USENIX Association

Proceedings of LISA ’11:
25th Large Installation System
Administration Conference

December 4–9, 2011

Boston, Massachusetts

Conference Organizers

Program Co-Chairs
Thomas A. Limoncelli, *Google, Inc.*
Doug Hughes, *D. E. Shaw Research, LLC*

Program Committee
Narayan Desai, *Argonne National Lab*
Andrew Hume, *AT&T Labs—Research*
Duncan Hutty, *ZOLL Medical Corporation*
Dinah McNutt, *Google, Inc.*
Tim Nelson, *Worcester Polytechnic Institute*
Mario Obejas, *Raytheon*
Mark Roth, *Google, Inc.*
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*
Federico D. Sacerdoti, *Aien Capital & Aien Technology*
Marc Stavely, *Consultant*
Nicole Forsgren Velasquez, *Pepperdine University*
Avleen Vig, *Etsy, Inc.*
David Williamson, *Microsoft Tellme*

Invited Talks Coordinators
Æleen Frisch, *Exponential Consulting*
Kent Skaar, *VMware, Inc.*

Workshops Coordinator
Cory Lueninghoener, *Los Alamos National Laboratory*

Guru Is In Coordinator
Chris St. Pierre, *Oak Ridge National Laboratory*

Poster Session Coordinator
Matt Disney, *Oak Ridge National Laboratory*

Work-in-Progress Reports (WiPs) Coordinator
William Bilancio, *Arora and Associates, P.C.*

Training Program
Daniel V. Klein, *USENIX Association*

USENIX Board Liaison
David N. Blank-Edelman, *Northeastern University*

Steering Committee
Paul Anderson, *University of Edinburgh*
David N. Blank-Edelman, *Northeastern University*
Mark Burgess, *CFEngine*
Alva L. Couch, *Tufts University*
Rudi van Drunen, *Competa IT*
Æleen Frisch, *Exponential Consulting*
Xev Gittler, *Morgan Stanley*
William LeFebvre, *Digital Valence, LLC*
Mario Obejas, *Raytheon*
Ellie Young, *USENIX Association*
Elizabeth Zwicky, *Consultant*

The USENIX Association Staff

External Reviewers

Paul Armstrong
Derek J. Balling
Steve Barber
Matthew Barr
Lois Bennett
Ken Breeman
Travis Campbell
Brent Chapman
Marc Chiarini
Alva L. Couch
Matt Disney
Rudi van Drunen
Bill Lefebvre
Cory Lueninghoener
Chris McEniry
Adam Moskowitz
Mario Obejas
Tobias Oetiker
Cat Okita
Eric Radman
Benoit Sigoure
Josh Simon
Kent Skaar
Ozan Yigit

LISA ’11:
25th Large Installation System Administration Conference
December 4–9, 2011
Boston, Massachusetts

Wednesday, December 7

Perspicacious Packaging

Clusters and Configuration Control

Security 1

Thursday, December 8

From Small Migration to Big Iron

Backup Bonanza

To the Cloud!

Honey and Eggs: Keeping Out the Bad Guys with Food

Seriously Snooping Packets

Friday, December 9

Network Security

Networking 1

Migrations, Mental Maps, and Make Modernization

Message from the Program Co-Chairs

Dear LISA ’11 Attendee,

There are two kinds of LISA attendees: those who read this letter at the conference and those who read it after they’ve returned home. To the first group, get ready for six days of brain-filling, technology-packed, geek-centric tutorials, speakers, papers, and more! To those that are reading this after the conference, we ask, “What’s it like living in the future? How was the conference? What cool tips and tools did you take home with you to make your job easier?”

Being a sysadmin is kind of like living in the future. You work with technology every day that would make Buck Rogers jealous. Most of our friends are jealous, too. When LISA started 25 years ago, a “large site” had 10 computers, each the size of a dishwasher, with a few gigabytes of combined storage. Today our cell phones have 32GB of “compact flash,” which is often more than the NFS quota we give our users.

Attending LISA is kind of like spending a week living in the future. We learn technologies that are cutting-edge—little known now, but next year everyone will be talking about them. When we return from LISA we sound like time travelers visiting from the future talking about new and futuristic stuff. LISA makes us look good.

LISA rarely has a cohesive conference theme, but this year we thought it was important to highlight DevOps, as it is a significant cultural change. Although DevOps is often thought of as “something big Web sites do,” the lessons learned transfer well to enterprise computing.

LISA has always been assembled using the sweat of many dedicated volunteers. It takes a lot of effort to put a conference like this together, and this year is no different. Most prominent are the Invited Talks committee (Æleen Frisch and Kent Skaar) and the Program Committee (Narayan Desai, Andrew Hume, Duncan Hutty, Dinah McNutt, Tim Nelson, Mario Obejas, Mark Roth, Carolyn Rowland, Federico D. Sacerdoti, Marc Stavely, Nicole Forsgren Velasquez, Avleen Vig, and David Williamson), but also important are the Workshops Coordinator (Cory Lueninghoener), the Guru Is In Coordinator (Chris St. Pierre), the Poster Session Coordinator (Matt Disney), and the Work-in-Progress Reports Coordinator (William Bilancio). We couldn’t have done it without every one of them. Of course, nothing would happen without the leadership of the USENIX staff. We are indebted to you all!

Of the 63 papers submitted, we accepted 28. These papers represent the best “deep thought” research, as well as Practice and Experience Reports that tell the stories from people “in the trenches.” We encourage you to read them all. However, the power of LISA is the personal interaction: introduce yourself to the attendees standing in line near you, strike up a conversation with the person sitting next to you. And remember to have fun!

Sincerely,
Thomas A. Limoncelli, *Google, Inc.*
Doug Hughes, *D. E. Shaw Research, LLC*
Program Co-Chairs

Staging Package Deployment via Repository Management

Chris St. Pierre - stpierreca@ornl.gov
Matt Hermanson - mjhermanson@ornl.gov
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, USA[*]

Abstract

This paper describes an approach for managing package versions and updates in a homogenous manner across a heterogenous environment by intensively managing a set of software repositories rather than by managing the clients. This entails maintaining multiple local mirrors, each of which is aimed at a different class of client: One is directly synchronized from the upstream repositories, while others are maintained from that repository according to various policies that specify which packages are to be automatically pulled from upstream (and therefore automatically installed without any local vetting) and which are to be considered more carefully – likely installed in a testing environment, for instance – before they are deployed widely.

Background

It is important to understand some points about our environment, as they provide important constraints to our solution.

We are lucky enough to run a fairly homogenous set of operating systems consisting primarily of Red Hat Enterprise Linux and CentOS servers, with fair numbers of Fedora and SuSE outliers. In short, we are dealing entirely with RPM-based packaging, and with operating systems that are capable of using yum [12]. As yum is the default package management utility for the majority of our servers, we opted to use yum rather than try to switch to another package management utility.

For configuration management, we chose to use Bcfg2 [3] for reasons wholly unrelated to package and software management. Bcfg2 is a Python and XML-based configuration management engine that "helps system administrators produce a consistent, reproducible, and verifiable description of their environment" [3]. It is in particular the focus on reproducibility and verification that forced us to consider updating and patching anew.

In order to guarantee that a given configuration – where a "configuration" is defined as the set of paths, files, packages, and so forth, that describes a single system – is fully replicable, Bcfg2 ensures that every package specified for a system is the latest available from that system's software repositories [8]. (As will be noted, this can be overridden by specifying an explicit package version.) This grants the system administrator two important abilities: to provision identical machines that will remain identical; and to reprovision machines to the exact same state they were previously in. But it also makes it unreasonable to simply use the vendor's software repositories (or other upstream repositories), since all updates will be installed immediately without any vetting. The same problem presents itself even with a local mirror.

Bcfg2 can also use "the client's response to the specification ... to assess the completeness of the specification" [3]. For this to happen, the Bcfg2 server must be able to understand what a "complete" specification entails, and so the server does not entirely delegate package installation to the Bcfg2 client. Instead, it performs package dependency resolution on the server rather than allowing the client to set its own configuration. This necessitates ensuring that the Bcfg2 Packages plugin uses the same

[*] This paper has been authored by contractors of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

yum configuration as the clients; Bcfg2 has support for making this rather simple [8], but the Packages plugin does not support the full range of yum functionality, so certain functions like the “versionlock” plugin and even package excludes, are not available. Due to the architecture of Bcfg2 – architecture designed to guarantee replicability and verification of server configurations – it is not feasible or, in most cases, possible to do client-based package and repository management. This became critically important in selecting a solution.

Other Solutions

There are a vast number of potential solutions to this problem that would seem to be low-hanging fruit – far simpler to implement, at least initially, than our ultimate solution – but that would not work, for various reasons.

Yum Excludes

A core yum feature is the ability to exclude certain packages from updates or installation [13]. At first, this would seem to be a solution to the problem of package versioning: simply install the package version you want, and then exclude it from further updates. But this has several issues that made it unsuitable for our use (or, we believe, this use case in general):

- It does not (and cannot) guarantee a specific version. Using excludes to set a version depends on that version being installed (manually) prior to adding the package to the exclude list.
- There is no guarantee that the package is still in the repository. Many mainstream repositories[1] do not retain older versions in the same repository as current packages. Consequently, when reinstalling a machine where yum excludes have been used to set package versions (or when attempting to duplicate such a machine), there is no guarantee that the package version expected will even be available.
- In order to use yum excludes to control package versions, a very specific order of events must occur: first, the machine must be installed without the target package included (as Kickstart, the RHEL installation tool, does not support installing a specific version of a package [1]); next, the correct package version must be installed; and finally, the package must be added to the exclude list. If this happens out of order, then the wrong version of the package might be installed, or the package might not be installed at all.
- Supplying a permitted update to a package is even more difficult, as it involves removing the package exclusion, updating to the correct version, and then restoring the exclusion. A configuration management system would have to have tremendously granular control over the order in which actions are performed to accomplish this delicate goal.
- As discussed earlier, Bcfg2 performs dependency resolution on the server side in order to provide a guarantee that a client’s configuration is fully specified. By using yum excludes – which cannot be configured in Bcfg2’s internal dependency resolver – the relationship between the client and the server is broken, and Bcfg2 will in perpetuity claim that the client is out of sync with the server, thus reducing the usefulness of the Bcfg2 reporting tools.

While yum excludes appear at first to be a viable option, their use to set package versions is not replicable, consistent, and cannot be trivially automated.

Specifying Versions in Bcfg2

Bcfg2 is capable of specifying specific versions of packages in the specification, e.g.:

```
<BoundPackage name="glibc" type="yum">
  <Instance version="2.13" release="1"
            arch="i686"/>
  <Instance version="2.13" release="1"
            arch="x86_64"/>
</BoundPackage>
```

This is obviously quite verbose (more so because the example uses a multi-arch package), and as a result of its verbosity it is also error-prone. Having to recopy the version, release, and architecture of a package – separately – is not always a trivial process, and the relatively few constraints of version and release strings makes it less so. For instance, given the package:

```
iomemory-vsl-2.6.35.12-88.fc14.x86_64-
    2.3.0.281-1.0.fc14.x86_64.rpm
```

The package name is "iomemory-vsl-2.6.35.12-88.fc14.x86_64" (which refers to the specific kernel for which it was built), the version is "2.3.0.281" and the release is "1.0.fc14".[2] This can be clarified through use of the `--queryformat` option to `rpm`, but the fact that more advanced RPM commands are necessary makes it clear that this approach is untenable in general. Even more worrisome is the package epoch, a sort of "super-version," which RPM cleverly hides by default, but could cause a newer package to be installed if it was not specified properly.

Maintenance is also tedious, as it involves endlessly updating verbose version strings; recall that a given version is just shorthand for what we actually care about – that a package *works*.

This approach also does not abrogate the use of yum on a system to update it beyond the appropriate point. The only thing keeping a package at the chosen version is Bcfg2's own self-restraint; if an admin on a machine lacks that same self-restraint, then he or she could easily update a package that was not to be updated, whereupon Bcfg2 would try to downgrade it.

Finally, this approach presents specific difficulties for us, as our adoption of Bcfg2 is far from complete; large swaths of the center still use Cfengine 2, and some machines – particularly compute and storage platforms – operate in a diskless manner and do not use configuration management tools in a traditional manner. They depend entirely on their images for package versions, so specifying versions in Bcfg2 would not help.

To clarify, using Bcfg2 forced us to reconsider this problem, and any solution must be capable of working with Bcfg2, but it cannot be assumed that the solution may leverage Bcfg2.

Yum versionlock

Using yum's own version locking system would appear to improve upon pegging versions in Bcfg2: it works on all systems, regardless of whether or not they use Bcfg2; and a shortcut command, `yum versionlock <package-name>`, is provided to make the process of maintaining versions less error-prone.[3]

It also solves many of the problems of yum excludes, but suffers from a critical flaw in that approach: by setting package versions on the client, the relationship between the Bcfg2 client and server would be broken.

Combinations of these three approaches merely exhibit combinations of their flaws. For instance, the promising combination of yum's versionlock plugin and specifying the version in Bcfg2 would ensure that the Bcfg2 client and server were of a mind about package versions, and would work on non-Bcfg2 machines; however, it would forfeit versionlock's ease of use and require the administrator to once again manually copy package versions.

Spacewalk

Spacewalk was the first full-featured solution we looked at that aims to replace the mirroring portion of this relationship; all of the other potential solutions listed thus far have attempted to work with a "dumb" mirror and use yum features to work around the problem we have described. Spacewalk is a local mirror system that "manages software content updates for Red Hat derived [*sic*] distributions" [10]; it is a tremendously full-featured system, with support for custom "channels," collections of packages assembled in an ad-hoc basis.

Unfortunately, Spacewalk was a non-starter for us for the same reason that it has failed to gain much traction in the community at large: of the two versions of Spacewalk, only the Oracle version actually implements all of the features; the PostgreSQL version is deeply underfeatured, even after several years of work by the Spacewalk team to port all of the Oracle stored procedures.

As it turns out, Red Hat has a successor in mind for Spacewalk and Satellite: CloudForms [14]. The content management portion of CloudForms – roughly corresponding to the mirror and repository management functionality of Spacewalk – is Pulp.

A solution: Pulp

Pulp is a tool "for managing software repositories and their associated content, such as packages, errata, and distributions" [7]. It is, as noted, the spiritual successor to Spacewalk, and so implements the vast majority of Spacewalk's repository management features without the dependency on Oracle.

Pulp's usage model involves syncing multiple upstream repositories locally; these repositories can then be *cloned*, which uses hard links to sync them locally with almost no disk space used. This allows us to sync a repository once, then duplicate it as many times as necessary to support multiple teams and multiple stability levels. The sync process supports *filters*, which allow us to blacklist or whitelist packages and thus exclude "impactful" packages from automatic updates.

Pulp also supports manually adding packages to and removing packages from repositories, so we can later update a given package across all machines that use a repository with a single command. Adding and removing also tracks dependencies, so it's not possible to add a package to a repository without adding the dependencies necessary to install it.[4]

Workflow

Pulp provides us with the framework to implement a solution to the problem outlined earlier, but even as featureful as it is it remains a fairly basic tool. Our workflow – enforced by the features Pulp provides, by segregating repositories, by policy, and by a nascent in-house web interface – provides the bulk of the solution. Briefly, we segregate repositories by tier to test packages before site-wide roll-outs, and by team to ensure operational separation. Packages are automatically synced between tiers based on package filters, which blacklist certain packages that must be promoted manually. This ensures that most packages benefit from up to two weeks of community testing before being deployed site-wide, and packages that we have judged to be more potentially "impactful" from more focused local testing as well.

Tiered Repositories

We maintain different repository sets for different "levels" of stability. We chose to maintain three tiers:

`live` Synced daily from upstream repositories; not used on any machines, but maintained due to operational requirements within Pulp[5] and for reference.

`unstable` Synced daily from `live`, with the exception of selected "impactful" packages (more about which shortly), which can be manually promoted from `live`.

`stable` Synced daily from `unstable`, with the exception of the same "impactful" packages, which can be manually promoted from `unstable`.

This three-tiered approach guarantees that packages in `stable` are at least two days old, and "impactful" packages have been in testing by machines using the `unstable` branch. When a package is released from upstream and sync to public mirrors, those packages are pulled down into local repositories. From then on the package in under the control of Pulp. Initially, a package is considered unstable and is only deployed to those systems that look at the repositories in the `unstable` tier. After a period of time, the package is then promoted into the `stable` repositories, and thus to production machines.

In order to ensure that packages in `unstable` receive ample testing before being promoted to `stable`, we divide machines amongst those two tiers thusly:

- All internal test machines – that is, all machines whose sole purpose is to provide test and development platforms to customers within the group – use the `unstable` branch. Many of these machines are similar, if not identical, to production or external test machines.
- Where multiple identical machines exist for a single purpose, whether in an active-active or active-passive configuration, exactly one machine will use the `unstable` branch and the rest will use the `stable` branch.

Additionally, we maintain separate sets of repositories, branched from `live`, for different teams or projects that require different patching policies appropriate to the needs of those teams or projects. Pulp has strong built-in ACLs that support these divisions.

In order to organize multiple tiers across multiple groups, we use a strict convention to specify the repository ID, which acts as the primary key across all repositories[6], namely:

```
<team name>-<tier>-<os name>-<os version>-
    <arch>-<repo name>
```

For example, `infra-unstable-centos-6-x86_64-updates` would denote the Infrastructure team's `unstable` tier of the 64-bit CentOS 6 "updates" repository. This allows us to tell at a glance the parent-child relationships between repositories.

Sync Filters

The syncs between the `live` and `unstable` and between `unstable` and `stable` tiers are mediated by filters[7]. Filters are regular expression lists of packages to either blacklist from the sync, or whitelist in the sync; in our workflow, only blacklists are used. A package filtered from the sync may still remain in the repository; that is, if we specify `^kernel(-.*)?` as a blacklist filter, that does not remove `kernel` packages from the repository, but rather refuses to sync new `kernel` packages from the repository's parent. This is critical to our version-pegging system.

Given our needs, whitelist filters are unnecessary; our systems tend to fall into one of two types:

- Systems where we generally want updates to be installed insofar as is reasonable, with some prudence about installing updates to "impactful" packages.
- Systems where, due to vendor requirements, we must set all packages to a specific version. Most often this is in the form of a requirement for a minor release of RHEL[8], in which case there are no updates we wish to install on an automatic basis. (We may wish to update specific packages to respond to security threats, but that happens with manual package promotion, not with a sync; this workflow gives us the flexibility necessary to do so.)

A package that may potentially cause issues when updated can be blacklisted on a per-team basis[9]. Since the repositories are hierarchically tiered, a package that is blacklisted from the `unstable` tier will never make it to the `stable` tier.

Manual Package Promotion and Removal

The lynchpin of this process is manually reviewing packages that have been blacklisted from the syncs and *promoting* them manually as necessary. For instance, if a filter for a set of repositories blacklisted `^kernel(-.*)?` from the sync, without manually promoting new kernel packages no new kernel would ever be installed.

To accomplish this, we use Pulp's *add package* functionality, exposed via the REST API as a POST to `/repositories/<id>/add_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.add_package()`, and via the CLI as `pulp-admin repo add_package`. In the CLI implementation, `add_package` follows dependencies, so promoting a package will promote everything that package requires that is not already in the target repository. This helps ensure that each repository stays consistent even as we manipulate it to contain only a subset of upstream packages[10].

Conversely, if a package is deployed and is later found to cause problems it can be removed from the tier and the previous version, if such is available in the repository, will be (re)installed. Bcfg2 will helpfully flag machines where a newer package is installed than is available in that machine's repositories, and will try to downgrade packages appropriately. Pulp can be configured to retain old packages when it performs a sync; this is helpful for repositories like EPEL that remove old packages themselves, and guarantees that a configurable number of older package versions are available to fall back on.

The *remove package* functionality is exposed via Pulp's REST API as a POST to `/repositories/<id>/delete_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.remove_package()`, and via the CLI as `pulp-admin repo remove_package`. As with `add_package`, the CLI implementation follows dependencies and will try to remove packages that require the package being removed; this also helps ensure repository consistency.

Optimally, security patches are applied 10 or 30 days after the initial patch release [2]; this workflow allows us to follow these recommendations to some degree, promoting new packages to the `unstable` tier on an approximately weekly basis. Packages that have been in the `unstable` tier for at least a week are also promoted to the `stable` tier every week; in this we deviate from Beattie et al.'s recommendations somewhat, but we do so because the updates being promoted to `stable` have been vetted and tested by the machines using the `unstable` tier.

This workflow also gives us something very important: the ability to install updates across all machines much sooner than the optimal 10- or 30-day period. High profile vulnerabilities require immediate action – even to the point of imperiling uptime – and by promoting a new package immediately to both `stable` and `unstable` tiers we can ensure that it is installed across all machines in our environment in a timely fashion.

Selecting "impactful" packages

Throughout this paper, we have referred to "impactful" packages – those to which automatic updates we determined to be particularly dangerous – as a driving factor. Were it not for our reticence to automatically update all packages, we could have simply used an automatic update facility – `yum-cron` or `yum-updatesd` are both popular – and been done with it.

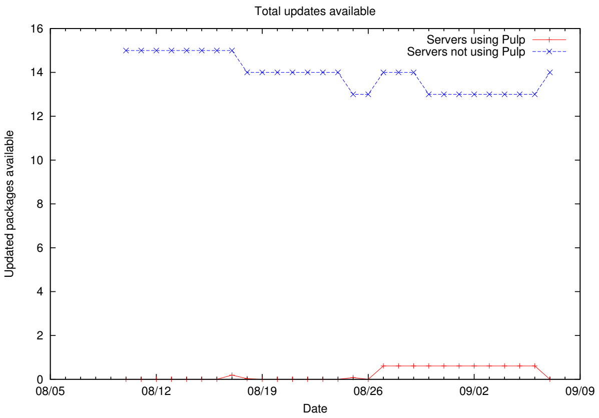
We didn't feel that was appropriate, though. For instance, installing a new kernel can be problematic – particularly in an environment with a wide variety of third-party kernel modules and other kernel-space modifications – and we wanted much closer control over that process. We flagged packages as "impactful" according to a simple set of criteria:

- The kernel, and packages otherwise directly tied to kernel space (e.g., kernel modules and Dynamic Kernel Module Support (DKMS) packages);
- Packages that provide significant, customer-facing services. On the Infrastructure team, this included packages like `bind`, `httpd` (and related modules), `mysql`, and so on.
- Packages related to InfiniBand and Lustre [9]; as one of the world's largest unclassified Lustre installations, it's very important that the Lustre versions on our systems stay in lockstep with all other systems in the center. Parts of Lustre reside directly in kernel space, an additional consideration.

The first two criteria provided around 20 packages to be excluded – a tiny fraction of the total packages installed across all of our machines. The vast majority of supporting packages continue to be automatically updated, albeit with a slight time delay for the multiple syncs that must occur.

Results

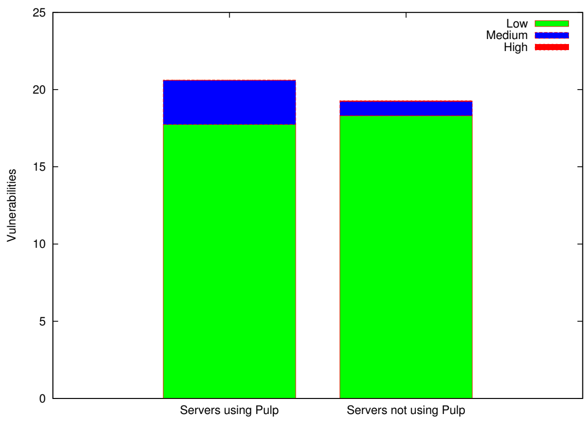
Our approach produces results in a number of areas that are difficult to quantify: improved automation reduces the amount of time we spend installing patches; not installing patches immediately improves patch quality and reduces the likelihood of flawed patches [2]; and increased compartmentalization makes it easier for our diverse teams to work to different purposes without stepping on toes. But it also provides testable, quantifiable improvements: since replacing a manual update process with Pulp and Bcfg2's automated update process, we can see that the number of available updates has decreased and remained low on the machines using Pulp.



The practice of staging package deployment makes is difficult to quantify just how out of date a client is, as yum on the client will only report the number of updates available from the repositories in `yum.conf`. To find the number of updates available from upstream, we collect an aggregate of all the package differences starting at the client and going up the heirarchy to the upstream repository. E.g., for a machine using the `unstable` tier, we calculate the number of updates available on the machine itself, and then the number of updates available to the `unstable` tier from the `live` tier.

The caveat to this approach is when, for instance, a package splits into two new packages. This results in two new packages, and one missing package, totaling three "updates" according to `yum check-update`, or zero "updates" when comparing repositories themselves, when in reality it is a single package update. For example, if package `foo` recieves an update that results in packages `foo-client` and `foo-server`, this could result in a margin of error of -1 or +2. This gives a slight potential benefit to machines using Pulp in our metrics, as updates of this sort are underestimated when calculating the difference between repositories, but overestimated when using `yum` to report on updates available to a machine. In practice, this is extremely rare, though, and should not significantly affect the results.

Ensuring, with a high degree of confidence, that updates are installed is wonderful, but even more important is ensuring that vulnerabilities are being mitigated. Using the data from monthly Nessus [11] vulnerability scans, we can see that machines using Pulp do indeed reap the benefits of being patched with more frequency:[11]



This graph is artificially skewed against Pulp due to the sorts of things Nessus scans for; for instance, web servers are more likely to be using Pulp at this time simply due to our implementation plan, and they also have disproportionately more vulnerabilities in Nessus because they have more services exposed.

Future Development

Sponge

At this time, Pulp is very early code; it has been in use in another Red Hat product for a while, so certain paths are well-tested, but other paths are pre-alpha. Consequently, its command line interface lacks polish, and many tasks within Pulp require extraordinary verbosity to accomplish. It is also not clear if Pulp is intended for standalone use, although such is possible.

To ease management of Pulp, we have written a web frontend for management of Pulp and its objects, called “Sponge.” Sponge, powered by the Django [4] web framework, provides views into the state of Pulp repositories along with the ablity to manage its contents. Sponge leverages Pulp’s Python client API to provide convience functions that ease our workflow.

By presenting the information visually, Sponge makes repository management much more intuitive. Sponge extends the functionality of Pulp by displaying the differences between a repository and its parent in the form of a diff. These diffs give greater insight into exactly how `stable`, `unstable`, and `live` tiers differ. They also provide insight into the implications of a package promotion or removal.

This is particularly important with package removal, since, as noted, removing a package will also remove anything that requires that specific package. Without Sponge’s diff feature and a confirmation step, that is potentially very dangerous; Pulp itself only gives you confirmation of the packages removed without an opportunity to confirm or reject a removal. The contrapositive situation – promoting a package pulling in unintended dependencies – is also potentially dangerous, albeit less so. Sponge helps avert both dangers.

Guaranteeing a minimum package age

As Beattie at al. observe [2], the optimal time to apply security patches is either 10 or 30 days after the patches have been released. Our workflow currently doesn’t provide any way to guarantee this; our weekly manual promotion of new packages merely suggests that a patch be somewhere between 0 and 6 days old before it is promoted to `unstable`, and 7 and 13 days old before being promoted to `stable`. We plan to add a feature – either to Sponge or to Pulp – to promote packages only once they have aged properly.

Other packaging formats

In this paper we have dealt with systems using yum and RPM, but the approach can, at least in theory, be expanded to other packaging systems. Pulp intends eventually to support not only Debian packages, but actually any sort of generic content at all [6], making it useful for any packaging system. Bcfg2, for its part, already has package drivers for a wide array of packaging systems, including APT, Solaris packages (Blastwave- or SystemV-style), Encap, FreeBSD packages, IPS, Mac Ports, Pacman, and Portage. This gives a hint of the future potential for this approach.

Availability

Most of the software involved in the approach discussed in this paper is free and open source. The various elements of our solution can be found at:

Pulp `http://pulpproject.org`

Bcfg2 `http://trac.mcs.anl.gov/projects/bcfg2`

Yum `http://yum.baseurl.org/`

Sponge, the web UI to Pulp listed in the Future Development section, is currently incomplete and unreleased. We have already worked closely with the Pulp developers to incorporate features into the Pulp core itself, and we will continue to do so. We hope that Sponge will become unnecessary as Pulp matures.

Author Information

Chris St. Pierre leads the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He is deeply involved with the development of Bcfg2, contributing in particular to the specification validation tool and Packages plugin for the upcoming 1.2.0 release. He has taught widely on internal documentation, LDAP, and spam. Chris serves on the LOPSA Board of Directors.

Matt Hermanson is a member of the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He holds a B.A. in Computer Science from Tennessee Technological University.

References

[1] Anaconda/Kickstart. http://fedoraproject.org/wiki/Anaconda/Kickstart#Chapter_3._Package_Selection.

[2] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the Application of Security Patches for Optimal Uptime. Proceedings of LISA '02: Sixteenth Systems Administration Conference, USENIX, pp. 233–42.

[3] Desai, N. Bcfg2. http://trac.mcs.anl.gov/projects/bcfg2.

[4] Django Software Foundation. Django — The Web framework for perfectionists with deadlines. https://www.djangoproject.com/.

[5] Dobies, J. GCRepoApis. https://fedorahosted.org/pulp/wiki/GCRepoApis.

[6] Dobies, J. Generic Content Support. http://blog.pulpproject.org/2011/08/08/generic-content-support/.

[7] Dobies, J. Pulp - Juicy software repository management. http://pulpproject.org.

[8] Jerome, S., Laszlo, T., and St. Pierre, C. Packages. http://docs.bcfg2.org/server/plugins/generators/packages.html.

[9] Oracle Corporation. Lustre. http://wiki.lustre.org/index.php/Main_Page.

[10] Red Hat, Inc. Spacewalk: Free & Open Source Linux Systems Management. http://spacewalk.redhat.com/.

[11] Tenable Network Security. Tenable Nessus. http://www.tenable.com/products/nessus.

[12] Vidal, S. yum. http://yum.baseurl.org/.

[13] Vidal, S. yum.conf - configuration file for yum(8). man 5 yum.conf.

[14] Warner, T., and Sanders, T. The Future of RHN Satellite: A New Architecture Enabling the Traditional Data Center and the Cloud. Red Hat Summit, Red Hat, Inc.

Notes

[1] For instance, Extra Packages for Enterprise Linux (EPEL) and the CentOS repositories themselves.

[2] Admittedly, this is a non-standard naming scheme, but no solution can be predicated on the idea that all RPMs are well-built.

[3] The command in question merely maintains a local file on a machine, so that file would still have to be copied into the Bcfg2 specification, but we believe this would be less error-prone than copying package version details.

[4] This is actually only true if the package is being added from another repository; it is possible to add a package directly from the filesystem, in which case dependency checking is not performed. This is not a use case for us, though.

[5] In Pulp, filters can only be applied to repositories with local feeds.

[6] This may change in future versions of Pulp, as multiple users, ourselves included, have asked for stronger grouping functionality [5].

[7] As noted earlier, in Pulp, filters can only be applied to repositories with local feeds, so no filter mediates the sync between upstream and `live`.

[8] It is lost on many vendors that it is unreasonable and foolish to require a specific RHEL minor release. As much work as has gone into this solution, it is still less than would be required to convince most vendors of this fact, though.

[9] Technically, filters can be applied on a per-repository basis, so black- and whitelists can be applied to individual repositories. This is very rare in our workflow, though.

[10] It is true that our approach does not *guarantee* consistency. A repository sync might result in an inconsistency if a package that was not listed on that sync's blacklist required a package that was listed on the blacklist. In practice this can be limited by using regular expressions to filter families of packages (e.g., `^mysql.*` or `^(.*-)?mysql.*` to blacklist all MySQL-related packages rather than just blacklisting the `mysql-server` package itself

[11] Unfortunately long-term data was not available for vulnerabilities for a number of reasons: CentOS 5 stopped shipping updates in their mainline repositories between July 21st and September 14th; the August security scan was partially skipped; and Pulp hasn't been in production long enough to get meaningful numbers prior to that. Still, the snapshot of data is compelling.

CDE: Run Any Linux Application On-Demand Without Installation

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Abstract

There is a huge ecosystem of free software for Linux, but since each Linux distribution (distro) contains a different set of pre-installed shared libraries, filesystem layout conventions, and other environmental state, it is difficult to create and distribute software that works without hassle across all distros. Online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring Linux software and their myriad of dependencies. To address this ubiquitous problem, we have created an open-source tool called CDE that automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines. Creating a CDE package is as simple as running the target application under CDE's monitoring, and executing a CDE package requires no installation, configuration, or root permissions. CDE enables Linux users to instantly run any application on-demand without encountering "dependency hell".

1 Introduction

The simple-sounding task of taking software that runs on one person's machine and getting it to run on another machine can be painfully difficult in practice. Since no two machines are identically configured, it is hard for developers to predict the exact versions of software and libraries already installed on potential users' machines and whether those conflict with the requirements of their own software. Thus, software companies devote considerable resources to creating and testing one-click installers for products like Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers must carefully specify the proper dependencies in order to integrate their software into package management systems [4] (e.g., RPM on Linux, MacPorts on Mac OS X). Despite these efforts, online forums and mailing lists are still filled with discussions of users' troubles with compiling, installing, and configuring software and their myriad of dependencies. For example, the official Google Chrome help forum for "install/uninstall issues" has over 5800 threads.

In addition, a study of US labor statistics predicts that by 2012, 13 million American workers will do programming in their jobs, but amongst those, only 3 million will be professional software developers [24]. Thus, there are potentially millions of people who still need to get their software to run on other machines but who are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their primary job is not to release production-quality software. For example:

- **System administrators** often hack together ad-hoc utilities comprised of shell scripts and custom-compiled versions of open-source software, in order to perform system monitoring and maintenance tasks. Sysadmins want to share their custom-built tools with colleagues, quickly deploy them to other machines within their organization, and "future-proof" their scripts so that they can continue functioning even as the OS inevitably gets upgraded.
- **Research scientists** often want to deploy their computational experiments to a cluster for greater performance and parallelism, but they might not have permission from the sysadmin to install the required libraries on the cluster machines. They also want to allow colleagues to run their research code in order to reproduce and extend their experiments.
- **Software prototype designers** often want clients to be able to execute their prototypes without the hassle of installing dependencies, in order to receive continual feedback throughout the design process.

In this paper, we present an open-source tool called CDE [1] that makes it easy for people of all levels of IT expertise to get their software running on other machines without the hassle of manually creating a robust

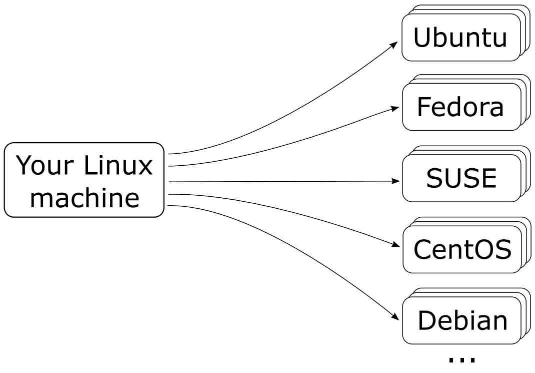


Figure 1: CDE enables users to package up any Linux application and deploy it to all modern Linux distros.

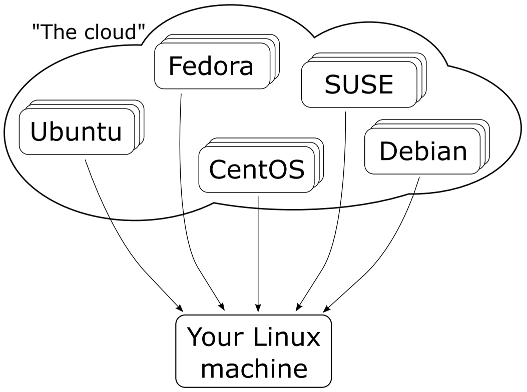


Figure 2: CDE's streaming mode enables users to run any Linux application on-demand by fetching the required files from a farm of pre-installed distros in the cloud.

installer or dealing with user complaints about dependencies. CDE automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines without any installation (see Figure 1). To use CDE, the user simply:

1. Prepends any set of Linux commands with the `cde` executable. `cde` executes the commands and uses `ptrace` system call interposition to collect all the code, data files, and environment variables used during execution into a self-contained package.
2. Copies the resulting CDE package to an x86-Linux machine running any distro from the past 5 years.
3. Prepends the original packaged commands with the `cde-exec` executable to run them on the target machine. `cde-exec` uses `ptrace` to redirect file-related system calls so that executables can load the required dependencies from within the package. Execution can range from 0% to 30% slower.

The main benefits of CDE are that creating a package is as easy as executing the target program under its supervision, and that running a program within a package requires no installation, configuration, or root permissions.

The design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to other Linux machines with as little effort as possible. However, CDE is not meant to replace traditional installers or package managers; its intended role is to serve as a convenient *ad-hoc* solution for people like sysadmins, research scientists, and prototype makers.

Since its release in Nov. 2010, CDE has been downloaded over 3,000 times [1]. We have exchanged hundreds of emails with users throughout both academia and industry. In the past year, we have made several significant enhancements to the base CDE system in response to user feedback. Although we introduced an early version of CDE in a short paper [20], this paper presents a more complete CDE system with three new features:

- To overcome CDE's primary limitation of only being able to package dependencies collected on executed paths, we introduce new tools and heuristics for making CDE packages complete (Section 3).
- To make CDE-packaged programs behave just like native applications on the target machine rather than executing in an isolated sandbox, we introduce a new *seamless execution mode* (Section 4).
- Finally, to enable users to run any Linux application on-demand, we introduce a new *application streaming mode* (Section 5). Figure 2 shows its high-level architecture: The system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). The user connects to that distro farm via an ssh-based protocol from any x86-Linux machine. The user can now run *any* application available within the package managers of any of the distros in the farm. CDE's streaming mode fetches the required files on-demand, caches them locally on the user's machine, and creates a portable distro-independent execution environment. Thus, Linux users can instantly run the hundreds of thousands of applications already available in the package managers of all distros without being forced to use one specific release of one specific distro[1].

This paper continues with descriptions of real-world use cases (Section 6), evaluations of portability and performance (Section 7), comparisons to related work (Section 8), and concludes with discussions of design philosophy, limitations, and lessons learned (Section 9).

[1] The package managers included in different releases of the same Linux distro often contain incompatible versions of many applications!

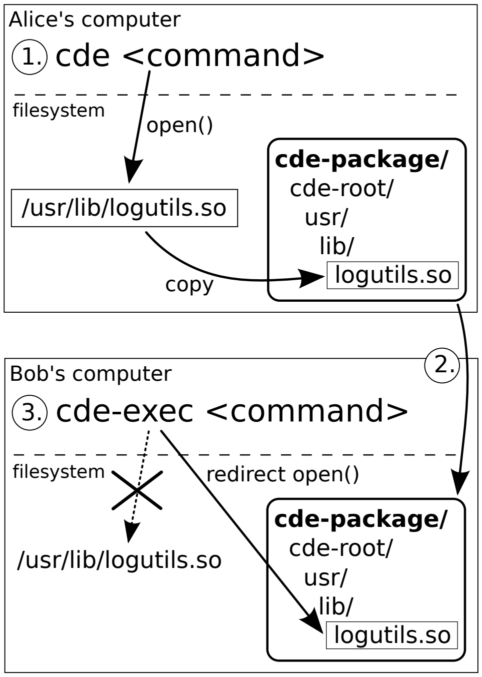


Figure 3: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends her package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

2 CDE system overview

We described the details of CDE's design and implementation in a prior paper and its accompanying technical report [20]. We will now summarize the core features of CDE using an example.

Suppose that Alice is a system administrator who is developing a Python script to detect anomalies in network log files. She normally runs her script using this Linux command:

```
python detect_anomalies.py net.log
```

Suppose that Alice's script (`detect_anomalies.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ log parsing code compiled into shared libraries. If Alice wants her colleague Bob to be able to run her analysis, then it is not sufficient to just send her script and `net.log` data file to him.

Even if Bob has a compatible version of Python on his Linux machine, he will not be able to run her script until he compiles, installs, and configures the exact extension modules that her script used (and all of their transitive dependencies). Since Bob is probably using a different Linux distribution (distro) than Alice, even if Alice precisely recalled all of the steps involved in installing all of the original dependencies on her machine, those instructions probably will not work on Bob's machine.

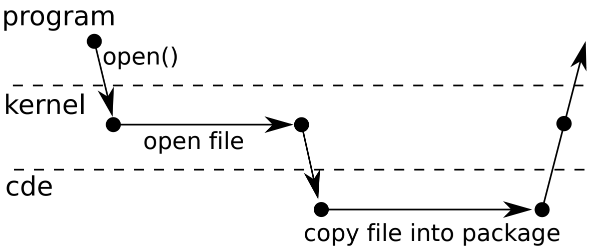


Figure 4: Timeline of control flow between target program, kernel, and `cde` process during an `open` syscall.

2.1 Creating a new CDE package

To create a self-contained package with all of the dependencies required to run her anomaly detection script on another Linux machine, Alice simply prepends her command with the `cde` executable:

```
cde python detect_anomalies.py net.log
```

`cde` runs her command normally and uses the Linux `ptrace` system call to monitor all of the files it accesses throughout execution. `cde` creates a new subdirectory called `cde-package/cde-root/` and copies all of those accessed files into there, mirroring the original directory structure. Figure 4 shows an overview of the control flow between the target program, Linux kernel, and `cde` during a file-related system call.

For example, if Alice's script dynamically loads an extension module as a shared library named `/usr/lib/logutils.so` (i.e., log parsing utility code), then `cde` will copy it to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). `cde` also saves the values of environment variables in a text file within `cde-package/`.

When execution terminates, the `cde-package/` subdirectory (which we call a "CDE package") contains all of the files required to run Alice's original command.

2.2 Executing a CDE package

Alice zips up the `cde-package/` directory and transfers it to Bob's Linux machine. Now Bob can run Alice's anomaly detection script without first installing anything on his machine. To do so, he unzips the package, changes into the sub-directory containing the script, and prepends her original command with the `cde-exec` executable (also included in the package):

```
cde-exec python detect_anomalies.py net.log
```

`cde-exec` sets up the environment variables saved from Alice's machine and executes the versions of `python` and its extension modules that are *located within the package*. `cde-exec` uses `ptrace` to monitor all

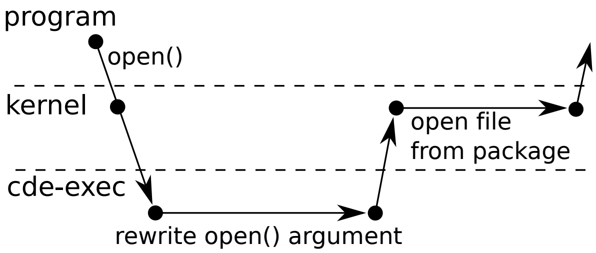


Figure 5: Timeline of control flow between target program, kernel, and `cde-exec` during an `open` syscall.

system calls that access files and dynamically rewrites their path arguments to the corresponding paths within the `cde-package/cde-root/` sub-directory. Figure 5 shows the control flow between the target program, kernel, and `cde-exec` during a file-related system call.

For example, when her script requests to load the `/usr/lib/logutils.so` library using an `open` system call, `cde-exec` rewrites the path argument of the `open` call to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). This run-time path redirection is essential, because `/usr/lib/logutils.so` probably does not exist on Bob's machine.

2.3 CDE package portability

Alice's CDE package can execute on any Linux machine with an architecture and kernel version that are compatible with its constituent binaries. CDE currently works on 32-bit and 64-bit variants of the x86 architecture (i386 and x86-64, respectively). In general, a 32-bit `cde-exec` can execute 32-bit packaged applications on 32- and 64-bit machines. A 64-bit `cde-exec` can execute both 32-bit and 64-bit packaged applications on a 64-bit machine. Extending CDE to other architectures (e.g., ARM) is straightforward because the `strace` tool that CDE is built upon already works on many architectures. However, CDE packages cannot be transported *across* architectures without using a CPU emulator.

Our portability experiments (§7.1) show that packages are portable across Linux distros released within 5 years of the distro where the package originated. Besides sharing with colleagues like Bob, Alice can also deploy her package to run on a cluster for more computational power or to a public-facing server machine for real-time online monitoring. Since she does not need to install anything as root, she does not risk perturbing existing software on those machines. Also, having her script and all of its dependencies (including the Python interpreter and extension modules) encapsulated within a CDE package makes it somewhat "future-proof" and likely to continue working on her machine even when its version of Python and associated extensions are upgraded in the future.

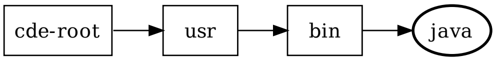


Figure 6: The result of copying a file named `/usr/bin/java` into the `cde-root/` directory.

3 Semi-automated package completion

CDE's primary limitation is that it can only package up files accessed on executed program paths. Thus, programs run from within a CDE package will fail when executing paths that access new files (e.g., libraries, configuration files) that the original execution(s) did not access.

Unfortunately, *no automatic tool* (static or dynamic) can find and package up all the files required to successfully execute all possible program paths, since that problem is undecidable in general. Similarly, it is also impossible to automatically quantify how "complete" a CDE package is or determine what files are missing, since every file-related system call instruction could be invoked with complex or non-deterministic arguments. For example, the Python interpreter executable has only one `dlopen` call site for dynamically loading extension modules, but that `dlopen` could be called many times with different dynamically-generated string arguments derived from script variables or configuration files.

There are two ways to cope with this package incompleteness problem. First, if the user executes additional program paths, then CDE will add new files into the same `cde-package/` directory. However, making repeated executions can get tedious, and it is unclear how many or which paths are necessary to complete the package[2].

Another way to make CDE packages more complete is by manually copying additional files and subdirectories into `cde-package/cde-root/`. For example, while executing a Python script, CDE might automatically copy the few Python standard library files it accesses into, say, `cde-package/cde-root/usr/lib/python/`. To complete the package, the user could copy the entire `/usr/lib/python/` directory into `cde-package/cde-root/` so that *all* Python libraries are present. A user can usually make his/her package complete by copying only a few crucial directories into the package, since programs store all of their files in several top-level directories (see Section 3.3).

However, programs also depend on shared libraries that reside in system-wide directories like `/lib` and `/usr/lib`. Copying all the contents of those directories into a package results in lots of wasted disk space. In Section 3.2, we present an automatic heuristic technique that finds nearly all shared libraries that a program requires and copies them into the package.

[2] similar to trying to achieve 100% coverage during software testing

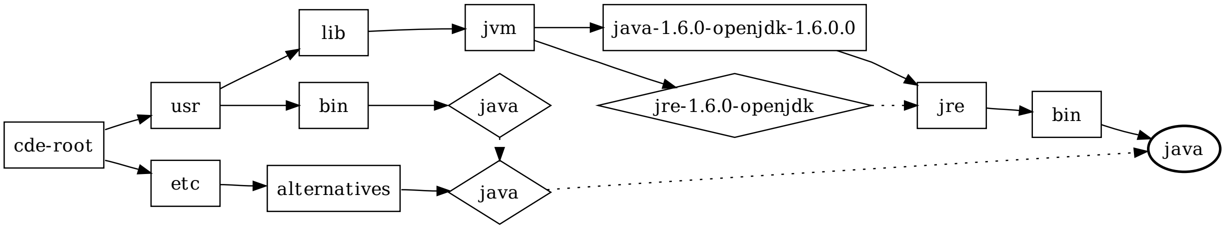


Figure 7: The result of using OKAPI to deep-copy a single `/usr/bin/java` file into `cde-root/`, preserving the exact symlink structure from the original directory tree. Boxes are directories (solid arrows point to their contents), diamonds are symlinks (dashed arrows point to their targets), and the bold ellipse is the actual `java` executable file.

3.1 The OKAPI utility for deep file copying

Before describing our heuristics for completing CDE packages, we first introduce a utility library we built called OKAPI (pronounced "*oh-copy*"), which performs detailed copying of files, directories, and symlinks. OKAPI does one seemingly-simple task that turns out to be tricky in practice: copying a filesystem entity (i.e., a file, directory, or symlink) from one directory to another while fully preserving its original sub-directory and symlink structure (a process that we call *deep-copying*). CDE uses OKAPI to copy files into the `cde-root/` sub-directory when creating a new package, and the support scripts of Sections 3.2 and 3.3 also use OKAPI.

For example, suppose that CDE needs to copy the `/usr/bin/java` executable file into `cde-root/` when it is packaging a Java application. The straightforward way to do this is to use the standard `mkdir` and `cp` utilities. Figure 6 shows the resulting sub-directory structure within `cde-root/`, with the boxes representing directories and the bold ellipse representing the copy of the `java` executable file located at `cde-root/usr/bin/java`. However, it turns out that if CDE were to use this straightforward copying method, the Java application would *fail to run* from within the CDE package! This failure occurs because the `java` executable introspects its own path and uses it as the search path for finding the Java standard libraries. On our Fedora Core 9 machine, the Java standard libraries are actually installed in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0`, so when `java` reads its own path as `/usr/bin/java`, it cannot possibly use that path to find its standard libraries.

In order for Java applications to properly run from within CDE packages, all of their constituent files must be "deep-copied" into the package while replicating their original sub-directory and symlink structures. Figure 7 illustrates the complexity of deep-copying a single file, `/usr/bin/java`, into `cde-root/`. The diamond-shaped nodes represent symlinks, and the dashed arrows point to their targets. Notice how `/usr/bin/java` is a symlink to `/etc/alternatives/java`, which is itself a symlink to `/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`. Another complicating factor is that `/usr/lib/jvm/jre-1.6.0-openjdk` is itself a symlink to the `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/` directory, so the actual `java` executable resides in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/`. Java can only find its standard libraries when these paths are all faithfully replicated within the CDE package.

The OKAPI utility library automatically performs the deep-copying required to generate the filesystem structure of Figure 7. Its interface is as simple as ordinary `cp`: The caller simply requests for a path to be copied into a target directory, and OKAPI faithfully replicates the sub-directory and symlink structure.

OKAPI performs one additional task: rewriting the contents of symlinks to transform absolute path targets into relative path targets within the destination directory (e.g., `cde-root/`). In our example, `/usr/bin/java` is a symlink to `/etc/alternatives/java`. However, OKAPI cannot simply create the `cde-root/usr/bin/java` symlink to also point to `/etc/alternatives/java`, since that target path is outside of `cde-root/`. Instead, OKAPI must rewrite the symlink target so that it actually refers to `../../etc/alternatives/java`, which is a relative path that points to `cde-root/etc/alternatives/java`.

The details of this particular example are not important, but the high-level message that Figure 7 conveys is that deep-copying even a single file can lead to the creation of over a dozen sub-directories and (possibly-rewritten) symlinks. The problem that OKAPI solves is not Java-specific; we have observed that many real-world Linux applications fail to run from within CDE packages unless their files are deep-copied in this detailed way.

OKAPI is also available as a free standalone command-line tool [1]. To our knowledge, no other Linux file copying tool (e.g., `cp`, `rsync`) can perform the deep-copying and symlink rewriting that OKAPI does.

3.2 Heuristics for copying shared libraries

When Linux starts executing a dynamically-linked executable, the dynamic linker (e.g., `ld-linux*.so*`) finds and loads all shared libraries that are listed in a special `.dynamic` section within the executable file. Running the `ldd` command on the executable shows these start-up library dependencies. When CDE is executing a target program to create a package, CDE finds all of these dependencies as well because they are loaded at start-up time via `open` system calls.

However, programs sometimes load shared libraries in the middle of execution using, say, the `dlopen` function. This run-time loading occurs mostly in GUI programs with a plug-in or extension architecture. For example, when the user instructs Firefox to visit a web page with a Flash animation, Firefox will use `dlopen` to load the Adobe Flash Player shared library. `ldd` will not find that dependency since it is not hard-coded in the `.dynamic` section of the Firefox executable, and CDE will only find that dependency if the user actually visits a Flash-enabled web page while creating a package for Firefox.

We have created a simple heuristic-based script that finds most or all shared libraries that a program requires[3]. The user first creates a base CDE package by executing the target program once (or a few times) and then runs our script, which works as follows:

1. Find all ELF binaries (executables and shared libraries) within the package using the Linux `find` and `file` utilities.
2. For each binary, find all constant strings using the `strings` utility, and look for strings containing "`.so`" since those are likely to be shared libraries.
3. Call the `locate` utility on each candidate shared library string, which returns the *full absolute paths* of all installed shared libraries that match each string.
4. Use OKAPI to copy each library into the package.
5. Repeat this process until no new libraries are found.

This heuristic technique works well in practice because programs often list all of their dependent shared libraries in string *constants* within their binaries. The main exception occurs in dynamic languages like Python or MATLAB, whose programs often dynamically generate shared library paths based on the contents of scripts and configuration files.

Another limitation of this technique is that it is overly conservative and can create larger-than-needed packages, since the `locate` utility can find more libraries than the target program actually needs.

3.3 OKAPI-based directory copying script

In general, running an application once under CDE monitoring only packages up a subset of all required files. In our experience, the easiest way to make CDE packages complete is to copy entire sub-directories into the package. To facilitate this process, we created a script that repeatedly calls OKAPI to copy an entire directory at a time into `cde-root/`, automatically following symlinks to other directories and recursively copying as needed.

Although this approach might seem primitive, it is effective in practice because applications often store all of their files in a few top-level directories. When a user inspects the directory structure within `cde-root/`, it is usually obvious where the application's files reside. Thus, the user can run our OKAPI-based script to copy the entirety of those directories into the package.

Evaluation: To demonstrate the efficacy of this approach, we have created complete self-contained CDE packages for six of the largest and most popular Linux applications. For each app, we made an initial packaging run with `cde`, inspected the package contents, and copied at most three directories into the package. The entire packaging process took several minutes of human effort per application. Here are our full results:

- **AbiWord** is a free alternative to Microsoft Word. After an initial packaging run, we saw that some plug-ins were included in the `cde-root/usr/lib/abiword-2.8/plugins` and `cde-root/usr/lib/goffice/0.8.1/plugins` directories. Thus, we copied the entirety of those two original directories into `cde-root/` to complete its package, thereby including all AbiWord plug-ins.
- **Eclipse** is a sophisticated IDE and software development platform. We completed its package by copying the `/usr/lib/eclipse` and `/usr/share/eclipse` directories into `cde-root/`.
- **Firefox** is a popular web browser. We completed its package by copying `/usr/lib/firefox-3.6.18` and `/usr/lib/firefox-addons` into `cde-root/` (plus another directory for the third-party Adobe Flash player plug-in).
- **GIMP** is a sophisticated graphics editing tool. We completed its package by copying `/usr/lib/gimp/2.0` and `/usr/share/gimp/2.0`.
- **Google Earth** is an interactive 3D mapping application. We completed its package by copying `/opt/google/earth` into `cde-root/`.
- **OpenOffice.org** is a free alternative to the Microsoft Office productivity suite. We completed its package by copying the `/usr/lib/openoffice` directory into `cde-root/`.

[3] always a superset of the shared libraries that `ldd` finds

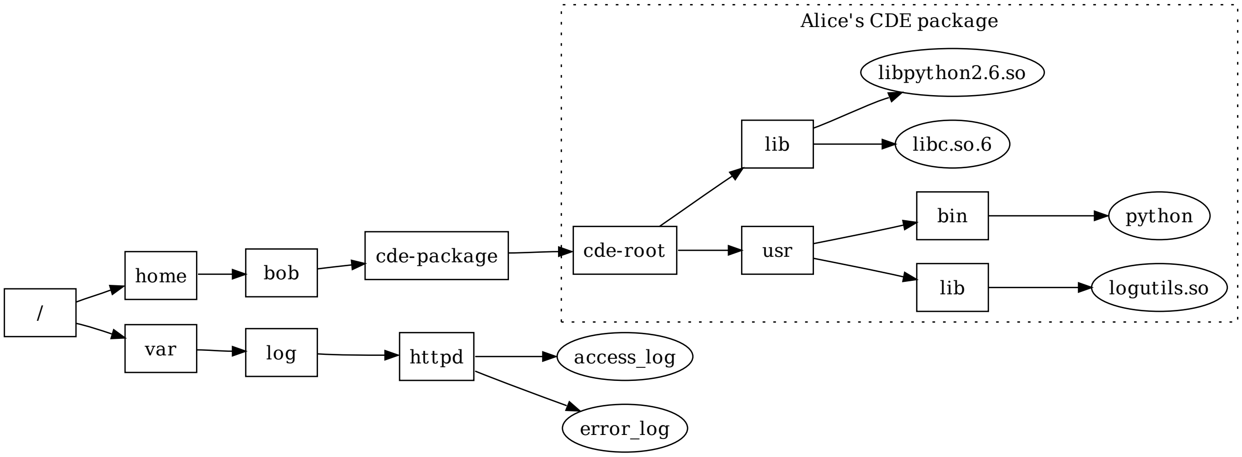


Figure 8: Example filesystem layout on Bob's machine after he receives a CDE package from Alice (boxes are directories, ellipses are files). CDE's seamless execution mode enables Bob to run Alice's packaged script on the log files in `/var/log/httpd/` without first moving those files inside of `cde-root/`.

4 Seamless execution mode

When executing a program from within a package, `cde-exec` redirects all file accesses into the package by default, thereby creating a chroot-like sandbox with `cde-package/cde-root/` as the pseudo-root directory (see Figure 3, Step 3). However, unlike chroot, CDE does not require root access to run, and its sandbox policies are flexible and user-customizable [20].

This default chroot-like execution mode is fine for running self-contained GUI applications like games or web browsers, but it is a somewhat awkward way to run most types of UNIX-style command-line programs that system administrators, developers, and hackers often prefer. If users are running, say, a compiler or command-line image processing utility from within a CDE package, they would need to first move their input data files into the package, run the target program using `cde-exec`, and then move the resulting output data files back out of the package, which is a cumbersome process.

In our Alice-and-Bob example from Section 2 (see Figure 3), if Bob wants to run Alice's anomaly detection script on his own log data (e.g., `bob.log`), he needs to first move his data file inside of `cde-package/cde-root/`, change into the appropriate sub-directory deep within the package, and then run:

```
cde-exec python detect_anomalies.py bob.log
```

In contrast, if Bob had actually installed the proper version of Python and its required extension modules on his machine, then he could run Alice's script from *anywhere* on his filesystem with no restrictions. Some CDE users wanted CDE-packaged programs to behave just like regularly-installed programs rather than requiring input files to be moved inside of a `cde-package/cde-root/` sandbox, so we implemented a new *seamless execution mode* that largely achieves this goal.

Seamless execution mode works using a simple heuristic: If `cde-exec` is being invoked from a directory *not* in the CDE package (i.e., from somewhere else on the user's filesystem), then only redirect a path into `cde-package/cde-root/` if the file that the path refers to actually exists within the package. Otherwise simply leave the path unmodified so that the program can access the file normally. No user intervention is needed in the common case.

The intuition behind why this heuristic works is that when programs request to load libraries and other mandatory components, those files must exist within the package, so their paths are redirected. On the other hand, when programs request to load an input file passed via, say, a command-line argument, that file does not exist within the package, so the original path is used to retrieve it from the native filesystem.

In the example shown in Figure 8, if Bob ran Alice's script to analyze an arbitrary log file on his machine (e.g., his web server log, `/var/log/httpd/access_log`), then `cde-exec` will redirect Python's request for its own libraries (e.g., `/lib/libpython2.6.so` and `/usr/lib/logutils.so`) inside of `cde-root/` since those files exist within the package, but `cde-exec` will *not* redirect `/var/log/httpd/access_log` and instead load the real file from its original location.

Seamless execution mode fails when the user wants the packaged program to access a file from the native filesystem, but an identically-named file actually exists within the package. In the above example, if `cde-package/cde-root/var/`

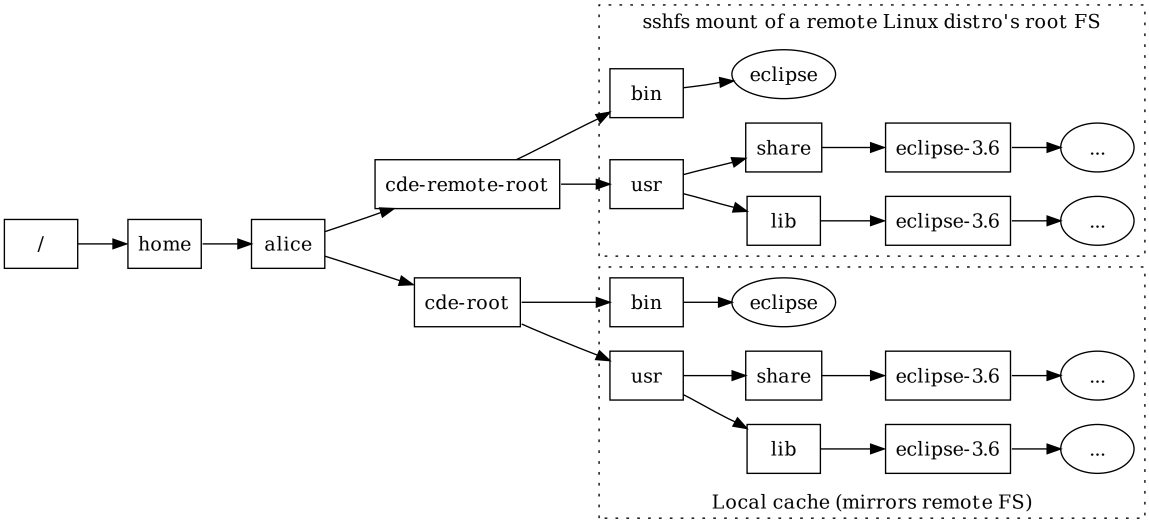


Figure 9: An example use of CDE's streaming mode to run Eclipse 3.6 on any Linux machine without installation. `cde-exec` fetches all dependencies on-demand from a remote Linux distro and stores them in a local cache.

`log/httpd/access_log` existed, then that file would be processed by the Python script instead of `/var/log/httpd/access_log`. There is no automated way to resolve such name conflicts, but `cde-exec` provides a "verbose mode" where it prints out a log of what paths were redirected within the package. The user can inspect that log and then manually write redirection/ignore rules in a configuration file to control which paths `cde-exec` redirects into `cde-root/`. For instance, the user could tell `cde-exec` to *not* redirect any paths starting with `/var/log/httpd/*`.

Using seamless execution mode, our users have been able to run software such as programming language interpreters and compilers, scientific research tools, and sysadmin scripts from CDE packages and have them behave just like regularly-installed programs.

5 On-demand application streaming

We now introduce a new application streaming mode where CDE users can instantly run any Linux application on-demand without having to create, transfer, or install any packages. Figure 2 shows a high-level architectural overview. The basic idea is that a system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). When a user wants to run some application that is available on a particular distro, they use sshfs (an ssh-based network filesystem [9]) to mount the root directory of that distro into a special `cde-remote-root/` mountpoint on their Linux machine. Then the user can use CDE's streaming mode to run any application from that distro locally on their own machine.

5.1 Implementation and example

Figure 9 shows an example of streaming mode. Let's say that Alice wants to run the Eclipse 3.6 IDE on her Linux machine, but the particular distro she is using makes it difficult to obtain all the dependencies required to install Eclipse 3.6. Rather than suffering through dependency hell, Alice can simply connect to a distro in the farm that contains Eclipse 3.6 and then use CDE's streaming mode to "harvest" the required dependencies on-demand.

Alice first mounts the root directory of the remote distro at `cde-remote-root/`. Then she runs "`cde-exec -s eclipse`" (`-s` activates streaming mode). `cde-exec` finds and executes `cde-remote-root/bin/eclipse`. When that executable requests shared libraries, plug-ins, or any other files, `cde-exec` will redirect the respective paths into `cde-remote-root/`, thereby executing the version of Eclipse 3.6 that resides in the cloud distro. However, note that the application is running locally on Alice's machine, not in the cloud.

An astute reader will immediately realize that running applications in this manner can be slow, since files are being accessed from a remote server. While sshfs performs some caching, we have found that it does not work well enough in practice. Thus, we have implemented our own caching layer within CDE: When a remote file is accessed from `cde-remote-root/`, `cde-exec` uses OKAPI to make a deep-copy into a local `cde-root/` directory and then redirects that file's path into `cde-root/`. In streaming mode, `cde-root/` initially starts out empty and then fills up with a subset of files from `cde-remote-root/` that the target program has accessed.

To avoid unnecessary filesystem accesses, CDE's cache also keeps a list of file paths that the target program tried to access from the remote server, even keeping paths for *non-existent files*. On subsequent runs, when the program tries to access one of those paths, `cde-exec` will redirect the path into the local `cde-root/` cache. It is vital to track non-existent files since programs often try to access non-existent files at start-up while doing, say, a search for shared libraries by probing a list of directories in a search path. If CDE did not track non-existent files, then the program would still access the directory entries on the remote server before discovering that those files still do not exist, thus slowing down performance.

With this cache in place, the first time an application is run, all of its dependencies must be downloaded, which could take several seconds to minutes. This one-time delay is unavoidable. However, subsequent runs simply use the files already in the local cache, so they execute at regular `cde-exec` speeds. An added bonus is that even running a *different* application for the first time might still result in some cache hits for, say, generic libraries like `libc`, so the entire application does not need to be downloaded.

Finally, the package incompleteness problem faced by regular CDE (see Section 3) no longer exists in streaming mode. When the target application needs to access new files that do not yet exist in the local cache (e.g., Alice loads a new Eclipse plug-in), those files are transparently fetched from the remote server and cached.

5.2 Synergy with package managers

Nearly all Linux users are currently running one particular distro with one default package manager that they use to install software. For instance, Ubuntu users must use APT, Fedora users must use YUM, SUSE users must use Zypper, Gentoo users must use Portage, etc. Moreover, different releases of the *same* distro contain different software package versions, since distro maintainers add, upgrade, and delete packages in each new release[4].

As long as a piece of software and all of its dependencies are present within the package manager of the exact distro release that a user happens to be using, then installation is trivial. However, as soon as even one dependency cannot be found within the package manager, then users must revert to the arduous task of compiling from source (or configuring a custom package manager).

CDE's streaming mode frees Linux users from this single-distro restriction and allows them to run software that is available within the package manager of any distro in the cloud distro farm. The system administrator is responsible for setting up the farm and provisioning access rights (e.g., ssh keys) to users. Then users can directly install packages in any cloud distro and stream the desired applications to run locally on their own machines.

Philosophically, CDE's streaming mode maximizes user freedom since users are now free to run any application in any package manager from the comfort of their own machines, regardless of which distro they choose to use. CDE complements traditional package managers by leveraging all of the work that the maintainers of each distro have already done and opening up access to users of all other distros. This synergy can potentially eliminate quasi-religious squabbles and flame-wars over the virtues of competing distros or package management systems. Such fighting is unnecessary since CDE allows users to freely choose from amongst all of them.

6 Real-world use cases

Since we released the first version of CDE on November 9, 2010, it has been downloaded at least 3,000 times as of September 2011 [1]. We cannot track how many people have directly checked out its source code from GitHub, though. We have exchanged hundreds of emails with CDE users and discovered six salient real-world use cases as a result of these discussions. Table 1 shows that we used 16 CDE packages, mostly sent in by our users, as benchmarks in the experiments reported in Section 7. They contain software written in diverse programming languages and frameworks. We now summarize the use case categories and benchmarks (highlighted in **bold**).

Distributing research software: The creators of two research tools found CDE online and used it to create portable packages that they uploaded to their websites:

The website for **`graph-tool`**, a Python/C++ module for analyzing graphs, lists these (direct) dependencies: "GCC 4.2 or above, Boost libraries, Python 2.5 or above, expat library, NumPy and SciPy Python modules, GCAL C++ geometry library, and Graphviz with Python bindings enabled." [11] Unsurprisingly, lots of people had trouble compiling it: 47% of all messages on its mailing list (137 out of 289) were questions related to compilation problems. The author of `graph-tool` used CDE to automatically create a portable package (containing 149 shared libraries and 1909 total files) and uploaded it to his website so that users no longer needed to suffer through the pain of manually compiling it.

`arachni`, a Ruby-based tool that audits web application security [10], requires six hard-to-compile Ruby extension modules, some of which depend on versions of Ruby and libraries that are not available in the pack-

[4] We once tried installing a machine learning application that depended on the `libcv` computer vision library. The required `libcv` version was found in the APT repository on Ubuntu 10.04, but it was not found in the repositories on the two immediately neighboring Ubuntu releases: 9.10 and 10.10.

Package name	Description	Dependencies	Creator
Distributing research software			
arachni	Web app. security scanner framework [10]	Ruby (+ extensions)	security researcher
graph-tool	Lib. for manipulation & analysis of graphs [11]	Python, C++, Boost	math researcher
pads	Language for processing ad-hoc data [19]	Perl, ML, Lex, Yacc	self
saturn	Static program analysis framework [13]	Perl, ML, Berkeley DB	self
Running production software on incompatible distros			
meld	Interactive visual diff and merge tool for text	Python, GTK+	software engineer
bio-menace	Classic video game within a MS-DOS emulator	DOSBox, SDL	game enthusiast
google-earth	3D interactive map application by Google	shell scripts, OpenGL	self
Creating reproducible computational experiments			
kpiece	Robot motion planning algorithm [26]	C++, OpenGL	robotics researcher
gadm	Genetic algorithm for social networks [21]	C++, make, R	self
Deploying computations to cluster or cloud			
ztopo	Batch processing of topological map images	C++, Qt	graduate student
klee	Automatic bug finder & test case generator [16]	C++, LLVM, μClibc	self
Submitting executable bug reports			
coq-bug-2443	Incorrect output by Coq proof assistant [2]	ML, Coq	bug reporter
gcc-bug-46651	Causes GCC compiler to segfault [3]	gcc	bug reporter
llvm-bug-8679	Runs LLVM compiler out of memory [5]	C++, LLVM	bug reporter
Collaborating on class programming projects			
email-search	Natural language semantic email search	Python, NLTK, Octave	college student
vr-osg	3D virtual reality modeling of home appliances	C++, OpenSceneGraph	college student

Table 1: CDE packages used as benchmarks in our experiments, grouped by use cases. 'self' in the 'Creator' column means package was created by the author; all other packages created by CDE users (mostly people we have never met).

age managers of most modern Linux distributions. Its creator, a security researcher, created and uploaded CDE packages and then sent us a grateful email describing how much effort CDE saved him: *"My guess is that it would take me half the time of the development process to create a self-contained package by hand; which would be an unacceptable and truly scary scenario."*

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as tarballs of source code: **pads** [19] and **saturn** [13]. 44% of the messages on the pads mailing list (38 / 87) were questions related to troubles with compiling it (22% for saturn). Once we successfully compiled these projects (after a few hours of improvising our own hacks since the instructions were outdated), we created CDE packages by running their regression test suites, so that others do not need to suffer through the compilation process.

Even the saturn team leader admitted in a public email, *"As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release."* [7] In contrast, our CDE packages are largely immune to "bit rot" (until the user-kernel ABI changes) because they contain all required dependencies.

Running software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions, especially when system upgrades are infeasible. For example, an engineer at Cisco wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port, and then ran the tools from within the packages on his work machines. He sent us one of the packages, which we used as a benchmark: the **meld** visual diff tool.

Hobbyists applied CDE in a similar way: A game enthusiast could only run a classic game (**`bio-menace`**) within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the game on his other machines. We also helped a user create a portable package for the Google Earth 3D map application (**`google-earth`**), so he can now run it on older distros whose libraries are incompatible with Google Earth.

Reproducible computational experiments: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. In the past few years, science journals and CS conferences (e.g., SIGMOD, FSE) have encouraged authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard for people to set up all of the (often-undocumented) dependencies required to re-run experiments. In fact, it can even be difficult to re-run *one's own experiments* in the future, due to inevitable OS and library upgrades. To ensure that he could later re-run and adjust experiments in response to reviewer critiques for a paper submission [16], our groupmate Cristian took the hard drive out of his computer at paper submission time and archived it in his drawer!

In our experience, the results of many computational science experiments can be reproduced within CDE packages since the programs are output-deterministic [15], always producing the same outputs (e.g., statistics, graphs) for a given input. A robotics researcher used CDE to make the experiments for his motion planning paper (**`kpiece`**) [26] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**`gadm`**) [21].

Deploying computations to cluster or cloud: People working on computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism. However, before they can deploy their computations to a cluster or cloud computing (e.g., Amazon EC2), they must first install all of the required executables and dependent libraries on the cluster machines. At best, this process is tedious and time-consuming; at worst, it can be impossible, since regular users often do not have root access on cluster machines.

A user can create a self-contained package using CDE on their desktop machine and then execute that package on the cluster or cloud (possibly many instances in parallel), without needing to install any dependencies or to get root access on the remote machines. For instance, our colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps (**`ztopo`**). However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive **`klee`** [16] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines. Klee can be hard to compile since it depends on LLVM, which is very picky about specific versions of GCC and other build tools being present before it will compile.

Submitting executable bug reports: Bug reporting is a tedious manual process: Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries, (e.g., *"I'm running Java version 1.6.0_13, Eclipse SDK Version 3.6.1, . . . "*), and maybe attaching an input file that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier and more reliable solution: The bug reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. The developer can also modify the input file and command-line parameters and then re-execute, in order to investigate the bug's root cause.

To show that this technique works, we asked people who recently reported bugs to popular open-source projects to use CDE to create executable bug reports. Three volunteers sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (**`coq-bug-2443`**) [2], one that segfaults the GCC compiler (**`gcc-bug-46651`**) [3], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**`llvm-bug-8679`**) [5].

Since CDE is not a record-replay tool, it is not guaranteed to reproduce non-deterministic bugs. However, at least it allows the developer to run the exact versions of the faulting executables and dependent libraries.

Collaborating on class programming projects: Two users sent us CDE packages they created for collaborating on class assignments. Rahul, a Stanford grad student, was using NLTK [22], a Python module for natural language processing, to build a semantic email search engine (**`email-search`**) for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their Linux machines due to conflicting library versions and dependency hell. This meant that they could only run one instance of the project at a time on Rahul's laptop for query testing and debugging. When Rahul discovered CDE, he created a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Joshua, an undergrad from Mexico, emailed us a similar story about how he used CDE to collaborate on and demo his virtual reality class project (**`vr-osg`**).

7 Evaluation

7.1 Evaluating CDE package portability

To show that CDE packages can successfully execute on a wide range of Linux distros and kernel versions, we tested our benchmark packages on popular distros from the past 5 years. We installed fresh copies of these distros (listed with the versions and release dates of their kernels) on a 3GHz Intel Xeon x86-64 machine:

- Sep 2006 — CentOS 5.5 (Linux 2.6.18)
- Oct 2007 — Fedora Core 8 (Linux 2.6.23)
- Oct 2008 — openSUSE 11.1 (Linux 2.6.27)
- Sep 2009 — Ubuntu 9.10 (Linux 2.6.31)
- Feb 2010 — Mandriva Free Spring (Linux 2.6.33)
- Aug 2010 — Linux Mint 10 (Linux 2.6.35)

We installed 32-bit and 64-bit versions of each distro and executed our 32-bit benchmark packages (those created on 32-bit distros) on the 32-bit versions, and our 64-bit packages on the 64-bit versions. Although all of these distros reside on one physical machine, none of our benchmark packages were created on that machine: CDE users created most of the packages, and we made sure to create our own packages on other machines.

Results: Out of the 96 unique configurations we tested (16 CDE packages each run on 6 distros), all executions succeeded except for one[5]. By "succeeded", we mean that the programs ran correctly, as far as we could observe: Batch programs generated identical outputs across distros; regression tests passed; we could interact normally with the GUI programs; and we could reproduce the symptoms of the executable bug reports.

In addition, we were able to successfully execute all of our 32-bit packages on the 64-bit versions of CentOS, Mandriva, and openSUSE (the other 64-bit distros did not support executing 32-bit binaries).

In sum, we were able to use CDE to successfully execute a diverse set of programs (Table 1) "out-of-the-box" on a variety of Linux distributions from the past 5 years, without performing any installation or configuration.

7.2 Comparing against a one-click installer

To show that the level of portability that CDE enables is substantive, we compare CDE against a representative one-click installer for a commercial application. We installed and ran Google Earth (Version 5.2.1, Sep 2010) on our 6 test distros using the official 32-bit installer from Google. Here is what happened on each distro:

- CentOS (Linux 2.6.18) — installs fine but Google Earth crashes upon start-up with variants of this error message repeated several times, because the GNU Standard C++ Library on this OS is too old:

  ```
  /usr/lib/libstdc++.so.6:
  version `GLIBCXX_3.4.9' not found
  (required by ./libgoogleearth_free.so)
  ```

- Fedora (Linux 2.6.23) — same error as CentOS
- openSUSE (Linux 2.6.27) — installs and runs fine
- Ubuntu (Linux 2.6.31) — installs and runs fine
- Mandriva (Linux 2.6.33) — installs fine but Google Earth crashes upon start-up with this error message because a required graphics library is missing:

  ```
  error while loading shared libraries:
  libGL.so.1: cannot open shared object
  file: No such file or directory
  ```

- Linux Mint (Linux 2.6.35) — installer program crashes with this cryptic error message because the XML processing library on this OS is *too new* and thus incompatible with the installer:

  ```
  setup.data/setup.xml:1: parser error :
    Document is empty
  setup.data/setup.xml:1: parser error :
    Start tag expected, '<' not found
  Couldn't load 'setup.data/setup.xml'
  ```

In summary, on 4 out of our 6 test distros, a binary installer for the fifth major release of Google Earth (v5.2.1), a popular commercial application developed by a well-known software company, failed in its *sole goal* of allowing the user to run the application, despite advertising that it should work on any Linux 2.6 machine.

If a team of professional Linux developers had this much trouble getting a widely-used commercial application to be portable across distros, then it is unreasonable to expect researchers or hobbyists to be able to easily create portable Linux packages for their prototypes.

In contrast, once we were able to install Google Earth on just *one machine* (Dell desktop running Ubuntu 8.04), we ran it under CDE supervision to create a self-contained package, copied the package to all 6 test distros, and successfully ran Google Earth on all of them without any installation or configuration.

[5] `vr-osg` failed on Fedora Core 8 with a known error related to graphics drivers.

Benchmark	Native run time	CDE slowdown pack	CDE slowdown exec
`400.perlbench`	23.7s	3.0%	**2.5%**
`401.bzip2`	47.3s	0.2%	**0.1%**
`403.gcc`	0.93s	2.7%	**2.2%**
`410.bwaves`	185.7s	0.2%	**0.3%**
`416.gamess`	129.9s	0.1%	**0%**
`429.mcf`	16.2s	2.7%	**0%**
`433.milc`	15.1s	2%	**0.6%**
`434.zeusmp`	36.3s	0%	**0%**
`435.gromacs`	133.9s	0.3%	**0.1%**
`436.cactusADM`	26.1s	0%	**0%**
`437.leslie3d`	136.0s	0.1%	**0%**
`444.namd`	13.9s	3%	**0.3%**
`445.gobmk`	97.5s	0.4%	**0.2%**
`447.dealII`	28.7s	0.5%	**0.2%**
`450.soplex`	5.7s	2.2%	**1.8%**
`453.povray`	7.8s	2.2%	**1.9%**
`454.calculix`	1.4s	5%	**4%**
`456.hmmer`	48.2s	0.2%	**0.1%**
`458.sjeng`	121.4s	0%	**0.2%**
`459.GemsFDTD`	55.2s	0.2%	**1.6%**
`462.libquantum`	1.8s	2%	**0.6%**
`464.h264ref`	87.2s	0%	**0%**
`465.tonto`	229.9s	0.8%	**0.4%**
`470.lbm`	31.9s	0%	**0%**
`471.omnetpp`	51.0s	0.7%	**0.6%**
`473.astar`	103.7s	0.2%	**0%**
`481.wrf`	161.6s	0.2%	**0%**
`482.sphinx3`	8.8s	3%	**0%**
`483.xalancbmk`	58.0s	1.2%	**1.8%**

Table 2: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package on the SPEC CPU2006 benchmarks, using the "train" datasets.

7.3 Evaluating CDE run-time slowdown

The primary drawback of executing a CDE-packaged application is the run-time slowdown due to extra user-kernel context switches. Every time the target application issues a system call, the kernel makes two extra context switches to enter and then exit the `cde-exec` monitoring process, respectively. `cde-exec` performs some computations to calculate path redirections, but its run-time overhead is dominated by context switching[6].

We informally evaluated the run-time slowdown of `cde` and `cde-exec` on 34 diverse Linux applications. In summary, for CPU-bound applications, CDE causes almost no slowdown, but for I/O-bound applications, CDE causes a slowdown of up to 30%.

Command	Native time	CDE slowdown pack	CDE slowdown exec	Syscalls per sec
`gadm` (algorithm)	4187s	0%†	**0%**†	19
`pads` (inferencer)	18.6s	3%†	**1%**†	478
`klee`	7.9s	31%	**2%**†	260
`gadm` (make plots)	7.2s	8%	**2%**†	544
`gadm` (C++ comp)	8.5s	17%	**5%**	1459
`saturn`	222.7s	18%	**18%**	6477
`google-earth`	12.5s	65%	**19%**	7938
`pads` (compiler)	1.7s	59%	**28%**	6969

Table 3: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package. Each entry reports the mean taken over 5 runs; standard deviations are negligible. Slowdowns marked with † are *not* statistically significant at $p < 0.01$ according to a t-test.

We first ran CDE on the entire SPEC CPU2006 benchmark suite (both integer and floating-point benchmarks) [8]. We chose this suite because it contains CPU-bound applications that are representative of the types of programs that computational scientists and other researchers are likely to run with CDE. For instance, SPEC CPU2006 contains benchmarks for video compression, molecular dynamics simulation, image ray-tracing, combinatorial optimization, and speech recognition.

We ran these experiments on a Dell machine with a 2.67GHz Intel Xeon CPU running a 64-bit Ubuntu 10.04 distro (Linux 2.6.32). Each trial was run three times, but the variances in running times were negligible.

Table 2 shows the percentage slowdowns incurred by using `cde` to create each package (the 'pack' column) and by using `cde-exec` to execute each package (the 'exec' column). The 'exec' column slowdowns are shown in **bold** since they are more important for our users: A package is only created once but executed multiple times. In sum, slowdowns ranged from non-existent to 4%, which is unsurprising since the SPEC CPU2006 benchmarks were designed to be CPU-bound and not make much use of system calls.

To test more realistic I/O-bound applications, we measured running times for executing the following commands in the five CDE packages that we created (those labeled with "self" in the "Creator" column of Table 1):

- `pads` — Compile a PADS [19] specification into C code (the "`pads` (compiler)" row in Table 3), and then infer a specification from a data file (the "`pads` (inferencer)" row in Table 3).
- `gadm` — Reproduce the GADM experiment [21]: Compile its C++ source code ('C++ comp'), run genetic algorithm ('algorithm'), and use the R statistics software to visualize output data ('make plots').

[6] Disabling path redirection still results in similar overheads.

- `google-earth` — Measure startup time by launching it and then quitting as soon as the initial Earth image finishes rendering and stabilizes.
- `klee` — Use Klee [16] to symbolically execute a C target program (a STUN server) for 100,000 instructions, which generates 21 test cases.
- `saturn` — Run the regression test suite, which contains 69 tests (each is a static program analysis).

We measured the following on a Dell desktop (2GHz Intel x86, 32-bit) running Ubuntu 8.04 (Linux 2.6.24): number of seconds it took to run the original command ('Native time'), percent slowdown vs. native when running a command with `cde` to create a package ('pack'), and percent slowdown when executing the command from within a CDE package with `cde-exec` ('exec'). We ran each benchmark five times under each condition and report mean running times. We used an *independent two-group t-test* [17] to determine whether each slowdown was statistically significant (i.e., whether the means of two sets of runs differed by a non-trivial amount).

Table 3 shows that the more system calls a program issues per second, the more CDE causes it to slow down due to the extra context switches. Creating a CDE package ('pack' column) is slower than executing a program within a package ('exec' column) because CDE must create new sub-directories and copy files into the package.

CDE execution slowdowns ranged from negligible (not statistically significant) to 30%, depending on system call frequency. As expected, CPU-bound workloads like the `gadm` genetic algorithm and the `pads` inferencer machine learning algorithm had almost no slowdown, while those that were more I/O- and network-intensive (e.g., `google-earth`) had the largest slowdowns.

When using CDE to run GUI applications, we did not notice any loss in interactivity due to the slowdowns. When we navigated around the 3D maps within the `google-earth` GUI, we felt that the CDE-packaged version was just as responsive as the native version. When we ran GUI programs from CDE packages that users sent to us (the `bio-menace` game, `meld` visual diff tool, and `vr-osg`), we also did not perceive any visible lag.

The main caveat of these experiments is that they are informal and meant to characterize "typical-case" behavior rather than being stress tests of worst-case behavior. One could imagine developing adversarial I/O intensive benchmarks that issue tens or hundreds of thousands of system calls per second, which would lead to greater slowdowns. We have not run such experiments yet.

Finally, we also ran some informal performance tests of `cde-exec`'s seamless execution mode. As expected, there were no noticeable differences in running times versus regular `cde-exec`, since the context-switching overhead dominates `cde-exec` computation overhead.

8 Related work

We know of no published system that automatically creates portable software packages *in situ* from a live running machine like CDE does. Existing tools for creating self-contained applications all require the user to manually specify dependencies at package creation time. For example, Mac OS X programmers can create application bundles using Apple's developer tools IDE [6]. Research prototypes like PDS [14], which creates self-contained Windows apps, and the Collective [23], which aggregates a set of software into a portable *virtual appliance*, also require the user to manually specify dependencies.

VMware ThinApp is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software [12]. Unlike CDE, ThinApp cannot be used to create packages from existing software already installed on a live machine, which is our most common use case.

Package management systems are often used to install open-source software and their dependencies. Generic package managers exist for all major operating systems (e.g., RPM for Linux, MacPorts for Mac OS X, Cygwin for Windows), and specialized package managers exist for ecosystems surrounding many programming languages (e.g., CPAN for Perl, RubyGems for Ruby) [4].

From the package creator's perspective, it takes time and expertise to manually bundle up one's software and list all dependencies so that it can be integrated into a specific package management system. A banal but tricky detail that package creators must worry about is adhering to platform-specific idioms for pathnames and avoiding hard-coding non-portable paths into their programs [25]. In contrast, creating a CDE package is as easy as running the target program, and hard-coded paths are fine since `cde-exec` redirects all file accesses into the package.

From the user's perspective, package managers work great as long as the *exact* desired versions of software exist within the system. However, version mismatches and conflicts are common frustrations, and installing new software can lead to a library upgrade that breaks existing software [18]. The Nix package manager is a research project that tries to eliminate dependency conflicts via stricter versioning, but it still requires package creators to manually specify dependencies at creation time [18]. In contrast, CDE packages can be run without any installation, configuration, or risk of breaking existing software.

Virtual machine snapshots achieve CDE's main goal of capturing all dependencies required to execute a set of programs on another machine. However, they require the user to always be working within a VM from the start of a project (or else re-install all of their software within a new VM). Also, VM snapshot disk images are (by definition) larger than the corresponding CDE packages since they must also contain the OS kernel and other extraneous applications. CDE is a more lightweight solution because it enables users to create and run packages natively on their own machines rather than through a VM.

9 Discussion and conclusions

Our design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to run on other Linux machines with as little effort as possible. However, we are not proposing CDE as a replacement for traditional software installation. CDE packages have a number of limitations. Most notably,

- They are not guaranteed to be complete.
- Their constituent shared libraries are "frozen" and do not receive regular security updates. (Static linking also shares this limitation.)
- They run slower than native applications due to `ptrace` overhead. We measured slowdowns of up to 28% in our informal experiments (§7.3), but slowdowns can be worse for I/O-heavy programs.

Software engineers who are releasing production-quality software should obviously take the time to create and test one-click installers or integrate with package managers. But for the millions of system administrators, research scientists, prototype designers, programming course students and teachers, and hobby hackers who just want to deploy their *ad-hoc* software as quickly as possible, CDE can emulate many of the benefits of traditional software distribution with much less required labor: In just minutes, users can create a base CDE package by running their program under CDE supervision, use our *semi-automated heuristic tools* to make the package complete, deploy to the target Linux machine, and then execute it in *seamless execution mode* to make the target program behave like it was installed normally.

In particular, we believe that the lightweight nature of CDE makes it a useful tool in the Linux system administrator's toolbox. Sysadmins need to rapidly and effectively respond to emergencies, hack together scripts and other utilities on-demand, and run diagnostics without compromising the integrity of production machines. Ad-hoc scripts are notoriously brittle and non-portable across Linux distros due to differences in interpreter versions (e.g., bash vs. dash shell, Python 2.x vs. 3.x), system libraries, and availability of the often-obscure programs that the scripts invoke. Encapsulating scripts and their dependencies within a CDE package can make them portable across distros and minor kernel versions; we have been able to take CDE packages created on 2010-era Linux distros and run them on 2006-era distros [20].

Lessons learned: We would like to conclude by sharing some generalizable system design lessons that we learned throughout the past year of developing CDE.

- First and foremost, start with a conceptually-clear core idea, make it work for basic non-trivial cases, document the still-unimplemented tricky cases, launch your system, and then get feedback from real users. User feedback is by far the easiest way for you to discover what bugs are important to fix and what new features to add next.
- A simple and appealing quick-start webpage guide and screencast video demo are essential for attracting new users. No potential user is going to read through dozens of pages of an academic research paper before deciding to try your system. In short, even hackers need to learn to be great salespeople.
- To maximize your system's usefulness, you must design it to be easy-to-use for beginners but also to allow advanced users to customize it to their liking. One way to accomplish this goal is to have well-designed default settings, which can be adjusted via command-line options or configuration files. The defaults must work well "out-of-the-box" without any tuning, or else beginners will get frustrated.
- Resist the urge to add new features just because they're "interesting", "cool", or "potentially useful". Only add new features when there are compelling real users who demand it. Instead, focus your development efforts on fixing bugs, writing more test cases, improving your documentation, and, most importantly, attracting new users.
- Users are the best sources of bug reports, since they often stress your system in ways that you could have never imagined. Whenever a user reports a bug, try to create a representative minimal test case and add it to your regression test suite.
- If a user has a conceptual misunderstanding of how your system works, then think hard about how you can improve your documentation or default settings to eliminate this misunderstanding.

In sum, get real users, make them happy, and have fun!

Acknowledgments

Special thanks to Dawson Engler for supporting my efforts on this project throughout the past year, to Bill Howe for inspiring me to develop CDE's streaming mode, to Yaroslav Bulatov for being a wonderful CDE power-user and advocate, to Federico D. Sacerdoti (my paper shepherd) for his insightful critiques that greatly improved the prose, and finally to the NSF fellowship for funding this portion of my graduate studies.

References

[1] CDE public source code repository, https://github.com/pgbovine/CDE.

[2] Coq proof assistant: Bug 2443, http://coq.inria.fr/bugs/show_bug.cgi?id=2443.

[3] GCC compiler: Bug 46651, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651.

[4] List of software package management systems, http://en.wikipedia.org/wiki/List_of_software_package_management_systems.

[5] LLVM compiler: Bug 8679, http://llvm.org/bugs/show_bug.cgi?id=8679.

[6] Mac OS X Bundle Programming Guide: Introduction, http://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html.

[7] Saturn online discussion thread, https://mailman.stanford.edu/pipermail/saturn-discuss/2009-August/000174.html.

[8] Spec cpu2006 benchmarks, http://www.spec.org/cpu2006/.

[9] SSH Filesystem, http://fuse.sourceforge.net/sshfs.html.

[10] arachni project home page, https://github.com/Zapotek/arachni.

[11] graph-tool project home page, http://projects.skewed.de/graph-tool/.

[12] VMware ThinApp User's Guide, http://www.vmware.com/pdf/thinapp46_manual.pdf.

[13] AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. An overview of the Saturn project. PASTE '07, ACM, pp. 43–48.

[14] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: A virtual execution environment for software deployment. VEE '05, ACM, pp. 175–185.

[15] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. SOSP '09, ACM, pp. 193–206.

[16] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, USENIX Association, pp. 209–224.

[17] CHAMBERS, J. M. *Statistical Models in S.* CRC Press, Inc., Boca Raton, FL, USA, 1991.

[18] DOLSTRA, E., DE JONGE, M., AND VISSER, E. Nix: A safe and policy-free system for software deployment. In *LISA '04, the 18th USENIX conference on system administration* (2004).

[19] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. PLDI '05, ACM, pp. 295–304.

[20] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages (short paper). In *USENIX Annual Technical Conference* (June 2011).

[21] LAHIRI, M., AND CEBRIAN, M. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI Press.

[22] LOPER, E., AND BIRD, S. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics* (2002).

[23] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *LISA '03, the 17th USENIX conference on system administration* (2003).

[24] SCAFFIDI, C., SHAW, M., AND MYERS, B. Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (2005).

[25] STAELIN, C. mkpkg: A software packaging tool. In *LISA '98, the 12th USENIX conference on system administration* (1998).

[26] SUCAN, I. A., AND KAVRAKI, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics* (2008), pp. 449–464.

Improving Virtual Appliance Management through Virtual Layered File Systems

Shaya Potter   Jason Nieh
Computer Science Department
Columbia University
{spotter, nieh}@cs.columbia.edu

Abstract

Managing many computers is difficult. Recent virtualization trends exacerbate this problem by making it easy to create and deploy multiple virtual appliances per physical machine, each of which can be configured with different applications and utilities. This results in a huge scaling problem for large organizations as management overhead grows linearly with the number of appliances.

To address this problem, we introduce Strata, a system that combines unioning file system and package management semantics to enable more efficient creation, provisioning and management of virtual appliances. Unlike traditional systems that depend on monolithic file systems, Strata uses a collection of individual sotware layers that are composed together into the Virtual Layered File System (VLFS) to provide the traditional file system view. Individual layers are maintained in a central repository and shared across all file systems that use them. Layer changes and upgrades only need to be done once in the repository and are then automatically propagated to all virtual appliances, resulting in management overhead independent of the number of appliances. Our Strata Linux prototype requires only a single loadable kernel module providing the VLFS support and doesn't require any application or source code level kernel modifications. Using this prototype, we demonstrate how Strata enables fast system provisioning, simplifies system maintenance and upgrades, speeds system recovery from security exploits, and incurs only modest performance overhead.

1 Introduction

A key problem organizations face is how to efficiently provision and maintain the large number of machines deployed throughout their organizations. This problem is exemplified by the growing adoption and use of virtual appliances (VAs). VAs are pre-built software bundles run inside virtual machines (VMs). Since VAs are often tailored to a specific application, these configurations can be smaller and simpler, potentially resulting in reduced resource requirements and more secure deployments.

While VAs simplify application deployment and decrease hardware costs, they can tremendously increase the human cost of administering these machines As VAs are cloned and modified, organizations that once had a few hardware machines to manage now find themselves juggling many more VAs with diverse system configurations and software installations.

This causes many management problems. First, as these VAs share a lot of common data, they are inefficient to store, as there are multiple copies of many common files. Second, by increasing the number of systems in use, we increase the number of systems needing security updates. Finally, machine sprawl, especially non actively maintained machines, can give attackers many places to hide as well as make attack detection more difficult. Instead of a single actively used machine, administrators now have to monitor many irregularly used machines.

Many approaches have been used to address these problems, including diskless clients [5], traditional package management systems [6, 1], copy-on-write disks [9], deduplication [16] and new VM storage formats [12, 4]. Unfortunately, they suffer from various drawbacks that limit their utility and effectiveness in practice. They either do not directly help with management, incur management overheads that grow linearly with the number of VAs, or require a homogenous configuration, eliminating the main advantages of VAs.

The fundamental problem with previous approaches is that they are based on a monolithic file system or block device. These file systems and block devices address their data at the block layer and are simply used as a storage entity. They have no direct concept of what the file system contains or how it is modified. However, managing VAs is essentially done by making changes to the file system. As a result, any upgrade or maintenance operation needs to be done to each VA independently, even when they all need the same maintenance.

We present Strata, a novel system that integrates file system unioning with package management semantics and uses the combination to solve VA management problems. Strata makes VA creation and provisioning fast. It automates the regular maintenance and upgrades that must be performed on provisioned VA instances. Finally, it improves the ability to detect and recover from security exploits.

Strata achieves this by providing three architectural components: layers, layer repositories, and the Virtual Layered File System (VLFS). A layer is a set of files that are installed and upgraded as a unit. Layers are analogous to software packages in package management systems. Like software packages, a layer may require other layers to function correctly, just as applications often require various system libraries to run. Strata associates dependency information with each layer that defines relationships among distinct layers. Unlike software packages, which are installed into each VA's file system, layers can be shared directly among multiple VAs.

Layer repositories are used to store layers centrally within a virtualization infrastructure, enabling them to be shared among multiple VAs. Layers are updated and maintained in the layer repository. When a new version of an application becomes available, due to added features or a security patch, a new layer is added to the repository. Different versions of the same application may be available through different layers in the layer repository. The layer repository is typically stored in a shared storage infrastructure accessible by the VAs, such as an SAN. Storing layers on the SAN does not impact VA performance because an SAN is where a traditional VA's monolithic file system is stored.

The VLFS implements Strata's unioning mechanism and provides the file system for each VA. Like a traditional unioning file system, it is a collection of individual layers composed into a single view. It enables, a file system to be built out of many shared read-only layers while providing each file system with its own private read-write layer to contain all file system modifications that occur during runtime. In addition, it provides new semantics that enable unioning file systems to be used as the basis for package management type system. These include how layers get added and removed from the union structure as well as how the file system handles files deleted from a read-only layer.

Strata, by combining the unioning and package management semantics, provides a number of management benefits. First, Strata is able to create and provision VAs quickly and easily. By leveraging each layer's dependency information, Strata allows an administrator to quickly create template VAs by only needing to explicitly select the application and tool layers of interest. These template VAs can then be instantly provisioned by end users as no copying or on demand paging is needed to instantiate any file system as all the layers are accessed from the shared layer repository.

Second, Strata automates upgrades and maintenance of provisioned VAs. If a layer contains a bug to be fixed, the administrator only updates the template VA with a replacement layer containing the fix. This automatically informs all provisioned VAs to incorporate the updated layer into their VLFS's namespace view, thereby requiring the fix to only be done once no matter how many VAs are deployed. Unlike traditional VAs, who are updated by replacing an entire file system [12, 4], Strata does not need to be rebooted to have these changes take effect. Unlike package management, all VLFS changes are atomic as no time is spent deleting and copying files.

Finally, this semantic allows Strata to easily recover VAs in the presence of security exploits. The VLFS allows Strata to distinguish between files installed via its package manager, which are stored in a shared read-only layer, and the changes made over time, which are stored in the private read-write layer. If a VA is compromised, the modifications will be confined to the VLFS's private read-write layer, thereby making the changes easy to both identify and remove.

We have implemented a Strata Linux prototype without any application or source code operating system kernel changes and provide the VLFS as a loadable kernel module. We show that by combining traditional package management with file system unioning we provide powerful new functionality that can help automate many machine management tasks. We have used our prototype with VMware ESX virtualization infrastructure to create and manipulate a variety of desktop and server VAs to demonstrate its utility for system provisioning, system maintenance and upgrades, and system recovery. Our experimental results show that Strata can provision VAs in only a few seconds, can upgrade a farm of fifty VAs with several different configurations in less than two minutes, and has scalable storage requirements and modest file system performance overhead.

2 Related Work

The most common way to provision and maintain machines today is using the package management system built into the operating system [6, 1]. Package management provides a number of benefits. First, it divides the installable software into independent chunks called packages. When one wants to install a piece of software or upgrade an already installed piece of software, all one has to do is download and install that single item. Second, these packages can include dependency information that instructs the system about what other packages must be installed with this package. This enables tools [2, 10] to automatically determine the entire set of packages one needs to install when one wants to install a piece of software, making it significantly easier for an end-user to install software.

However, package managers view the file system as a simple container for files and not as a partner in the management of the machine. This causes them to suffer from a number of flaws in their management of large numbers of VAs. They are not space or time efficient, as each provisioned VA requires time-consuming copying of many megabytes or gigabytes into each VA's file system. These inefficiencies affect both provisioning and updating of a system as a lot of time is spent, downloading, extracting and installing the individual packages into the many independent VAs.

As the package manager does not work in partnership with the file system, the file system does not distinguish between a file installed from a package and a file modified or created in the course of usage. Specialized tools are needed to traverse the entire file system to determine if a file has been modified and therefore compromised. Finally, package management systems work in the context of a running system to modify the file system directly. These tools often cannot not work if the VA is suspended or turned off.

For local scenarios, the size and time efficiencies of provisioning a VA can be improved by utilizing copy-on-write (COW) disks, such as QEMU's QCOW2 [9] format. These enables VAs to be provisioned quickly, as little data has to be written to disk immediately due to the COW property. However, once provisioned, each COW copy is now fully independent from the original, is equivalent to a regular copy, and therefore suffers from all the same maintenance problems as a regular VA. Even if the original disk image is updated, the changes would be incompatible with the cloned COW images. This is because COW disks operate at the block level. As files get modified, they use different blocks on their underlying device. Therefore, it is likely that the original and cloned COW images address the same blocks for different pieces of data. For similar reasons, COW disks do not help with VA creation, as multiple COW disks cannot be combined together into a single disk image.

Both the Collective [4] and Ventana [12] attempt to solve the VA maintenance problem by building upon COW concepts. Both systems enable VAs to be provisioned quickly by performing a COW copy of each VA's `system` file system. However, they suffer from the fact that they manage this file system at either the block device or monolithic file system level, providing users with only a single file system. While ideally an administrator could supply a single homogeneous shared image for all users, in practice, users want access to many heterogeneous images that must be maintained independently and therefore increase the administrator's work. The same is true for VAs provisioned by the end user, while they both enable the VAs to maintain a separate disk from the shared system disk that persists beyond upgrades.

Mirage [17] attempts to improve the disk image sprawl problem by introducing a new storage format, the Mirage Index Format (MIF), to enumerate what files belong to a package. However, it does not help with the actual image sprawl in regard to machine maintenance, because each machine reconstituted by Mirage still has a fully independent file system, as each image has its own personal copy. Although each provisioned machine can be tracked, they are now independent entities and suffer from the same problems as a traditional VA.

Stork [3] improves on package management for container-based systems by enabling containers to hard link to an underlying shared file system so that files are only stored once across all containers. By design, it cannot help with managing independent machines, virtual machines, or VAs, because hard links are a function internal to a specific file system and not usable between separate file systems.

Union file systems [11, 19] provide the ability to compose multiple different file namespaces into a single view. Unioning file systems are commonly used to provide a COW file system from a read-only copy, such as with Live-CDs. However, unioning file system by themselves do not directly help with VA management, as the underlying file system has to be maintained using regular tools. Strata builds upon and leverages this mechanism by improving its ability to handle deleted files as well as managing the layers that belong to the union. This allows Strata to provide a solution that enables efficient provisioning and management of VAs.

Strata focuses on improving virtual appliance management, but the VLFS idea can be used to address other management and security problems as well. For example, our previous work on Apiary [14] demonstrates how the VLFS can be combined with containers to provide a transparent desktop application fault containment architecture that is effective at limiting the damage from exploits to enable quick recovery while being as easy to use as a traditional desktop system.

3 Strata Basics

Figure 1 shows Strata's three architectural components: layers, layer repositories, and VLFSs. A layer is a distinct self-contained set of files that corresponds to a specific functionality. Strata classifies layers into three categories: software layers with self-contained applications and system libraries, configuration layers with configuration file changes for a specific VA, and private layers allowing each provisioned VA to be independent. Layers can be mixed and matched, and may depend on other layers. For example, a single application or system library is not fully independent, but depends on the presence of other layers, such as those that provide needed shared libraries. Strata enables layers to enumerate their dependencies on other layers. This dependency scheme

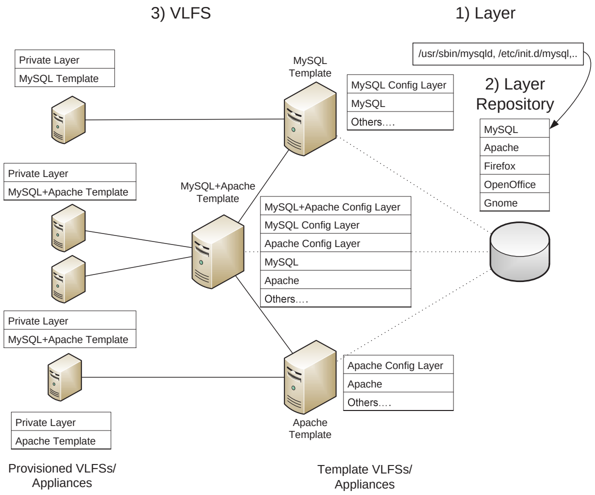


Figure 1: How Strata's Components Fit Together

allows automatic provisioning of a complete, fully consistent file system by selecting the main features desired within the file system.

Layers are provided through layer repositories. As Figure 1 shows, a layer repository is a file system share containing a set of layers made available to VAs. When an update is available, the old layer is not overwritten. Instead, a new version of the layer is created and placed within the repository, making it available to Strata's users. Administrators can also remove layers from the repository, e.g., those with known security holes, to prevent them from being used. Layer repositories are generally stored on centrally managed file systems, such as a SAN or NFS, but they can also be provided by protocols such as FTP and HTTP and mirrored locally. Layers from multiple layer repositories can form a VLFS as long as they are compatible with one another. This allows layers to be provided in a distributed manner. Layers provided by different maintainers can have the same layer names, causing a conflict. This, however, is no different from traditional package management systems as packages with the same package name, but different functionality, can be provided by different package repositories.

As Figure 1 shows, a VLFS is a collection of layers from layer repositories that are composed into a single file system namespace. The layers making up a particular VLFS are defined by the VLFS's layer definition file (LDF), which enumerates all the layers that will be composed into a single VLFS instance. To provision a VLFS, an administrator selects software layers that provide the desired functionality and lists them in the VLFS's LDF.

Within a VLFS, layers are stacked on top of another and composed into a single file system view. An implication of this composition mechanism is that layers on top can obscure files on layers below them, only allowing the contents of the file instance contained within the higher level to be used. This means that files in the private or configuration layers can obscure files in lower layers, such as when one makes a change to a default version of a configuration file located within a software layer. However, to prevent an ambiguous situation from occurring, where the file system's contents depend on the order of the software layers, Strata prevents software layers that contain a subset of the same file from being composed into a single VLFS.

4 Using Strata

Strata's usage model is centered around the usage of layers to quickly create VLFSs for VAs as shown in Figure 1. Strata allows an administrator to compose together layers to form template VAs. These template VAs can be used to form other template appliances that extend their functionality, as well as to provide the VA that end users will provision and use. Strata is designed to be used within the same setup as a traditional VM architecture. This architecture includes a cluster of physical machines that are used to host VM execution as well as a shared SAN that stores all of the VM images. However, instead of storing complete disk images on the SAN, Strata uses the SAN to store the layers that will be used by the VMs it manages.

4.1 Creating Layers and Repositories

Layers are first created and stored in layer repositories. Layer creation is similar to the creation of packages in a traditional package management system, where one builds the software, installs it into a private directory, and turns that directory into a package archive, or in Strata's case, a layer. For instance, to create a layer that contains the MySQL SQL server, the layer maintainer would download the source archive for MySQL, extract it, and build it normally. However, instead of installing it into the system's root directory, one installs it into a virtual root directory that becomes the file system component of this new layer. The layer maintainer then defines the layer's metadata, including its name (`mysql-server` in this case) and an appropriate version number to uniquely identify this layer. Finally, the entire directory structure of the layer is copied into a file system share that provides a layer repository, making the layer available to users of that repository.

4.2 Creating Appliance Templates

Given a layer repository, an administrator can then create template VAs. Creating a template VA involves: (1) Creating the template VA with an identifiable name. (2)

Determining what repositories are available to it. (3) Selecting a set of layers that provide the functionality desired.

To create a template VA that provides a MySQL SQL server, an administrator creates an appliance/VLFS named `sql-server` and selects the layers needed for a fully functional MySQL server file system, most importantly, the mysql-server layer. Strata composes these layers together into the VLFS in a read-only manner along with a read-write private layer, making the VLFS usable within a VM. The administrator boots the VM and makes the appropriate configuration changes to the template VA, storing them within the VLFS's private layer. Finally, the private layer belonging to the template appliance's VLFS is converted into the template's read-only configuration layer by being moved to a SAN file-system that the VAs can only access in a read-only manner. As another example, to create an Apache web server appliance, an administrator creates an appliance/VLFS named `web-server`, and selects the layers required for an Apache web server, most importantly, the layer containing the Apache program.

Strata extends this template model by allowing multiple template VAs to be composed together into a single new template. An administrator can create a new template VA/VLFS, `sql+web-server`, composed of the MySQL and Apache template VAs. The resulting VLFS has the combined set of software layers from both templates, both of their configuration layers, and a new configuration layer containing the configuration state that integrates the two services together, for a total of three configuration layers.

4.3 Provisioning and Running Appliance Instances

In Strata, a VLFS can be created by building off a previously defined VLFS set of layers and combining those layers with a new read-write private layer. Therefore, given previously defined templates, Strata enables VAs to be efficiently and quickly provisioned and deployed by end users. Provisioning a VA involves (1) creating a virtual machine container with a network adapter and an empty virtual disk, (2) using the network adapter's unique MAC address as the machine's identifier for identifying the VLFS created for this machine, and (3) forming the VLFS by referencing the already existing respective template VLFS and combining the template's read-only software and configuration layers with a read-write private layer provided by the VM's virtual disk.

As each VM managed by Strata does not have a physical disk off which to boot, Strata network boots each VM. When the VM boots, its BIOS discovers a network boot server which provides it with a boot image, including a base Strata environment. The VM boots this base environment, which then determines which VLFS should be mounted for the provisioned VM using the MAC address of the machine. Once the proper VLFS is mounted, the machine transitions to using it as its root file system.

4.4 Updating Appliances

Strata upgrades provisioned VAs efficiently using a simple three-step process. First, an updated layer is installed into a shared layer repository. Second, administrators are able to modify the template appliances under their control to incorporate the update. Finally, all provisioned VAs based on that template will automatically incorporate the update as well. Note that updating appliances is much simpler than updating generic machines, as appliances are not independently managed machines. This means that extra software that can conflict with an upgrade will not be installed into a centrally managed appliance. Centrally managed appliance updates are limited to changes to their configuration files and what data files they store.

Strata's updates propagate automatically even if the VA is not currently running. If a provisioned VA is shut down, the VA will compose whatever updates have been applied to its templates automatically, never leaving the file system in a vulnerable state, because it composes its file system afresh each time it boots. If it is suspended, Strata delays the update to when the VA is resumed, as updating layers is a quick task. Updating is significantly quicker than resuming, so this does not add much to its cost.

Furthermore, VAs are upgraded atomically, as Strata adds and removes all the changed layers in a single operation. In contrast, traditional package management system, when upgrading a package, first uninstalls it before reinstalling the newer version. This traditional method leaves the file system in an inconsistent state for a short period of time. For instance, when the libc package is upgraded, its contents are first removed from the file system before being replaced. Any application that tries to execute during the interim will fail to dynamically link because the main library on which it depends is not present within the file system at that moment.

4.5 Improving Security

Strata makes it much easier to manage VAs that have had their security compromised. By dividing a file system into a set of shared read-only layers and storing all file system modifications inside the private read-write layer, Strata separates changes made to the file system via layer management from regular runtime modifications. This enables Strata to easily determine when system files have been compromised, because making a compromise persistent requires the file system be modified, modifying or adding files to the file system to create a compromise will be readily visible in the private layer. This allows Strata to not rely on tools like Tripwire [8] or maintain separate databases to determine if files have been modified from their installed state. Similarly, this check can be run external to the VA, as it just needs access to the private layer, thereby preventing an attacker from disabling it. This reduces management load due to not requiring any external databases be kept in sync with the file system state as it changes. While an attacker could try to compromise files on the shared layers, they would have to exploit the SAN containing the layer repository. In a regular virtualization architecture, if an attacker could exploit the SAN, he would also have access to all

This segregation of modified file system state also enables quick recovery from a compromised system. By sreplacing the VA's private layer with a fresh private layer, the compromised system is immediately fixed and returned to its default, freshly provisioned state. However, unlike reinstalling a system from scratch, replacing the private layer does not require throwing away the contents of the old private layer. Strata enables the layer to be mounted within the file system, enabling administrators to have easy access to the files located within it to move the uncompromised files back to their proper place.

5 Strata Architecture

Strata introduces the concept of a virtual layered file system in place of traditional monolithic file systems. Strata's VLFS allows file systems to be created by composing layers together into a single file system namespace view. Strata allows these layers to be shared by multiple VLFSs in a read-only manner or to remain read-write and private to a single VLFS.

Every VLFS is defined by a layer definition file, which specifies what software layers should be composed together. An LDF is a simple text file that lists the layers and their respective repositories. The LDF's layer list syntax is `repository/layer version` and can be proceeded by an optional modifier command. When an administrator wants to add or remove software from the file system, instead of modifying the file system directly, they modify the LDF by adding or removing the appropriate layers.

```
main/mysql-server 5.0.51a-3

main/base 1
main/libdb4.2 4.2.52-18
main/apt-utils 0.5.28.6
main/liblocale-gettext-perl 1.01-17
main/libtext-charwidth-perl 0.04-1
main/libtext-iconv-perl 1.2-3
main/libtext-wrapi18n-perl 0.06-1
main/debconf 1.4.30.13
main/tcpd 7.6-8
main/libgdbm3 1.8.3-2
main/perl 5.8.4-8
main/psmisc 21.5-1
main/libssl0.9.7 0.9.7e-3
main/liblockfile1 1.06
main/adduser 3.63
main/libreadline4 4.3-11
main/libnet-daemon-perl 0.38-1
main/libplrpc-perl 0.2017-1
main/libdbi-perl 1.46-6
main/ssmtp 2.61-2
=main/mailx 3a8.1.2-0.20040524cvs-4
```

Figure 2: LDF for MySQL Server Template

Figure 2 contains an example LDF for a MySQL SQL server template appliance. The LDF lists each individual layer included in the VLFS along with its corresponding repository. Each layer also has a number indicating which version will be composed into the file system. If an updated layer is made available, the LDF is updated to include the new layer version instead of the old one. If the administrator of the VLFS does not want to update the layer, they can hold a layer at a specific version, with the = syntax element. This is demonstrated by the `mailx` layer in Figure 2, which is being held at the version listed in the LDF.

Strata allows an administrator to select explicitly only the few layers corresponding to the exact functionality desired within the file system. Other layers needed in the file system are implicitly selected by the layers' dependencies as described in Section 5.2. Figure 2 shows how Strata distinguishes between explicitly and implicitly selected layers. Explicitly selected layers are listed first and separated from the implicitly selected layers by a blank line. In this case, the MySQL server has only one explicit layer, mysql-server, but has 21 implicitly selected layers. These include utilities such as Perl and TCP Wrappers (tcpd), as well as libraries such as OpenSSL (libssl). Like most operating systems that require a minimal set of packages to always be installed, Strata also always includes a minimal set of shared layers that are common to all VLFSs that it denotes as `base`. In our Strata prototype, these are the layers that correspond to packages that Debian makes essential and are therefore not removable. Strata also distinguishes explicit layers from implicit layers to allow future reconfigurations to remove one implicit layer in favor of another if dependencies need to change.

When an end user provisions an appliance by cloning a template, an LDF is created for the provisioned VA. Fig-

```
@main/sql-server
```

Figure 3: LDF for Provisioned MySQL Server VA

ure 3 shows an example introducing another syntax element, @, that instructs Strata to reference another VLFS's LDF as the basis for this VLFS. This lets Strata clone the referenced VLFS by including its layers within the new VLFS. In this case, because the user wants only to deploy the SQL server template, this VLFS LDF only has to include the single @ line. In general, a VLFS can reference more than one VLFS template, assuming that layer dependencies allow all the layers to coexist.

5.1 Layers

Strata's layers are composed of three components: metadata files, the layer's file system, and configuration scripts. They are stored on disk as a directory tree named by the layer's name and version. For instance, version 5.0.51a of the MySQL server, with a strata layer version of 3, would be stored under the directory `mysql-server_5.0.51a-3`. Within this directory, Strata defines a `metadata` file, a `filesystem` directory, and a `scripts` directory corresponding to the layer's three components.

The metadata files define the information that describes the layer. This includes its name, version, and dependency information. This information is important to ensure that a VLFS is composed correctly. The `metadata` file contains all the metadata that is specified for the layer. Figure 4 shows an example metadata file. Figure 5 shows the full metadata syntax. The metadata file has a single field per line with two elements, the field type and the field contents. In general, the metadata file's syntax is `Field Type: value`, where value can be either a single entry or a comma-separated list of values.

The layer's file system is a self-contained set of files providing a specific functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the types of files that can be included. They can be regular files, symbolic links, hard links, or device nodes. Similarly, each directory entry can be given whatever permissions are appropriate. A layer can be seen as a directory stored on the shared file system that contains the same file and directory structure that would be created if the individual items were installed into a traditional file system. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but one limitation is that hard links cannot exist between layers.

The layer's configuration scripts are run when a layer is added or removed from a VLFS to allow proper integration of the layer within the VLFS. Although many layers are just a collection of files, other layers need to be integrated into the system as a whole. For example, a layer that provides `mp3` file playing capability should register itself with the system's MIME database to allow programs contained within the layer to be launched automatically when a user wants to play an `mp3` file. Similarly, if the layer were removed, it should remove the programs contained within itself from the MIME database.

```
Layer: mysql-server
Version: 5.0.51a-3
Depends: ..., perl (>= 5.6),
  tcpd (>= 7.6-4),...
```

Figure 4: Metadata for MySQL-Server Layer

```
Layer: Layer Name
Version: Version of Layer Unit
Conflicts: layer1 (opt. constraint), ...
Depends: layer1 (...),
   layer2 (...) | layer3, ...
Pre-Depends: layer1 (...), ...
Provides: virtual_layer, ...
```

Figure 5: Metadata Specification

Strata supports four types of configuration scripts: preremove, post-remove, pre-install, and post-install. If they exist in a layer, the appropriate script is run before or after a layer is added or removed. For example, a preremove script can be used to shut down a daemon before it is actually removed, while a post-remove script can be used to clean up file system modifications in the private layer. Similarly, a pre-install script can ensure that the file system is as the layer expects, while the postinstall script can start daemons included in the layer. The configuration scripts can be written in any scripting language. The layer must include the proper dependencies to ensure that the scripting infrastructure is composed into the file system in order to allow the scripts to run.

5.2 Dependencies

A key Strata metadata element is enumeration of the dependencies that exist between layers. Strata's dependency scheme is heavily influenced by the dependency scheme in Linux distributions such as Debian and Red Hat. In Strata, every layer composed into Strata's VLFS is termed a *layer unit*. Every layer unit is defined by its name and version. Two layer units that have the same name but different layer versions are different units of the same layer. A *layer* refers to the set of layer units of a particular name. Every layer unit in Strata has a set of dependency constraints placed within its metadata. There are four types of dependency constraints: (a) dependency, (b) pre-dependency, (c) conflict and (d) provide.

Dependency and Pre-Dependency: Dependency and pre-dependency constraints are similar in that they require another layer unit to be integrated at the same time as the layer unit that specifies them. They differ only in the order the layer's configuration scripts are executed to integrate them into the VLFS. A regular dependency does not dictate order of integration. A pre-dependency dictates that the dependency has to be integrated before the dependent layer. Figure 4 shows that the MySQL layer depends on TCP Wrappers, (`tcpd`), because it dynamically links against the shared library `libwrap.so.0` provided by TCP Wrappers. MySQL cannot run without this shared library, so the layer units that contain MySQL must depend on a layer unit containing an appropriate version of the shared library. These constraints can also be versioned to further restrict which layer units satisfy the constraint. For example, shared libraries can add functionality that breaks their application binary interface (ABI), breaking in turn any applications that depend on that ABI. Since MySQL is compiled against version 0.7.6 of the libwrap library, the dependency constraint is versioned to ensure that a compatible version of the library is integrated at the same time.

Conflict: Conflict constraints indicate that layer units cannot be integrated into the same VLFS. There are multiple reasons this can occur, but it is generally because they depend on exclusive access to the same operating system resource. This can be a TCP port in the case of an Internet daemon, or two layer units that contain the same file pathnames and therefore would obscure each other. For this reason, Strata defines that two layer units of the same layer are by definition in conflict because they will contain some of the same files.

An example of this constraint occurs when the ABI of a shared library changes without any source code changes, generally due to an ABI change in the tool chain that builds the shared library. Because the ABI has changed, the new version can no longer satisfy any of the previous dependencies. But because nothing else has changed, the file on disk will usually not be renamed either. A new layer must then be created with a different name, ensuring that the library with the new ABI is never used to satisfy an old dependency on the original layer. Because the new layer contains the same files as the old layer, it must conflict with the older layer to ensure that they are not integrated into the same file system.

Provide: Provide dependency constraints introduce virtual layers. A regular layer provides a specific set of files, but a virtual layer indicates that a layer provides a particular piece of general functionality. Layer units that depend on a certain piece of general functionality can depend on a specific virtual layer name in the normal manner, while layer units that provide that functionality will explicitly specify that they do. For example, layer units that provide HTML documentation depend on the presence of a web server to enable a user to view them, but which one is not important. Instead of depending on a particular web server, they depend on the virtual layer name `httpd`. Similarly, layer units containing a web server and obeying system policy for the location of static html content, such as Apache or Boa, are defined to provide the `httpd` virtual layer name and therefore satisfy those dependencies. Unlike regular layer units, virtual layers are not versioned.

Example: Figure 2 shows how dependencies can affect a VLFS in practice. This VLFS has only one explicit layer, mysql-server, but 21 implicitly selected layers. The mysql-server layer itself has a number of direct dependencies, including Perl, TCP Wrappers, and the mailx program. These dependencies in turn depend on the Berkeley DB library and the GNU dbm library, among others. Using its dependency mechanism, Strata is able to automatically resolve all the other layers needed to create a complete file system by specifying just a single layer

Returning to Figure 4, this example defines a subset of the layers that the mysql-server layer requires to be composed into the same VLFS to allow MySQL to run correctly. More generally, Figure 5 shows the complete syntax for the dependency metadata. Provides is the simplest, with only a comma separated list of virtual layer names. Conflicts adds an optional version constraint to each conflicted layer to limit the layer units that are actually in conflict. Depends and Pre-Depends add a boolean OR of multiple layers in their dependency constraints to allow multiple layers to satisfy the dependency.

Resolving Dependencies: To allow an administrator to select only the layers explicitly desired within the VLFS, Strata automatically resolves dependencies to determine which other layers must be included implicitly.

Linux distributions already face this problem and tools have been developed to address it, such as Apt [2] and Smart [10]. To leverage Smart, Strata adopts the same metadata database format that Debian uses for packages for its own layers. In Strata, when an administrator requests that a layer be added to or removed from a template appliance, Smart also evaluates if the operation can succeed and what is the best set of layers to add or remove. Instead of acting directly on the contents of the file system, however, Strata only has to update the template's VLFS's definition file with the set of layers to be composed into the file system.

5.3 Layer Creation

Strata allows layers to be created in two ways. First, Strata allows the .deb packages used by Debian-derived distributions and the .rpm packages used by RedHat-derived distributions to be converted into layers that Strata users can use. Strata converts packages into layers in two steps. First, Strata extracts the relevant metadata from the package, including its name and version. Second, Strata extracts the package's file contents into a private directory that will be the layer's file system components. When using converted packages, Strata leverages the underlying distribution's tools to run the configuration scripts belonging to the newly created layers correctly. Instead of using the distribution's tools to unpack the software package, Strata composes the layers together and uses the distribution's tools as though the packages have already been unpacked. Although Strata is able to convert packages from different Linux distributions, it cannot mix and match them because they are generally ABI incompatible with one another.

More commonly, Strata leverages existing packaging methodologies to simplify the creation of layers from scratch. In a traditional system, when administrators install a set of files, they copy the files into the correct places in the file system using the root of the file system tree as their starting point. For instance, an administrator might run `make install` to install a piece of software compiled on the local machine. But in Strata layer creation is a three step process. First, instead of copying the files into the root of the local file system, the layer creator installs the files into their own specific directory tree. That is, they make a blank directory to hold a new file system tree that is created by having the `make install` copy the files into a tree rooted at that directory, instead of the actual file system root.

Second, the layer maintainer extracts programs that integrate the files into the underlying file system and creates scripts that run when the layer is added to and removed from the file system. Examples of this include integration with Gnome's GConf configuration system, creation of encryption keys, or creation of new local users and groups for new services that are added. This leverages skills that package maintainers in a traditional package management world already have.

Finally, the layer maintainer needs to set up the metadata correctly. Some elements of the metadata, such as the name of the layer and its version, are simple to set, but dependency information can be much harder. But because package management tools have already had to address this issue, Strata is able to leverage the tools they have built. For example, package management systems have created tools that infer dependencies using an executable dynamically linking against shared libraries [15]. Instead of requiring the layer maintainer to enumerate each shared library dependency, we can programmatically determine which shared libraries are required and populate the dependency fields based on those versions of the library currently installed on the system where the layer is being created.

5.4 Layer Repositories

Strata provides local and remote layer repositories. Local layer repositories are provided by locally accessible file system shares made available by a SAN. They contain layer units to be composed into the VLFS. This is similar to a regular virtualization infrastructure in which all the virtual machines' disks are stored on a shared SAN. Each layer unit is stored as its own directory; a local layer repository contains a set of directories, each of which corresponds to a layer unit. The local layer repository's contents are enumerated in a database file providing a flat representation of the metadata of all the layer units present in the repository. The database file is used for making a list of what layers can be installed and their dependency information. By storing the shared layer repository on the SAN, Strata lets layers be shared securely among different users' appliances. Even if the machine hosting the VLFS is compromised, the read-only layers will stay secure, as the SAN will enforce the read-only semantic independently of the VLFS.

Remote layer repositories are similar to local layer repositories, but are not accessible as file system shares. Instead, they are provided over the Internet, by protocols such as FTP and HTTP, and can be mirrored into a local layer repository. Instead of mirroring the entire remote repository, Strata allows on-demand mirroring, where all the layers provided by the remote repository are accessible to the VAs, but must be mirrored to the local mirror before they can be composed into a VLFS. This allows administrators to store only the needed layers while maintaining access to all the layers and updates that the repository provides. Administrators can also filter which layers should be available to prevent end users from using layers that violate administration policy. In general, an administrator will use these remote layer repositories to provide the majority of layers, much as administrators use a publicly managed package repository from a regular Linux distribution.

Layer repositories let Strata operate within an enterprise environment by handling three distinct yet related issues. First, Strata has to ensure that not all end users have access to every layer available within the enterprise. For instance, administrators may want to restrict certain layers to certain end users for licensing or security reasons. Second, as enterprises get larger, they gain levels of administration. Strata must support the creation of an enterprise-wide policy while also enabling small groups within the enterprise to provide more localized administration. Third, larger enterprises supporting multiple operating systems cannot rely on a single repository of layers because of inherent incompatibilities among operating systems.

By allowing a VLFS to use multiple repositories, Strata solves these three problems. First, multiple repositories let administrators compartmentalize layers according to the needs of their end users. By providing end users with access only to needed repositories, organizations prevent their end users from using the other layers. Strata depends on traditional file system access control mechanisms to enforce these permissions. Second, by allowing sub-organizations to set up their own repositories, Strata lets a sub-organization's administrator provide the layers that end users need without requiring intervention by administrators of global repositories. Finally, multiple repositories allow Strata to support multiple operating systems, as each distinct operating system has its own set of layer repositories.

5.5 VLFS Composition

To create a VLFS, Strata has to solve a number of file system-related problems. First, Strata has to support the ability to combine numerous distinct file system layers into a single static view. This is equivalent to installing software into a shared read-only file system. Second, because users expect to treat the VLFS as a normal file system, for instance, by creating and modifying files, Strata has to let VLFSs be fully modifiable. By the same token, users must also be able to delete files that exist on the read-only layer.

By basing the VLFS on top of unioning file systems [11, 19], Strata solves all these problems. Unioning file systems join multiple layers into a single namespace. Unioning file systems have been extended to apply attributes such as read-only and read-write to their layers. The VLFS leverages this property to force shared layers to be read-only, while the private layer remains read-write. If a file from a shared read-only layer is modified, it is copied-on-write (COW) to the private read-write layer before it is modified. For example, Live-CDs use this functionality to provide a modifiable file system on top of the read-only file system provided by the CD. Finally, unioning file systems use white-outs to obscure files located on lower layers. For example, if a file located on a read-only layer is deleted, a white-out file will be created on the private read-write layer. This file is interpreted specially by the file-system and is not revealed to the user while also preventing the user from seeing files with the same name.

But end users need to be able to recover deleted files by reinstalling or upgrading the layer containing them. This is equivalent to deleting a file from a traditional monolithic file system, but reinstalling the package containing the file in order to recover it. Also, Strata supports adding and removing layers dynamically without taking the file system off line. This is equivalent to installing, removing, or upgrading a software package while a monolithic file system is online.

Unlike a traditional file system, where deleted system files can be recovered simply by reinstalling the package containing that file, in Strata, white-outs in the private layer persist and continue to obscure the file even if the layer is replaced. To solve this problem, Strata provides a VLFS with additional writeable layers associated with each read-only shared layer. Instead of containing file data, as does the topmost private writeable layer, these layers just contain white-out marks that will obscure files contained within their associated read-only layer. The user can delete a file located in a shared read-only layer, but the deletion only persists for the lifetime of that particular instance of the layer. When a layer is replaced during an upgrade or reinstall, a new empty white-out layer will be associated with the replacement, thereby removing any preexisting white-outs. In a similar way, Strata handles he case where a file belonging to a shared read-only layer is modified and therefore copied to the VLFS's private read-write layer. Strata provides a *revert* command that lets the owner of a file that has been modified revert the file to its original pristine state. While a regular VLFS *unlink* operation would have removed the modified file from the private layer and created a white-out mark to obscure the original file, *revert* only removes the copy in the private layer, thereby revealing the original below it.

Strata also allows a VLFS to be managed while being used. Some upgrades, specifically of the kernel, will require the VA to be rebooted, but most should be able to occur without taking the VA off line. However, if a layer is removed from a union, the data is effectively removed as well because unions operate only on file system namespaces and not on the data the underlying files contain. If an administrator wants to remove a layer from the VLFS, they must take the VA off line, because layers cannot be removed while in use.

To solve this problem, Strata emulates a traditional monolithic file system. When an administrator deletes a package containing files in use, the processes that are currently using those files will continue to work. This occurs by virtue of `unlink`'s semantic of first removing a file from the file system's namespace, and only removing its data after the file is no longer in use. This lets processes continue to run because the files they need will not be removed until after the process terminates. This creates a semantic in which a currently running program can be using versions of files no longer available to other programs.

Existing package managers use this semantic to allow a system to be upgraded online, and it is widely understood. Strata applies the same semantic to layers. When a layer is removed from a VLFS, Strata marks the layer as `unlinked`, removing it from the file system namespace. Although this layer is no longer part of the file system namespace and thus cannot be used by any operations such as `open` that work on the namespace, it does remain part of the VLFS, enabling data operations such as `read` and `write` to continue working correctly for previously opened files.

6 Experimental Results

We have implemented Strata's VLFS as a loadable kernel module on an unmodified Linux 2.6 series kernel as well as a set of userspace management tools. The file system is a stackable file system and is an extended version of UnionFS [19]. We present experimental results using our Strata Linux prototype to manage various VAs, demonstrating its ability to reduce management costs while incurring only modest performance overhead. Experiments were conducted on VMware ESX 3.0 running on an IBM BladeCenter with 14 IBM HS20 eServer blades with dual 3.06 GHz Intel Xeon CPUs, 2.5 GB RAM, and a Q-Logic Fibre Channel 2312 host bus adapter connected to an IBM ESS Shark SAN with 1 TB of disk space. The blades were connected by a gigabit Ethernet switch. This is a typical virtualization infrastructure in an enterprise computing environment where all virtual machines are centrally stored and run. We compare plain Linux VMs with a virtual block device stored on the SAN and formatted with the ext3 file system to VMs managed by Strata with the layer repository also stored on the SAN. By storing both the plain VM's virtual block device and Strata's layers on the SAN, we eliminate any differences in performance due to hardware architecture.

To measure management costs, we quantify the time taken by two common tasks, provisioning and updating VAs. We quantify the storage and time costs for provisioning many VAs and the performance overhead for running various benchmarks using the VAs. We ran experiments on five VAs: an Apache web server, a MySQL SQL server, a Samba file server, an SSH server providing remote access, and a remote desktop server providing a complete GNOME desktop environment. While the server VAs had relatively few layers, the desktop VA has very many layers. This enables the experiments to show how the VLFS performance scales as the number of layers increases. To provide a basis for comparison, we provisioned these VAs using (1) the normal VMware virtualization infrastructure and plain Debian package management tools, and (2) Strata. To make a conservative comparison to plain VAs and to test larger numbers of plain VAs in parallel, we minimized the disk usage of the VAs. The desktop VA used a 2 GB virtual disk, while all others used a 1 GB virtual disk.

	Apache	MySQL	Samba	SSH	Desktop
Plain	184s	179s	183s	174s	355s
Strata	0.002s	0.002s	0.002s	0.002s	0.002s
QCOW2	0.003s	0.003s	0.003s	0.003s	0.003s

Table 1: VA Provisioning Times

6.1 Reducing Provisioning Times

Table 1 shows how long it takes Strata to provision VAs versus regular and COW copying. To provision a VA using Strata, Strata copies a default VMware VM with an empty sparse virtual disk and provides it with a unique MAC address. It then creates a symbolic link on the shared file system from a file named by the MAC address to the layer definition file that defines the configuration of the VA. When the VA boots, it accesses the file denoted by its MAC address, mounts the VLFS with the appropriate layers, and continues execution from within it. To provision a plain VA using regular methods, we use QEMU's `qemu-img` tool to create both raw copies and COW copies in the QCOW2 disk image format.

Our measurements for all five VAs show that using COW copies and Strata takes about the same amount of time to provision VAs, while creating a raw image takes much longer. Creating a raw image for a VAs takes 3 to almost 6 minutes and is dominated by the cost of copying data to create a new instance of the VA. For larger VAs, these provisioning times would only get worse. In contrast, Strata provisions VAs in only a few milliseconds because a null VMware VM has essentially no data to copy. Layers do not need to be copied, so copying overhead is essentially zero. While COW images can be created in a similar amount of time, they do not provide any of the management benefits of Strata, as each new COW image is independent of the base image from which it was created.

6.2 Reducing Update Times

Table 2 shows how long it takes to update VAs using Strata versus traditional package management. We provisioned ten VA instances each of Apache, MySQL, Samba, SSH, and Desktop for a total of 50 provisioned VAs. All were kept in a suspended state. When a security patch was made available for the `tar` package installed in all the VAs, we updated them [18]. Strata simply updates the layer definition files of the VM templates, which it can do even when the VAs are not active. When the VA is later resumed during normal operation,

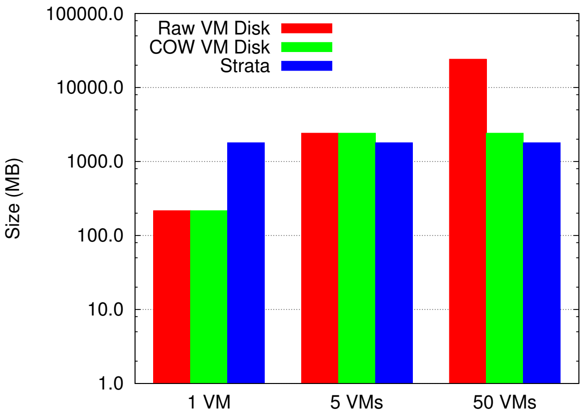


Figure 6: Storage Requirements

it automatically checks to see if the layer definition file has been updated and updates the VLFS namespace view accordingly, an operation that is measured in microseconds. To update a plain VA using normal package management tools, each VA instance needs to be resumed and put on the network. An administrator or script must ssh into each VA, fetch and install the update packages from a local Debian mirror, and finally re-suspend the VA.

Table 2 shows the total average time to update each VA using traditional methods versus Strata. We break down the update time into times to resume the VM, get access to the network, actually perform the update, and re-suspend the VA. The measurements show that the cost of performing an update is dominated by the management overhead of preparing the VAs to be updated and not the update itself. Preparation is itself dominated by getting an IP address and becoming accessible on a busy network. While this cost is not excessive on a quiet network, on a busy network it can take a significant amount of time for the client to get a DHCP address, and for the ARP on the machine controlling the update to find the target machine. The average total time to update each plain VA is about 73 seconds. In contrast, Strata takes only a second to update each VA. As this is an order of magnitude shorter even than resuming the VA, Strata is able to delay the update to a point when the VA will be resumed from standby normally without impacting its ability to quickly respond. Strata provides over 70 times faster update times than traditional package management when managing even a modest number of VAs. Strata's ability to decrease update times would only improve as the number of VAs being managed grows.

	Plain	**Strata**
VM Wake	14.66s	NA
Network	43.72s	NA
Update	10.22s	1.041s
Suspend	3.96s	NA
Total	73.2s	1.041s

Table 2: VA Update Times

6.3 Reducing Storage Costs

Figure 6 shows the total storage space required for different numbers of VAs stored with raw and COW disk images versus Strata. We show the total storage space for 1 Apache VA, 5 VAs corresponding to an Apache, MySQL, Samba, SSH, and Desktop VA, and 50 VAs corresponding to 10 instances of each of the 5 VAs. As expected, for raw images, the total storage space required grows linearly with the number of VA instances. In contrast, the total storage space using COW disk images and Strata is relatively constant and independent of the number of VA instances. For one VA, the storage space required for the disk image is less than the storage space required for Strata, as the layer repository used contains more layers than those used by any one of the VAs. In fact, to run a single VA, the layer repository size could be trimmed down to the same size as the traditional VA.

For larger numbers of VAs, however, Strata provides a substantial reduction in the storage space required, because many VAs share layers and do not require duplicate storage. For 50 VAs, Strata reduces the storage space required by an order of magnitude over the raw disk images. Table 3 shows that there is much duplication among statically provisioned virtual machines, as the layer repository of 405 distinct layers needed to build the different VLFSs for multiple services is basically the same size as the largest service. Although initially Strata does not have an significant storage benefit over COW disk images, as each COW disk image is independent from the version it was created from, it now must be managed independently. This increases storage usage, as the same updates must be independently applied to many independent disk images

6.4 Virtualization Overhead

To measure the virtualization cost of Strata's VLFS, we used a range of micro-benchmarks and real application workloads to measure the performance of our Linux Strata prototype, then compared the results against vanilla Linux systems within a virtual machine. The virtual machine's local file system was formatted with the `Ext3` file system and given read-only access to a SAN partition formatted with `Ext3` as well. We performed each benchmark in each scenario 5 times and provide the average of the results.

Repo	**Apache**	**MySQL**	**Samba**	**SSH**	**Desktop**
1.8GB	217MB	206MB	169MB	127MB	1.7GB
# Layer	43	23	30	12	404
Shared	191MB	162MB	152MB	123MB	169MB
Unique	26MB	44MB	17MB	4MB	1.6GB

Table 3: Layer Repository vs. Static VAs

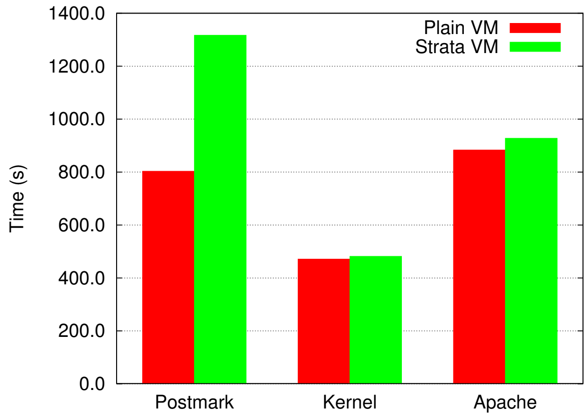


Figure 7: Application Benchmarks

To demonstrate the effect that Strata's VLFS has on system performance, we performed a number of benchmarks. Postmark [7], the first benchmark, is a synthetic test that measures how the system would behave if used as a mail server. Our postmark test operated on files between 512 and 10K bytes, with an initial set of 20,000 files, and performed 200,000 transactions. Postmark is very intensive on a few specific file system operations such as `lookup()`, `create()`, and `unlink()`, because it is constantly creating, opening, and removing files. Figure 7 shows that running this benchmark within a traditional VA is significantly faster than running it in Strata. This is because as Strata composes multiple file system namespaces together, it places significant overhead on those namespace operations.

To demonstrate that postmark's results are not indicative of application oriented performance, we ran two application benchmarks to measure the overhead Strata imposes in a desktop and server VA scenario. The first benchmark was a multi-threaded build of the Linux 2.6.18.6 kernel with two concurrent jobs using the two CPUs allocated to the VM. In all scenarios, we added the 8 software layers required to build a kernel to the layers needed to provide the service. Figure 7 shows that while Strata imposes a slight overhead on the kernel build compared to the underlying file system it uses, the cost is minimal, under 5% at worst.

The second benchmark measured the amount of HTTP transactions that were able to be completed per second to an Apache web server placed under load. We imported the database of a popular guitar tab search engine and used the `http_load` [13] benchmark to continuously performed a set of 20 search queries on the database until 60,000 queries in total have been performed. For each case that did not already contain Apache, we added the appropriate layers to the layer definition file to make Apache available. Figure 7 shows that Strata imposes a minimal overhead of only 5%.

While the Postmark benchmark demonstrated that the VLFS is not an appropriate file system for workloads that are heavy with namespace operations, this shouldn't prevent Strata from being used in those scenarios. No file system is appropriate for all workloads and no system has to be restricted to simply using one file system. One can use Strata and the VLFS to manage the system's configuration while also providing an additional traditional file system on a seperate partition or virtual disk drive to avoid all the overhead the VLFS imposes. This will be very effective for workloads, such as the mail server Postmark is emulating, where namespace heavy operations, such as a mail server processing its mail queue, can be kept on a dedicated file system.

7 Conclusions and Future Work

Strata introduces a new and better way for system administrators to manage virtual appliances using virtual layered file systems. Strata integrates package management semantics with the file system by using a novel form of file system unioning enable dynamic composition of file system layers. This provides powerful new management functionality for provisioning, upgrading, securing, and composing VAs. VAs can be quickly and simply provisioned as no data needs to be copied into place. VAs can be easily upgraded as upgrades can be done once centrally and applied atomically, even for a heterogeneous mix of VAs and when VAs are suspended or turned off. VAs can be more effectively secured since file system modifications are isolated so compromises can be easily identified. VAs can be composed as building blocks to create new systems since file system composition also serves as the core mechanism for creating and maintaining VAs. We have implemented Strata on Linux by providing the VLFS as a loadable kernel modules, but without requiring any source code level kernel changes, and have demonstrated how a Strata can be used in real life situations to improve the ability of system administrators to manage systems. Strata significantly reduces the amount of disk space required for multiple VAs, and allows them to be provisioned almost instantaneously and quickly updated no matter how many are in use.

While Strata just exists as a lab prototype today, there are few steps that could make it significantly more deployable. First, our changes to UnionFS should either be integrated with the current version of UnionFS or with another unioning file system. Second, better tools should be created for managing the creation and management of individual layers. This can include better tools for converting layers from existing Linux distributions as well as new tools that enable layers to be created in a way that takes full advantage of Strata's concepts. Third, the ability to to integrate Strata's concepts with cloud computing infrastructures, such as Eucalyptus, should be investigated.

Acknowledgments

Carolyn Rowland provided helpful comments on earlier drafts of this paper. This work was supported in part by AFOSR MURI grant FA9550-07-1-0527 and NSF grants CNS-1018355, CNS-0914845, and CNS-0905246.

References

[1] The RPM Package Manager. `http://www.rpm.org/`.

[2] B. Byfield. An Apt-Get Primer. `http://www.linux.com/articles/40745`, Dec. 2004.

[3] J. Capps, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: Package Management for Distributed VM Environments. In *The 21st Large Installation System Administration Conference*, Dallas, TX, Nov. 2007.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *The 2nd Symposium on Networked Systems Design and Implementation*, pages 259–272, Boston, MA, Apr. 2005.

[5] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[6] J. Fernandez-Sanguino. Debian GNU/Linux FAQ - Chapter 8 - The Debian Package Management Tools. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[7] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, Inc., Oct. 1997.

[8] G. Kim and E. Spafford. Experience with Tripwire: Using Integrity Checkers for Intrusion Detection. In *The 1994 System Administration, Networking, and Security Conference*, Washington, DC, Apr. 1994.

[9] M. McLoughlin. QCOW2 Image Format. `http://www.gnome.org/~markmc/qcow-image-format.htm`, Sept. 2008.

[10] G. Niemeyer. Smart Package Manager. `http://labix.org/smart`.

[11] J.-S. Pendry and M. K. McKusick. Union Mounts in 4.4BSD-lite. In *The 1995 USENIX Technical Conference*, New Orleans, LA, Jan. 1995.

[12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *3rd Symposium on Networked Systems Design and Implementation*, pages 353–366, San Jose, CA, May 2006.

[13] J. Poskanzer. `http://www.acme.com/software/http_load/`.

[14] S. Potter and J. Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *The 2010 USENIX Annual Technical Conference*, pages 103–116, June 2010.

[15] D. Project. DDP Developers' Manuals. `http://www.debian.org/doc/devel-manuals`.

[16] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *1st USENIX conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.

[17] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *The 2008 ACM International Conference on Virtual Execution Environments*, pages 111–120, Seattle, WA, Mar. 2008.

[18] F. Weimer. DSA-1438-1 Tar – Several Vulnerabilities. `http://www.ua.debian.org/security/2007/dsa-1438`, Dec. 2007.

[19] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1):1–32, Feb. 2006.