. In fact, it remained at the equivalent of Red Hat 7.1 for many years.

Changing a fleet of thousands of machines from one distribution to another is a rare event and there is no "best practice" for doing so. One could convert small groups of machines until the entire fleet is converted. During the transition the fleet would contain two different operating systems. That was unacceptable - the entire Google fleet is kept within one or two minor OS revisions at any given time. Adding multiple operating systems would have multiplied complexity.

Instead we chose to transition parts of the OS one at a time: the boot scripts, the user-space binaries, the package system, etc. Over 3 years the entire OS would change, sometimes file by file, until it was completely replaced. This permitted each little step to be fully tested and possibly reverted. Most importantly users would not see a "flag day" change. At our a large scale, a small error is multiplied by thousands of machines. The ability to move slowly, cautiously, and with large amounts of testing, was critical.

System Administrators often refer to their work as "changing the tires while the car is driving down the highway". In this case we changed the front left tire across the entire fleet. Once that was done we changed the steering wheel across the entire fleet. This process continued and after four years we had an entirely new car.

1. Google Servers and Linux, the early days

Like many startups, Google started with a Linux CD. It started around 1998 with a Red Hat 6.2 that was installed on the production machines. Soon thereafter, we got a kickstart network install, and it grew from there.

Updates and custom configurations were a problem. Machine owners had ssh loops to connect to machines and run custom install/update commands. At some point, they all got reinstalled with Red Hat 7.1 with custom software re-installed on top, but obviously this was not the right way to do things.

1.1. Better update management

The custom ssh loops were taking longer to run, and missing more machines each time. It was quick and dirty, but this has never scaled. Generally any push based method is doomed.

Now, it's not uncommon to run apt-get or yum from cron and hope updates will mostly work that way. However, for those of you who have tried running apt-get/dpkg/rpm/yum on thousands of servers, you may have found that random failures, database corruptions (for rpm) due to reboots/crashes during updates, and other issues make this not very reliable.

Even if the package DBs don't fail, it's often a pain to deal with updates to config files conflicting with packages, or unexpected machine state that breaks the package updates and causes all subsequent updates to fail until an admin fixes the machine manually, or a crude script simply wipes and re-installs the machine. The first method doesn't scale and the second one can cause data loss and outages.

1.2. Full file level filesystem sync

As crude as it is, file level syncing recovers from any state and bypasses package managers and their unexpected errors. It makes all your servers the same though, so custom packages and configs need to be outside of the synced area or manually excluded. Each server then has a list of custom files (network config, resolv.conf, syslog files, etc...) that are excluded from the sync.

Now, using rsync for entire machines off a master image doesn't scale well on the server side, and can bog the I/O on your clients, causing them to be too slow to serve requests with acceptable latency. You also need triggers that restart programs if certain files change.

So, we wrote custom rsync-like software where clients initiate file level syncs from a master image. It then allows for shell triggers to be run appropriately. IO is throttled so that it does not negatively impact machines serving live requests while they are being upgraded.

1.3. Isolating server packages from the Server OS

We have custom per machine software that is outside of the centrally managed root partition, and therefore does not interfere with updates. In other words, the distribution is a fancy boot loader with housekeeping and hardware monitoring tools. Applications go in a separate partition and are not allowed to touch the dpkg/rpm database, or modify the root partition in any other way.

The software run by the server is typically run in a chroot with a limited view of the root partition, allowing the application to be hermetic and protected from root filesystem changes. We also have support for multiple libcs and use static linking for most library uses. This combination makes it easy to have hundreds of different apps with their own dependencies that change at their own pace without breaking if the OS that boots the machine changes.

The limited view of the root partition was achieved by first having a blacklist of what not to include in the chroot for applications, and later transitioning to a whitelist. In other words, our restricted chroot for user applications only contains files that have been opted in.

This upgrade itself also gave us a chance to find places where we weren't fully hermetic like we should have been.

2. How we did updates

Because had decoupled the booting OS from the applications running on top, the actual OS saw a minimal amount of updates. Updates were mostly security updates for bugs that did potentially affect us. From time to time we also needed a new feature that was added in the userland tools that we used. In other words OS updates were few and far in between and done only on demand, . This is how we ended up still running something that was still mostly Red Hat 7.1 after about 10 years, managed by 2 or fewer people. In some ways, we pushed the “if it ain't broke, don't fix it” motto as far as we could.

2.1. Server image updates

We effectively had a filesystem image that got synced to a master machine, new packages were installed and the new image was snapshotted. We had scripts to store the new filesystem snapshot in Perforce, one of our source control systems, and allow for crude diffing between the two images. The new golden image was then pushed to test machines, had to pass regression tests, and pushed to a test cluster, eventually with some live traffic. When the new image has seen enough testing, it is pushed slowly to the entire fleet.

2.2. Dealing with filesystem image updates

After dealing with the obvious issues of excluding machine specific config files, and logs from full filesystem syncs, the biggest difference is dealing with package postinstalls. We removed most of them since anything that is meant to run differently on each machine doesn't work with a golden image that is file-synced.

Examples:

- Running ldconfig after a library change is ok.
- Creating files in postinstall works, but is undesirable since those don't show up in the package file list.
- For the case of files like ssh host keys, it's obviously bad to create a single host key that gets snapshotted and synced everywhere.
- Re-running lilo after updating lilo.conf would not work.
- Restarting daemons doesn't work either.
- Many postinstalls have code to deal with cleaning up for upgrades that weren't relevant to us, so they could be ignored or removed.

We dealt with postinstalls that were necessary on a case by case basis and we used our filesystem sync post push triggers that restart daemons or re-install lilo boot blocks after the relevant config files or binaries were updated.

2.3. Testing software before doing image updates

We wrote a test-rpm-install/test-deb-install script that takes a clean machine, installs the package, and gets a before/after snapshot of the entire filesystem. This allowed us to verify what gets added/removed to the filesystem, as well as review unix permission changes, and size increases. We always fought software bloat, which is how we managed to keep a small boot image after years of evolution (it actually shrunk in size over time as bloat was identified and removed).

Software engineers of course have mandated code reviews and unit tests for their software. Once those are done for a change, we build an image with just the new package and send it to our regression tester. The regression tester runs on a sample of our different platforms, applies the update without rebooting, and ensures that critical daemons and services continue to work after the update. Once that works, the machine is rebooted, the services checked again, after which the machine is rebooted first cleanly, and then a second time as a crash reboot. If this all passes, the image is then reverted, we make sure daemons do not misbehave when downgraded (this can happen if the old code cannot deal with state files from the new code), and the downgraded image is then rebooted to make sure everything comes back up as expected.

While this test suite is not foolproof, it has found a fair amount of bugs, and ideally let the software submitter find problems before submitting the package for inclusion in the next image cut.

2.4. Reviewing image updates before deployment, and test deployment.

We start with the old image's files checked into Perforce (Perforce was mostly chosen because it was our main already in use source control system at the time). Metadata was stored into a separate file that wasn't much reviewable (dev nodes, hardlinks, permissions, etc...), but we had a reviewer friendly ls -alR type file list to review permission and owner changes.

Image build input was a list of pre-approved packages to update with package owners providing their own testing notes, and features they're looking at adding. They got installed on a test machine, and the output was a new filesystem image where Perforce allowed reviewing diffs of ASCII files, and we could review changes in binary sizes as well as file permissions. From there, if approved, the image was sent to a pool of early test machines, and deployed slowly fleet-wide if no one complained about regressions.

2.5. Admin and debugging considerations

While the focus of this paper is on distribution management and upgrades, there are a few things worth noting related to management of headless machines and debugging boot issues. We have serial consoles on some test machines, but it's not worth the price on all machines. As a result we use bootlogd to capture boot messages without requiring the much heavier and buggy plymouth. We also start a debug sshd before the root filesystem is fsck'ed and remounted read-write. That way we can easily probe/debug machines that aren't rebooting properly or failing to fsck their root filesystem.

When you have so many machines, you want to keep the init system simple and dependable. Whenever possible we want all our machines to behave the same. As a result, we stuck with normal init, and looked at Debian's insserv and startpar for simple dependency booting that we can set in stone and review at image creation time. Both upstart and systemd require way too many moving pieces and introduce boot complexity and unpredictability that was not worth the extra boot time they could save.

While shaving a few seconds of boot isn't that important to us, we do save reboot time by avoiding double reboots when the root filesystem needs repair, and do so by doing a pivot-root to an initramfs with busybox, release the root filesystem, fsck it, and then pivot-root back to it to continue normal boot.

2.6 This worked amazingly well over time, but it had many issues

- Like is often the case, our system was not carefully thought out and designed from the ground up, but just a series of incremental "we have to fix this now" solutions that were the best the engineers with limited time could do at the time.
- Our entire distribution was really just a lot of overlayed patches on top of a Red Hat 7.1 live server snapshotted almost 10 years ago.
- A lot of software was still original Red Hat 7.1 and we had no good way to rebuild it on a modern system. Worse, we just assumed that the binaries we had were indeed built from the original Red Hat source.
- The core of our distribution was now very old, and we knew we couldn't postpone upgrading it forever, but had no good plan for doing so.

3.0. Upgrade Plan

3.1 Which distribution?

Back in the days, we wasted too much time building open source software as rpms, when they were available as debs. As a result, we were not very attached to Red Hat due to the lack of software available in rpm form vs what was available in Debian. We had already switched away from Red Hat on our Linux workstations years prior for the same reason (our workstations are running a separately maintained linux distribution because they have different requirements and tradeoffs than our servers do) Back then, Red Hat 9 had 1,500 packages vs 15,000 in Debian. Today Fedora Core 18 has 13,500 vs 40,000 in Debian testing. Arguably Red Hat fares better today than it did then, but still remains inferior in software selection.

As a result, ProdNG, our Linux distribution built from source, was originally based off Ubuntu Dapper. At the time Ubuntu was chosen because we were also using it on our workstations. Later on, we switched to straight Debian due to Ubuntu introducing several forced complexities that were not optional and not reliable when they were introduced, like upstart and plymouth.

3.2 ProdNG Design

Richard Gooch and Roman Mitnitski, who did the original design for the new distribution came up with these design points to address the limitations of our existing distribution:

- Self hosting.
- Entirely rebuilt from source.
- All packages stripped of unnecessary dependencies and libraries (xml2, selinux library, libacl2, etc..)
- Less is more: the end distribution is around 150MB (without our custom bits). Smaller is quicker to sync, re-install, and fsck.
- No complicated upstart, dbus, plymouth, etc. Tried and true wins over new, fancy and more complex, unless there is measurable benefit from the more complex version.
- Newer packages are not always better. Sometimes old is good, but only stay behind and fork if really necessary. On the flip side, do not blindly upgrade just because upstream did.
- Hermetic: we create a ProdNG chroot on the fly and install build tools each time for each new package build.
- Each image update is built by reassembling the entire distribution from scratch in a chroot. This means there are no upgrades as far as the package management is concerned, and no layers of patches on top of a filesystem that could have left-over forgotten cruft.

3.3 Upgrade Plan

Once we had a ProdNG distribution prototype that was booting, self-hosting, and ready to be tested, we all realized that switching over would be much harder than planned.

There was no way we could just roll out a brand new distribution that was 100% different from the old one, with software that was up to 10 years newer, and hope for the best. On top of that, our distribution contains our custom software that is required for new hardware bringup, or network changes, so we could not just have paused updates to the old distribution for months while we very slowly rolled out the new one. Cycles of find a bug, pause the rollout or revert, fix the bug (either in the distribution, or in the software that relied on the behavior of the old one), and try again, could have potentially lasted for months.

It could have been possible with a second team to maintain the old production image in parallel and at each review cycle build 2 distributions, but this had many problems. To list just a few:

- Double the review load, but it was obviously not desirable, nor really achievable with limited staffing.
- Would we really want to have a non-uniform setup in production for that long? That's not going to make debugging easy in case we notice failures in production and for months we'd now first have to worry about whether “Is it a ProdNG related problem, or a distribution independent problem?”. Our monitoring tools expect the same distribution everywhere, and weren't designed to quickly categorize errors based on ProdNG or not ProdNG. This could have been done with a lot of work, but wasn't deemed a good use of time when there was a better alternative (explained below).
- With one distribution made with rpms, while the other one is dpkg, using totally different build rules and inter package dependencies, our package owners would also have a lot more work.
- While it's true that we have few internal users who depend on the distribution bits, that small number, from people working on the installers, and people writing software managing machine monitoring, hardware, and software deployment are still a sizeable amount of people (more than just a handful we can sync with or help individually if we change/break too many things all at once).

One motto at Google is that one team should not create a lot of work for other teams to further their own agenda, unless it's absolutely unavoidable and the end goal is worth it. At the time, we were not able to make a good enough case about the risk and work we would have introduced. In hindsight, it was a good call, the switch if done all at once, would have introduced way too many problems that were manageable handled one by one over time, but not as much if thrown around all the same time.

Around that time, Roman had to go back to another project, and with no good way to push ProdNG forward due to the risk of such a big change, and impact on other internal teams, it stalled.

3.4 The seemingly crazy idea that worked

Later, at the time I joined the team working on the production image, Richard Gooch and I sat down to list the requirements for a successful upgrade:

- We need to keep all the machines in a consistent state, and only stay with 2 images: the current/old one and the new one being pushed.
- If flag day there must be, it must be as short a day as possible.
- Service owners should not notice the change, nor should their services go down.
- rpm vs dpkg should be a big switch for us, the maintainers, but not the server users.
- There are just too many changes, from coreutils to others, for the jump to be small enough to be safe.
- And since we can't have a big jump, we can't jump at all.

Richard came up with the idea of slowly feeding our ProdNG distribution into our existing production image, a few packages at a time during each release cycle. Yes, that did mean feeding debs into an rpm distro.

To most, it likely sounded like a crazy idea because it was basically akin to upgrading Red Hat 7.1 to Fedora Core 16, a totally unsupported and guaranteed to break upgrade, but also switching from rpm to dpkg in the process, and on live machines.

An additional factor that made this idea “crazy” is that our ProdNG packages were based on libc 2.3.6 whereas our production image was based on libc 2.2.2, thus ProdNG binaries would simply not run on the old image, and it was unsafe to upgrade the system libc without recompiling some amount of its users. Richard had a key insight and realized that binary patching the ProdNG binaries would allow them to run on the old image. Since the original ProdNG prototype was developed and shelved, the production image had acquired a hermetic C library for the use of applications outside of the OS (this allowed applications to be use a libc, and later among several available, without relying on the one from the OS).

At the time, his hermetic C library was also based on libc 2.3.6 and thus ProdNG binaries could use it as long as the run-time linker path in the ELF header was binary patched with a pathname of the same length.

Since doing unsupported live upgrades has been a side hobby of mine since Red Hat 2.1, including switching binary formats from zmagic to qmagic (libc4), then ELF with libc5, and finally glibc with libc6, I didn't know how long it would take, but I figured this couldn't be any worse and that I could make it happen.

3.5 Implementing the slow upgrade

By then, ProdNG was still self hosting, and could build new packages, so Richard wrote an alien(1) like package converter that took a built ProdNG package and converted it to an rpm that would install on our current production image (this did include some sed hackery to convert dependency names since Debian and Red Hat use different package names for base packages and libraries that are required for other packages to install), but overall it was not that complicated. The converter then ran the binary patcher described above, and ran an ugly script I wrote to turn Debian changelogs into Red Hat ones so that package upgrades would show expected changelog diffs for the reviewers.

Because by the time I joined to help the production image group, the ProdNG build had been stale for a couple of years, I started by refreshing ProdNG, and package by package, upgrading to more recent source if applicable, stripping all the new features or binaries we didn't need, and feeding the resulting package as a normal rpm package upgrade in the next production image release.

From there, I looked at our existing binaries, and checked whether they would just work if libc was upgraded and they weren't recompiled. Most passed the test without problem, while a few showed

```
Symbol `sys_siglist' has different size in shared object, consider re-linking
```

The other issue was that some binaries were statically linked, and those have hardcoded pathnames to libnss libraries, which were the ones we were trying to remove. Having non matching libc and libnss also caused those binaries to fail, which wasn't unexpected. This problem was however quickly solved by removing the old libc altogether and repointing ld-linux.so to the new libc. I then added a few symlinks between the location of the libnss libs from the old libc to the ones from the new libc.

Note that we had to run with this dual libc configuration for a while since we still had a self imposed rule of only upgrading a few packages at each cycle. Therefore we pushed fixed packages a few at a time until we were ready one day to remove the old libc and replace it with symlinks to the new one.

3.6 If you can delete it, you don't have to upgrade it

Despite of the fact that our image was a snapshot of a live Red Hat 7.1 server install, it contained a lot of packages that didn't belong in a base server install, or packages that we didn't need for our uses.

Distributions with crazy dependency chains have only been getting worse over time, but even in the Red Hat 7.1 days, dependencies in Red Hat were already far from minimal. Some were pure cruft we never needed (X server, fonts, font server for headless machines without X local or remote, etc...). Next, I looked for all things that made sense to ship as part of RH 7.1, but were useless to us (locales and man pages in other languages, i18n/charmaps, keyboard mappings, etc...).

After that, I looked for the next low hanging fruit and found libraries nothing was using anymore (left over from prior upgrade, or shipped by default, but not used by us). For some libraries, like libwrap, I was able to remove them after upgrading the few packages that used them, while omitting the library from their builds.

When it was all said and done, I had removed 2/3rd of the files we had in the initial image, and shed about 50% of the disk space used by the Linux image (not counting our custom in-house software).

3.7 The rest of the upgrade

What didn't get deleted, had to be upgraded however. Once the libc hurdle was past, it was a lot of painstaking work to deal with each weird upgrade differently, and qualify each big software jump for things like cron, or syslog, to be sure they would be safe and not fix a bug that we were relying on. Just upgrading rsync from 2.x to 3.x took 4 months of work because of semantics that changed in the code in how it handled permission syncs, and our dependence on the old behavior.

Our distribution was so old that it didn't even have coreutils. It had fileutils + textutils + sh-utils, which got replaced with fairly different binaries that unfortunately were not backward compatible so as to be more POSIX compliant. Upgrading just that took a lot of effort to scan all our code for instances of tail +1, or things scanning the output of ls -l. In the process, multiple utilities got moved from /bin to /usr/bin, or back, which broke some scripts that unfortunately had hardcoded paths.

Aside from a couple of upgrades like coreutils, there weren't too many upgrades with crazy dependency chains, so it was not a problem to upgrade packages a few at a time (5 to 10 max each time).

On some days, it was the little things. The day I removed /etc/redhat-release, it broke a bunch of java code that parsed this file to do custom things with fonts depending on the presence of that file. At Google, whoever touched something last is responsible for the breakage, even if the bug wasn't in that change, so that typically meant that I had to revert the change, get the right team to fix the bug, wait for them to deploy the fix on their side, and then try again later.

3.8 Dealing with left over junk

Because our original image was a full filesystem image that got snapshotted in Perforce, we ended up with files that were not runtime created, and not part of any package either. We had junk that we didn't always know the source of, or sometimes whether it was safe to remove.

I ended up finding the expected leftover files (.rpmsave, old unused files), lockfiles and logfiles that shouldn't have been checked in and /etc/rcxx initscript symlinks. Any actual program that wasn't part of a package, was identified and moved to a package.

Then, I had to scan the entire filesystem for files that were not in a package, work through what was left on the list and deal with the entries on an case by case basis.

That said, the goal was never to purge every single last trace of Red Hat. We have some Red Hat pathnames or functions left over to be compatible with things that expect Red Hat and aren't quite LSB compliant. We only removed Red Hat specific bits (like rpm itself) when it was simple to do so, or because maintaining them long term was going to be more work than the cost of removal.

3.9 A difficult problem with /etc/rc.d/...

Back in the day (mid 1990's) someone at Red Hat misread the linux filesystem standard and put the initscripts in /etc/rc.d/rc[0-6].d and /etc/rc.d/nit.d instead of /etc/rc[0-6].d, as implemented in other linux distributions including Debian. Migrating to Debian therefore included moving from /etc/rc.d/init.d to /etc/init.d.

Unfortunately I found a bug in our syncing program when switching from /etc/rc.d/init.d (Red Hat) to /etc/init.d (Debian): when the image syncer applied a new image that had /etc/init.d as the directory and /etc/rc.d/init.d as the compatibility symlink, that part worked fine, but then it also remembered that /etc/rc.d/init.d was a directory in the old image that got removed, and by the time it did a recursive delete of /etc/rc.d/init.d, it followed the /etc/rc.d/init.d symlink it had just created and proceeded to delete all of /etc/init.d/ it also had just created.

The next file sync would notice the problem and fix it, but this left machines in an unbootable state if they were rebooted in that time interval and this was not an acceptable risk for us (also the first file sync would trigger restarts of daemons that had changed, and since the initscripts were gone, those restarts would fail).

This was a vexing bug that would take a long time to fix for another team who had more urgent bugs to fix and features to implement. To be fair, it was a corner case that no one had ever hit, and no one has hit since then.

This was a big deal because I had to revert the migration to /etc/init.d, and some of my coworkers pushed for modifying Debian forever to use /etc/rc.d/init.d. Putting aside that it was a bad hack for a software bug that was our fault, it would have been a fair amount of work to modify all of Debian to use the non standard location, and it would have been ongoing work forever for my coworkers after me to keep doing so. I also knew that the changes to initscripts in Debian would force us to have local patches that would cause subsequent upstream changes to conflict with us, and require manual merges.

So, I thought hard about how to work around it, and I achieved that by keeping Debian packages built to use /etc/init.d, but by actually having the real filesystem directory be /etc/rc.d/init.d while keeping /etc/init.d as a symlink for the time being. This was done by setting those up before Debian packages were installed in our image builder. Dpkg would then install its files in /etc/init.d, but unknowing follow the symlink and install them in /etc/rc.d/init.d.

This was ok, but not great though because we'd have a mismatch between the Debian file database and where the files really were on disk, so I worked further to remove /etc/rc.d/init.d.

We spent multiple months finding all references to /etc/rc.d/init.d, and repoint them to /etc/init.d. Once this was finished, we were able to remove the image build hack that created /etc/rc.d/init.d.

The bug did not trigger anymore because our new image did not have a /etc/rc.d/init.d compatibility symlink, so when the file syncer deleted the /etc/rc.d/init.d directory, all was well.

3.10 Tracking progress

Our converted ProdNG packages had a special extension when they were converted to RPMs, so it was trivial to use rpm -qa, look at the package names and see which ones were still original RPMs and which ones were converted debs.

I then used a simple spreadsheet to keep track of which conversions I was planning on doing next, and for those needing help from coworkers who had done custom modifications to the RPMs, they got advance notice to port those to a newer Debian package, and I worked with them to make a ProdNG package to upgrade their old RPM. They were then able to monitor the upgrade of their package, and apply the relevant tests to ensure that the package still did what they needed. This allowed porting our custom patches and ensuring that custom packages were upgraded carefully and tested for their custom functionality before being deployed (we do send out patches upstream when we can, but not all can be accepted).

3.11 Communication with our internal users

We used different kinds of internal mailing lists to warn the relevant users of the changes we were about to make. Some of those users were the ones working on the root partition software, others were our users running all the software that runs google services inside the chroots we provide for them, and we also warned the people who watch over all the machines and service health when we felt we were making changes that were worth mentioning to them.

All that said, many of those users also had access to our release notes and announcements when we pushed a new image, and quickly knew how to get image diffs when debugging to see if we made an image change that might have something to do with a problem they are debugging.

4.0 Getting close to swichover time

After almost 3 years of effort (albeit part time since I was also working on maintaining and improving the current rpm based image, working with our package owners, as well as shepherding releases that continued to go out in parallel), the time came when everything had been upgraded outside of /sbin/init and Red Hat initscripts.

Checking and sometimes modifying Debian initscripts to ensure that they produced the same behavior that we were getting from our Red Hat ones took careful work, but in the end we got ProdNG to boot and provide the same environment as our old Red Hat based image. To make the migration easier, I fed shell functions from Red Hat's /etc/init.d/functions into Debian's /lib/lsb/init-functions and symlinked that one to /etc/init.d/functions. This allowed both Red Hat and Debian initscripts from 3rd party packages to just work.

4.1 Reverse conversions: rpms to debs

By then, a portion of our internal packages had been converted from rpms to debs, but not all had been, so we used reverse converter that takes rpms, and converts them to debs, with help from alien. The more tedious part was the converter I wrote to turn mostly free form Red Hat changelogs into Debian changelogs which have very structured syntax (for instance, rpms do not even require stating which version of the package a changelog entry was for, and if you do list the version number, it does not check that the latest changelog entry matches the version number of the package). Rpm changelogs also do not contain time of day, or timezones (I guess it was Raleigh-Durham Universal Time), so I just had to make those up, and problems happen if two rpm releases happened on the same day with no time since it creates a duplicate timestamp in the debian changelog. Some fudging and kludges were required to fix a few of those.

4.2 Time to switch

By then, ProdNG was being built in parallel with the rpm production image and they were identical outside of initscripts, and rpm vs dpkg. With some scripting I made the ProdNG image look like a patch image upgrade for the old image, and got a diff between the two. We did manual review of the differences left between 2 images (file by file diff of still 1000+ files). There were a few small differences in permissions, but otherwise nothing that wasn't initscripts or rpm vs dpkg database info.

It then became time to upgrade some early test machines to ProdNG, make sure it did look just like the older image to our internal users, and especially ensure that it didn't have some bugs that only happened on reboot 0.5% of the time on just one of our platforms. Then, it started going out to our entire fleet, and we stood around ready for complaints and alerts.

4.3 Switch aftermath

Early deployment reports found one custom daemon that was still storing too much data in /var/run. In Red Hat 7.1, /var/run was part of the root filesystem, while in ProdNG it was a small tmpfs. The daemon was rebuilt to store data outside of /var/run (we have custom locations for daemons to write bigger files so that we can control their sizes and assign quotas as needed, but this one wasn't following the rules).

Most of the time was actually spent helping our package owners convert their rpms to debs and switching to new upload and review mechanisms that came with ProdNG since the image generation and therefore review were entirely different.

As crazy as the project sounded when it started, and while it took awhile to happen, it did. Things worked out beautifully considering the original ambition.

4.4 Misc bits: foregoing dual package system support

We had code to install rpms unmodified in our ProdNG deb image, and even have them update the dpkg file list. We however opted for not keeping that complexity since dual package support would have rough edges and unfortunate side effects. We also wanted to entice our internal developers to just switch to a single system to make things simpler: debs for all. They are still able to make rpms if they wish, but they are responsible for converting them to debs before providing them to us.

As a side result, we were able to drop another 4MB or so of packages just for just rpm2cpio since rpm2cpio required 3-4MB of dependencies. I was able to find a 20 line shell script replacement on the net that did the job for us. This allowed someone to unpack an old legacy rpm if needed while allowing me to purge all of rpm and its many libraries from our systems.

Debian made a better choice by having an archive system that can be trivially unpacked with ar(1) and tar(1) vs RPM that requires rpm2cpio (including too many rpm libraries) and still loses some permissions which are saved as an overlay stored inside the RPM header and lost during rpm2cpio unpacking.

4.5 No reboots, really?

I stated earlier that the upgrades we pushed did not require to reboot servers. Most services could just be restarted when they got upgraded without requiring a reboot of the machine.

There were virtually no times where we had code that couldn't be re-exec'ed without rebooting (even /sbin/init can re-exec itself), that said our servers do get occasionally rebooted for kernel ugprades done by another team, and we did benefit from those indirectly for cleaning up anything in memory, and processed that didn't restart, if we missed anyway.

5.0 ProdNG Design Notes

While it's not directly related to the upgrade procedure, I'll explain quickly how the new image is designed.

5.1 ProdNG package generation

Since one of the goals of our new distribution was to be self-hosting, and hermetic, including building a 32bit multiarch distribution (32bit by default, but with some 64bit binaries), it made sense to build ProdNG packages within a ProdNG image itself. This is done by quickly unpacking a list of packages provided in a dependencies file (mix of basic packages for all builds, and extra dependencies you'd like to import in that image to build each specific package). Debian provides pbuilder which also achieves that goal, but our method of unpacking the system without using dpkg is much faster (1-2 minutes at most), so we prefer it.

We use the debian source with modifications to debian/rules to recompile with fewer options and/or exclude sub-packages we don't need. We then have a few shell scripts that install that unpacked source into a freshly built ProdNG image, build the package, and retrieve/store the output. You get flexibility in building a package in an image where for instance libncurses is not available and visible to configure, while being present in the image currently deployed (useful if you'd like to remove a library and start rebuilding packages without it).

After package build, we have a special filter to prune things we want to remove from all packages (info pages, man pages in other languages, etc...) without having to modify the build of each and every package to remove those. The last step is comparing the built package against the previous package, and if files are identical, but the mtime was updated, we revert the mtime to minimize image review diffs later.

5.2 ProdNG image generation

This is how we build our images in a nutshell: each new image to push is generated from scratch using the latest qualified packages we want to include into it (around 150 base Linux packages).

The image is built by retrieving the selected packages, unpacking them in a chroot (using ar and untar), and chrooting into that new directory. From there, the image is good enough to allow running dpkg and all its dependencies, so we re-install the packages using dpkg, which ensures that the dpkg database is properly seeded, and the few required postinstall scripts do run. There are other ways to achieve this result (debootstrap), but because our method runs in fewer than 10 minutes for us and it works, we've stuck with it so far.

As explained, package builds revert mtime only changes, and squash binary changes due to dates (like gzip of the same man page gives a new binary each time because gzip encodes the time in the .gz file). We have a similar patch for .pyc files. As a result of those efforts, rebuilding an image with the same input packages is reproducible and gives the same output.

5.3 ProdNG image reviews

The new ProdNG images are not checked in Perforce file by file. We get a full image tar.gz that is handed off to our pusher and reviews are done by having a script unpack 2 image tars, and generate reviewable reports for it:

- file changes (similar ls -alR type output)
- which packages got added/removed/updated
- changelog diffs for upgraded packages
- All ASCII files are checked into Perforce simply so that we can track their changes with Perforce review tools.
- compressed ASCII files (like man pages or docs) are uncompressed to allow for easy reviews.
- Other binary files can be processed by a plugin that turns them into reviewable ASCII.

6. Lessons learned or confirmed.

1. If you have the expertise and many machines, maintaining your own sub Linux distribution in house gives you much more control.
2. At large scales, forcing server users to use an API you provide, and not to write on the root FS definitely helps with maintenance.
3. File level syncing recovers from any state and is more reliable than other methods while allowing for complex upgrades like the one we did.
4. Don't blindingly trust and install upstream updates. They are not all good. They could conflict with your config files, or even be trojaned.
5. If you can, prune/remove all services/libraries you don't really need. Fewer things to update, and fewer security bugs to worry about.
6. Upgrading to the latest Fedora Core or Ubuntu from 6 months ago is often much more trouble than it's worth. Pick and chose what is worthwhile to upgrade. Consider partial upgrades in smaller bits depending on your use case and if your distribution is flexible enough to allow them.
7. Prefer a distribution where you are in control of what you upgrade and doesn't force you into an all or nothing situation. Ubuntu would be an example to avoid if you want upgrade flexibility since it directly breaks updates if you jump intermediate releases. Debian however offers a lot more leeway in upgrade timing and scope.
8. Keep your system simple. Remove everything you know you don't need. Only consider using upstart or systemd if you really know how their internals, possible race conditions, and are comfortable debugging a system that fails to boot.

References

As you may imagine, we didn't really have much existing work we were able to draw from. Alien(1) from Joey Hess definitely helped us out for the rpm to deb conversions, and here is the URL to the rpm2cpio I found to replace our 4MB of binaries:

https://trac.macports.org/attachment/ticket/33444/rpm2cpio

Acknowledgements

I need to single out my coworkers who helped design ProdNG: Richard Gooch and Roman Mitnitski for designing and implementing the first ProdNG images built from source and self-hosting. Richard Gooch also gets the credit for coming up with the initial design of feeding ProdNG in our existing file-synced image package by package by converting ProdNG debs to rpms, as well as reviewing this paper.

Joey Hess, Debian maintainer of debhelpers and man other packages related to dpkg, is also the maintainer of alien(1). He was helpful on multiple occasions with dpkg/debhelper related questions when I was doing complicated conversions. Joeh also helped me include a cleaner version of a patch I had to write to properly convert some unix ownership and permissions from rpm to deb that alien(1) was failing to conver at the time.

Next, I would like to thank Tom Limoncelli for reviewing this paper as well as encouraging me to write it in the first place. He also gets credit for contributing the introduction which he was able to write with an outsider's point of view.

Finally, I owe thanks to the LISA reviewers for their review comments, and specifically Andrew Lusk for a careful and thorough final review of this paper.

Of course, I also need to thank all the coworkers at Google who helped me in various ways with this project, or doing our day to day maintenance and release work.

Availability

This document is available at the USENIX Web site.