Lisa 2013

Case Study

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

http://marc.merlins.org/linux/talks/ProdNG-LISA/

Marc MERLIN
marc_soft@merlins.org

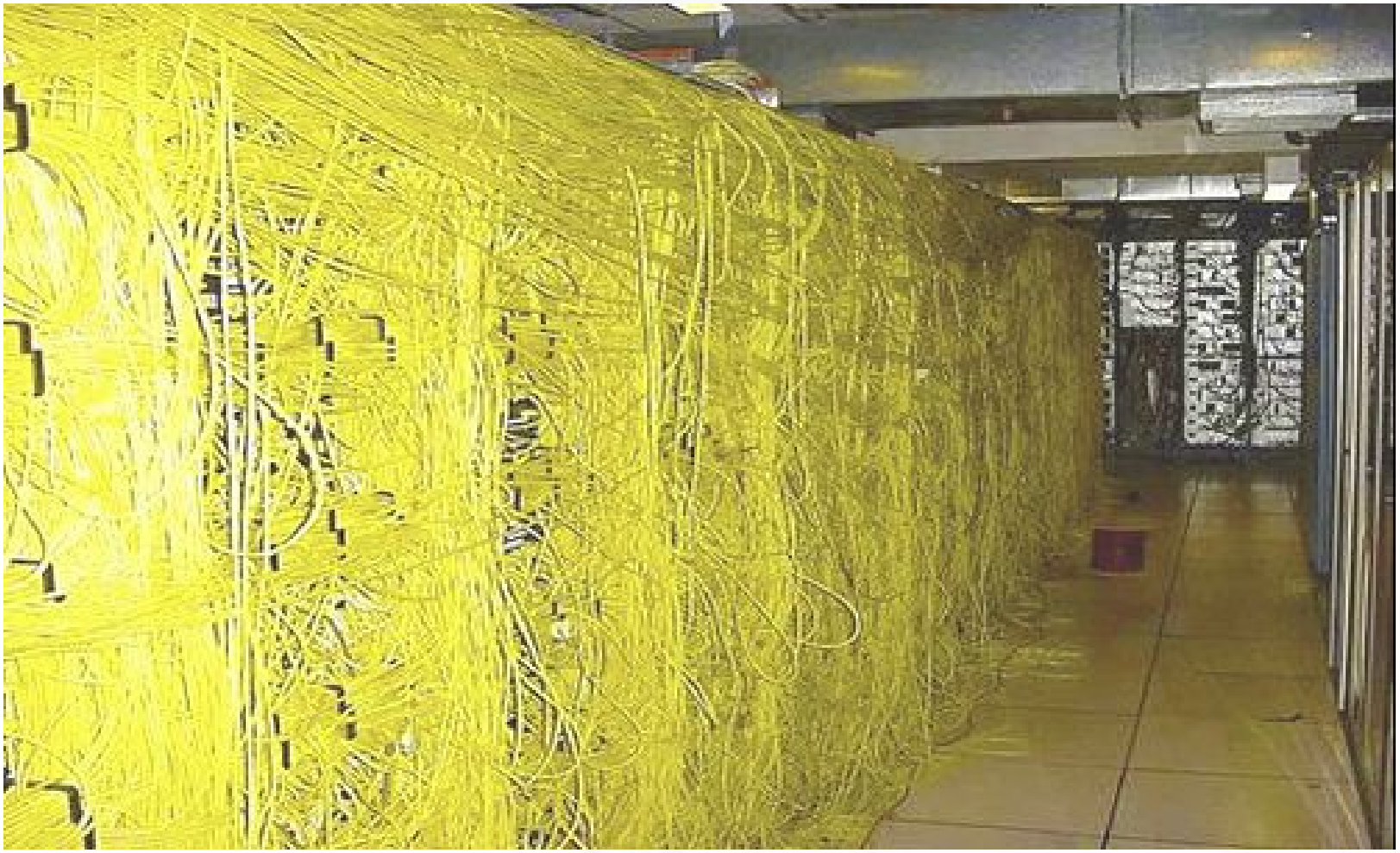
Google Servers and Linux, the early days

- Like many startups, Google started with a Linux CD (in our case Red Hat 6.2) installed on a bunch of machines (around 1998).
- Soon thereafter, we got a kickstart network install
- Updates and custom configurations were a problem though
- Machine owners had ssh loops to connect to machines and run custom install/update commands
- No, it's not scalable :)
- Eventually, they all got reinstalled with Red Hat 7.1

But, what about updates?

- Ah yes, updates... That ssh loop sure was taking longer to run, and missing more machines each time.
- Any push based method is doomed. If you're triggering updates by push, you're doing it wrong :)
- Now, it's common to run apt-get or yum from cron and hope updates will mostly work that way.
- 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 database doesn'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.

Do you ever feel like you have this?



When you really wanted this?

File level syncing, really?

- 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.
- Each server then has a list of custom files (network config, resolv.conf, syslog files, etc...) that are excluded from the sync.
- Rsync for entire machines off a master image doesn't scale well on the server side, and can bog the IO 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 that basically does file level syncs of all our servers from a master image and allows for shell triggers to be run appropriately.
- IO is throttled so that it does not negatively impact machines serving requests.

Wait, all your servers are the same?

- For the root partition, pretty much, yes.
- Custom per machine software is outside of the centrally managed root partition, and therefore does not interfere with updates.
- Software run by the server can be run in a chroot with a limited view of the root partition, allowing the application to be hermetic and therefore protected from changes on the root partition.
- We also have support for multiple libcs and use static linking for most library applications.
- This combination makes it easy to have hundreds of different apps with their own dependencies that change at their own pace.

How did you do big OS upgrades on the root partition?

- Because of our hermetic setup for apps, we mostly didn't need to upgrade the base OS, outside of security updates
- As a result, we ended up running a Red Hat 7.1 derivative for a very long time ;)

Ok, so how do you upgrade base packages?

- We effectively had a filesystem image that got synced to a master machine, new rpms were installed and the new image was then snapshotted.
- The new golden image can then be pushed to test machines, pass regression tests, and then pushed to a test colo, eventually with some live traffic.
- When the new image has seen enough testing, it is pushed slowly to the entire fleet.
- Normally, 2 images: the current/old one and the new one.

How about package pre/post installs?

- Most pre/post installs were removed since anything that is meant to run differently on each machine doesn't work with a golden image that is file-synced.
- Running ldconfig after a library change is ok.
- Re-running lilo after updating lilo.conf would not work.
- Restarting daemons doesn't work either.
- This is where we use our post push triggers that will restart daemons or re-install lilo boot blocks after the relevant config files or binaries were updated.

How did that strategy of patching Red Hat 7.1 work out?

- Turns out no one gets a medal for running a very old distro very long :)
- Making new RPMs to update a distribution from the last millennium was not a long term strategy, even if it worked for over 10 years.
- We needed something new.
- But doing an upgrade to a new distribution over 10 years later is scary.
- Very scary!
- Oh, and preferably on live machines without rebooting them :) (we reboot for kernel upgrades, but that happens asynchronously)

So, what new Linux distribution?

- We already switched away from Red Hat on our Linux workstations years prior due to lack of software packages
- Debian has many more packages than standard Red Hat (1500 vs 15000 then, 13500 (FC18) vs 40000 now for Debian testing)
- Fedora Core likes to test new linux technology, not good for our servers while Red Hat Server is much more limited in packages.
- Ubuntu is the new better Debian for cool kids, right?
- Well, kinda sorta then, much more debatable now.
- So we started with Ubuntu Dapper at the time, which we transformed into Debian testing as we upgraded our new distribution, aka ProdNG (we didn't want to migrate to upstart, nor did we like some things Canonical was force pushing into Ubuntu).

SysV vs Debian Insserv vs Upstart vs Systemd: **SysV**

- Sequential booting with /etc/rcX.d/Sxxdaemon is simple
- But it's slow
- A single hanging script can stop other daemons like ssh from starting (we manually start a rescue sshd and basic hardcoded networking before the root filesystem is even fsck'ed and remounted read-write)

SysV vs Debian Insserv vs Upstart vs Systemd: **Upstart**

- Mostly used on Ubuntu (also ChromeOS).
- Requires a totally different syntax (ideally better than shell).
- Does not guarantee any specific boot order.
- Very dependent on proper dependencies being found and specified by the maintainers. This is hard.
- It will deadlock and stop the boot if something is wrong, which can happen one boot out of three for instance.
- On occasion, upstart will enter states that require a reboot to clear.
- It can be hard to debug, especially on headless servers.

SysV vs Debian Insserv vs Upstart vs Systemd: **Systemd**

- Originally went into Fedora for testing and tuning.
- Not yet included in server distributions like RHEL.
- Very disruptive, big redesign of how Linux systems boot. It replaces many core parts of low level Linux.
- On the plus side, Lennart has done a very good job in explaining the rationale behind the required changes and the gains.
- Ideal design does not rely on dependencies being specified by the packagers, they are auto computed on demand. Real life is sometimes otherwise though, and requires manual dependencies.
- Like upstart, given boot order not specified, could trigger race conditions in our scripts or daemons, and only on 1% of our machines.
- Systemd sounded simple, but the implementation and getting everything right is much more complex than we're comfortable with.
- Maybe later as a separate effort if the price is worth the benefits.

SysV vs Debian Insserv vs Upstart vs Systemd: **Insserv**

- Debian had an upstart like dependency specified boot, before everyone else with insserv and startpar.
- Before reboot, insserv will analyze specified dependencies between scripts, and rename initscripts as S10, some as S20, and so forth.
- Everything under S10 is started at the same time, and things in S20 won't start until all of S10xx has started.
- It's easy to visualize and review dependencies before reboot.
- We can freeze them in our image, and deploy everywhere
- Simple, and everything is the same => winner for us for now.

ProdNG, tell us more...

- It's self hosting and entirely rebuilt from source.
- All packages are stripped of unnecessary dependencies and libraries (optional xml2 support, seLinux library, libacl2, etc..)
- Less is more: end distribution around 150MB/150 packages (without our custom bits). How many packages in your server image?
- Smaller is quicker to sync, re-install, and fsck.
- No complicated upstart, dbus, plymouth.
- Newer packages are not always better. Sometimes old is good (unfortunately OSS also suffers from feature creep, xml2 for rpm?)
- It's hermetic: we create a ProdNG chroot on the fly and install build tools each time you build a new package.

Ok, we have a new image, how do we push it?

- How do we convince our admins on call for google.com that we're not going to kill services by pushing the update?
- The new image was created with a distro 10 years newer, and packages that do not even contain the same binaries.
- Just upgrading coreutils from v4 to v7 is very scary since GNU willfully broke backward compatibility to make Posix happy.
- Doing a file by file compare across 20K+ files is not practical.
- Ultimately, no way to guarantee that we can just switch distributions and it'll work.
- Hard to convince our internal users to run production services on a very different distro, and even find beta testers.

Ok, we have a new image, how do we push it? (2)

- Do we end up rolling a very different distro over a month, or over a year and end up with a non uniform setup in production for that long? That's not going to make debugging easy...
- Does it mean we have to maintain 2 totally different distributions mostly in sync for that long?
- OMG, and one is made with RPMs, while the other one is dpkg, using totally different build rules and inter package dependencies.
- Argh!

Distros are hard!

Shopping didn't make the problem go away, what now?

- We need to keep all the machines in a consistent state, and stay with 2 distros: the current/old one and the new one being pushed
- If flag day there must be, as short a day as possible it will be.
- 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.
- So, we can't have a big jump.

Turning the scary big jump into many smaller safer jumps

- How about we feed those Debian packages into our Red Hat 7.1 based image a little bit at a time for each new image?
- Yes, that means building Debian packages, and converting them to rpm.
- This also means having both distributions be libc/binary compatible.
- Thankfully Ubuntu dapper was reasonably old, so we started by feeding libc 2.3.6 into our distro (replacing its libc 2.2.2)
- Then we wrote an alien-like script to unpack built debs from ProdNG, and repack them as RPMs (converting dependencies and changelogs).

Package conversions from deb to rpm

- Converting debs to rpms causes issues with pre/post install scripts.
- But thankfully we already had gotten rid of most of them.
- We had to use some sed hackery to change dependency names between Red Hat and Debian.
- In the end, we were able to build ProdNG debs that installed in our current Red Hat based image.
- So the migration process began.

Turning Red Hat 7.1 into Debian testing: cruft removal

- The first part was finding all packages that we inherited from RH 7.1 and were pure cruft we never needed (X server, fonts, font server for headless machines without X local or remote, etc...)
- Next is finding 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...)
- It's amazing how many packages come with a distro that you don't really need on servers.
- Old libraries that you find out nothing is using (libtiff, libjpeg, libpng, libdb2, libncurses4, libtermcap ...)
- You can recompile daemons without libwrap, etc...
- We actually ended up with no C++ left, so libstc++ could go.

Turning Red Hat 7.1 into Debian testing: one bit at a time

- First, upgrade libc from 2.2.2 to 2.3.6: most old binaries worked
- Warning: statically built binaries will dlopen old libc libraries directly (libnss) and fail if you upgrade libc without compat symlinks or rebuilding the static binaries.
- Start with easy packages like 'ed' and 'bc' and work your way up.
- A few binaries had warnings on stderr, we then rebuilt/upgraded them first):

```
mount --version
mount: Symbol `sys_siglist' has different size in shared object, consider re-linking
mount: mount-2.11b
```

Sometimes it's the little things that bite you

- The coreutils upgrade was scary due to the amount of breakage willfully created by upstream. Thanks to scanning our source code and fixing bad calls early, breakage was thankfully minimal for us.
- But the day I removed /etc/redhat-release, is the day I broke a bunch of java that parsed this file to do custom things with fonts depending on the presence of that file.
- Whoever touched something last, is responsible for breakage.
- So you revert the change, fix the bug, and try again later.


/etc/lsb-release you say?
Bah, /etc/redhat-release has worked for 15 years, we'll keep using that.

Going from an image snapshot to one built from scratch

- Then started the painstaking upgrade of around 150 packages, carefully stripped and built from scratch in the hermetic chroot.
- We upgraded them a few at a time, by building them in ProdNG and putting them in both ProdNG (not used yet) and the current image.
- And this was mostly a part time job done by one person.


A long time later...
PATIENT BEAR
Will be ready when you are

Getting closer to the merge: converting rpms to debs

- After a lot of effort, all standard Linux packages in the image had been upgraded to much newer ProdNG built ones.
- We were left with internally built packages, built natively as RPMs
- Then started the joy of converting RPMs to debs, especially changelogs (they are free-form in RPMs, without timezones in dates, and real syntax in changelog lines).
- But nothing that a very ugly parser script can't take care of.
- We ended up with a combination of alien(1) and our custom changelog converter (alien doesn't convert changelogs, and full changelogs are required for our package reviews)

What's left?

- Current image is now full of ProdNG debs converted to rpms.
- Custom RPMs are converted to debs and included in ProdNG.
- ProdNG now boots and seems to act like the current long ago Red Hat based image.
- Throw our regression tester at it, run all our daemons, make sure they come up (health checks), test crash reboots, test reverting to the old image, and make sure all daemon restarts work.
- Do a manual review of the differences left between 2 images (file by file diff of still 1000+ files), fix small differences left over.
- Push the new image, cross your fingers, and...

Go on vacation!

Well, things never work out like you hoped...

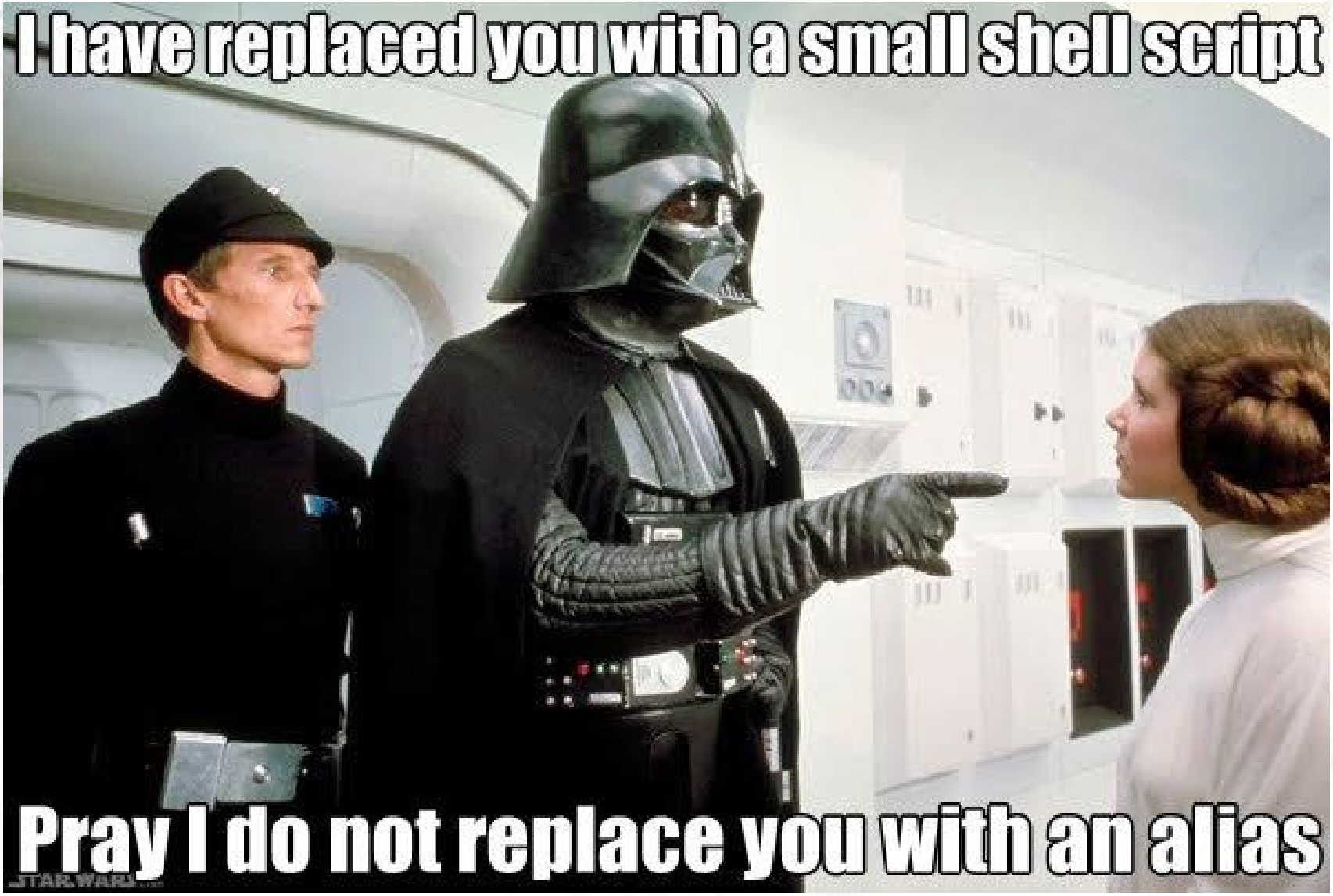
Ok, so how did the switch really go?

- I didn't go on vacation just after pushing the change of course ;)
- Testing went quite well. We only found one custom daemon that was still storing too much data in /var/run, which had become a small tmpfs, and failing (/var/run used to be part of the root filesystem). It was rebuilt to store data where it was supposed to.
- Most of the time was actually spent helping our package owners convert their RPMs to debs and switching to new upload and review mechanisms.
- We also wrote new review tools to diff and review our new image, built from scratch each time.

Misc bits: dual package system support

- We had the means to allow installing rpms unmodified to our ProdNG deb image, and even have them update the dpkg file list.
- We however opted for not adding 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.
- As a result, we were able to drop another 4MB or so of packages just for RPM support.
- rpm2cpio itself requires 3-4MB of dependencies, but it can be rewritten as a small shell script :)


Go away
or I will
replace
you with a
very small
shell script
#:
gg
shirts


I have replaced you with a small shell script
Pray I do not replace you with an alias

rpm2cpio, as small as it gets, in shell

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

```
leadsize=96
o=$(($leadsize + 8))
set -- $(od -j $o -N 8 -t u1 $pkg)
il=$((256 * ( 256 * ( 256 * $2 + $3 ) + $4 ) + $5))
dl=$((256 * ( 256 * ( 256 * $6 + $7 ) + $8 ) + $9))

sigsize=$((8 + 16 * $il + $dl))
o=$(($o + $sigsize + ( 8 - ( $sigsize % 8 ) ) % 8 + 8))
set -- $(od -j $o -N 8 -t u1 $pkg)
il=$((256 * ( 256 * ( 256 * $2 + $3 ) + $4 ) + $5))
dl=$((256 * ( 256 * ( 256 * $6 + $7 ) + $8 ) + $9))

hdrsize=$((8 + 16 * $il + $dl))
o=$(($o + $hdrsize))

dd if=$pkg ibs=$o skip=1 2>/dev/null | gunzip
```

And finally: Profit!

Lessons learned and tips #1

- If you have the expertise and many machines, maintaining your own sub Linux distribution in house gives you much more control.
- At large scales, forcing server users to use an API you provide, and not to write on the root FS definitely helps with maintenance.
- File level syncing recovers from any state and is more reliable than most other methods.
- You'll even be able to do crazy things like distribution switches.
- Don't blindingly trust and install upstream updates. They are not all good. They could conflict with your config, or even be trojanned.

Lessons learned and tips #2

- If you can, prune/remove all services/libraries you don't really need. Fewer things to update, and fewer security bugs to worry about.
- Running the latest Fedora Core or Ubuntu from 6 months ago is often much more trouble than it's worth.
- Full updates of servers every 3 years is not actually unreasonable, so Debian stable (plus testing cherrypicks) or RHEL are the way to go for servers.
- Depending on your server setup, partial upgrades may be better for you. If so, choose a distribution that allows this (like Debian).
- If it is possible for you to do so, the best way to do a huge update is incrementally a few packages at a time (this is still mostly possible with Debian where you can cherry-pick updates).

Questions?

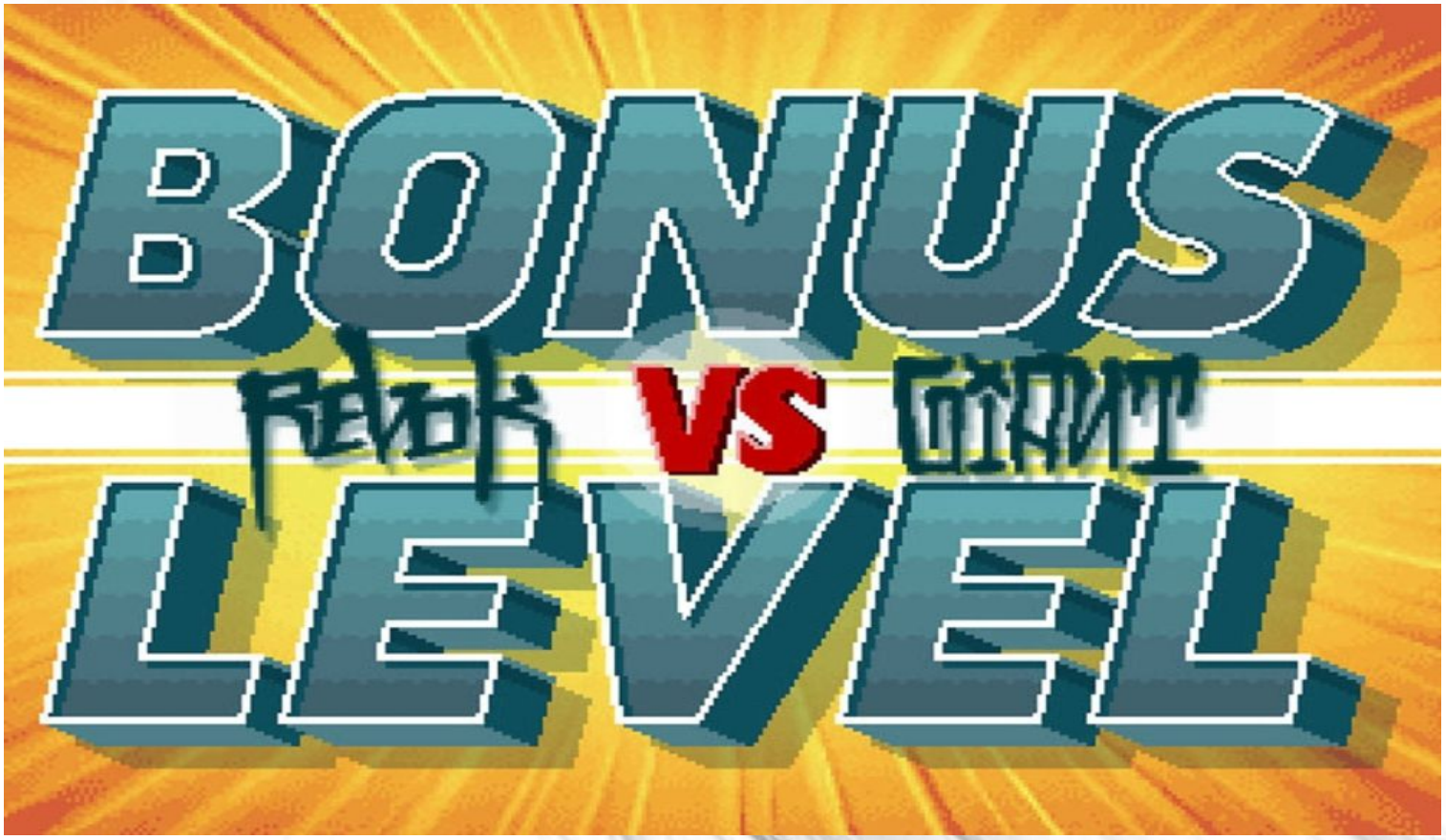
Talk slides for download:

http://marc.merlins.org/linux/talks/ProdNG-LISA/

Marc MERLIN
marc_soft@merlins.org

Bonus Round: Fight!

(aka extra slides to read later if if there is time left over)



ProdNG package generation: method

- We use the Debian source with modifications to Debian/rules to recompile with fewer options and/or exclude sub-packages we don't need.
- Package build creates a full ProdNG image on the fly using the latest packages you specified (ProdNG image + build packages), installs the source and just the dependencies you specified.
- It's fast (<1mn for build environment to be ready)
- You can for instance build without having libncurses visible to the build, even though it is otherwise in the image.
- Image build is of course fully hermetic: You can build 32bit Ubuntu dapper on 64bit lucid or precise, or even Red Hat.

ProdNG package generation: invariant builds → security

- For both security and our build system, we've modified packages to make them invariant (2 builds of the same source should give the same package bit for bit).
- This is a great way to verify quickly if your build servers are generating the same binaries than a cleanly installed workstation.
- 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...).
- We then compare the package's files against files from the previous version of the package, and revert mtime only changes.
- We have special code that doesn't encode the gzip time in compressed archives (like man pages), and reverts the mtime of the source .py file encoded in .pyc files.
- If after those cleanups the new package is identical to the old one, it's thrown out as identical.

ProdNG image generation, getting a secure image.

- 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).
- Then we have an image diff tool that shows diffs in ASCII files, list of files and permissions between 2 images.
- Package rebuilds 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). Same thing for .pyc files.
- As a result, rebuilding an image with the same input packages is reproducible and gives the same output.

And which Linux kernel did you run, surely not 2.4.2?

- We've long run custom kernels and maintained our own kernel source.
- Drivers for our custom hardware
- Oh look, we got a dump truck of really cheap RAM, but not all the bits are usable. Can we work around the bad RAM cells in software and use it anyway?
- Hardware or kernel bugs that only showed up with many machines (we do upstream such changes)
- Custom memory management and containers before they were available in the mainstream kernel.
- We do reboot machines in staggered fashion for kernel updates.

File tracking

- You don't want to create files in postinstall that won't be in the dpkg/rpm file list since it's good to be able to use rpm -qf /file or dpkg -S /file.
- Generally, you don't want to rely on pre/postinstalls to manage symlinks like /etc/rc3.d/S10startdaemon. The order of install/remove/trigger run/pre/postinstalls + inter package triggers is fiendishly complicated at times. We just made the symlinks part of the package itself, and we know for sure they get removed if we remove the package.

Testing software before doing an update image.

- We have a test-rpm-install/test-deb-install that takes a clean machine, installs the package, and gets a before/after snapshot of the entire filesystem.
- That way you ensure that your package is affecting just what you had in mind, and nothing more.
- This also lets you show 'this is what I'm changing' list along with your testing notes when you submit a package for inclusion review in the next image.
- You can also pick up side effect of restarting a daemon if it creates, modifies, or removes files.
- Then all the files are automatically reverted for the next test.

Debugging boots and reboots?

- When the rootfs is unclean, fsck often requires a reboot, so to avoid a double reboot, we pivot_root to a tmpfs, unmount the root partition, fsck it, mount it back, and pivot_root back to it.
- We have serial console on some test machines, but not worth the price on all machines.
- bootlogdwill capture boot messages without requiring the much heavier and buggy plymouth.
- Each initscript runt ime is logged, so we can find which ones can take too long.
- We also know if we booted with missing RAM, or a dead disk, and so forth.

Turning Red Hat 7.1 into Debian testing: coreutils

- A big one: coreutils-7.4-1.2 (was fileutils-4.0.36-4 + textutils-2.0.11-7 + sh-utils-2.0-13)
- tail +N/ head +N / sort +N / uniq +N are all broken
- cp/mv --reply is now invalid syntax
- seq N M with N > M breaks (use seq N -1 M)
- ls output formats (piped to awk/cut) broke
- Multiple utilities like basename, cut, nice, and sort got moved between /usr/bin and /bin, or back (and some people hardcoded the full pathnames).

Turning Red Hat 7.1 into Debian testing: other big chunks

- fileutils-4.0.36-4 + textutils-2.0.11-7 + sh-utils-2.0-13 ->

  coreutils-7.4-1.2

- util-Linux-2.12r-1.6.4 + schedutils-1.5.0-2 + mount-2.11b-3 ->

  util-Linux-2.17.2 + libblkid1-2.17.2 + libuuid1-2.17.2 + mount-2.17.2

- Other interesting chained upgrades like:

  cracklib + glib + pam + pwdb + passwd ->

  libpam-modules + libpam-runtime + libpam0g + libcap1

From an image snapshot to one built from scratch

- Our original image was a full filesystem image that got snapshotted in perforce (with a special file to store metadata).
- We ended up with files that were not runtime created, and not part of any package either.
- I ended up finding the expected leftover files (.rpmsave, old unused files), lockfiles and logfiles that shouldn't have been checked in, /etc/rcxx initscript symlinks.
- Any actual program that wasn't part of a package, was identified and moved to a package.
- This was the first step to having an image that could be rebuilt from scratch and replace the snapshotted image.

How did old image reviews work?

- The old image had all its files checked into Perforce.
- 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.
- and we used our Perforce review tools to review the file changes.

How does the new image review work?

- The new ProdNG image is not checked in perforce file by file. We get a full image in tar.gz format that is handed off to our pusher.
- 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.