Managing Distributions from the Software Vendor's Perspective

James E.J. Bottomley   Paul Clements
SteelEye Technology
Columbia, SC
James.Bottomley@SteelEye.com   Paul.Clements@SteelEye.com

Abstract

This paper describes one vendor's quest for a simple distribution model for a software product which will install on (almost) any Linux distribution. The focus of this paper will primarily be minimising the vendor production and support costs and secondly improving the product's install-ability from the customer perspective. The goal will be to describe a method which achieves a significant lowering of the development costs associated with deploying on multiple flavours of Linux[1]

1 Introduction

For Linux to achieve mass acceptance, and hence world domination, one of the prerequisites is having a sufficient application base (how often have you heard the lament "I would switch to Linux but for this one XYZ application that I need which only runs on NT, Solaris..."). One successful approach to this is emulation used by applications like `wine`[2] and `dosemu`[3]. However, emulators, particularly those of Microsoft operating systems, tend to be unable to emulate the entire operating system (mainly due to secret APIs and undocumented system calls). Furthermore incompatibility problems and inefficiencies with any emulation layer make this a solution which is feasible, but not optimal. True world domination will only be achieved when application vendors can be persuaded to write native applications for Linux.

1.1 Project History and Background

LifeKeeper[1] was initially produced by AT&T in 1992 and ran on a proprietary System V like UNIX OS. The product was bought by SteelEye Technology in 1999 with the object of porting it to Linux (this makes us extremely motivated application vendors).

We began in January 2000 with the aim of making LifeKeeper run on RedHat 6.0. During the initial project, which was scheduled to take about 3 months, RedHat 6.1 was released. "No big deal, we thought, we'll just use 6.1 as the base instead." The first product was finally released, on RedHat 6.1 in June 2000. Shortly thereafter, the Marketing Department agreed to go on a road-show tour with Caldera. "Of course it will work, it's just Linux", they said. "Er...Hang on, let us test it first!", we replied. Sure enough, the pristinely `rpm`[5] packaged software wouldn't even install on Caldera (some of the packages we list as requirements have different names on Caldera). When we forced it to install, it wouldn't start on machine boot up like it was supposed to (the init system is different). Finally, when we started it manually, it dumped core and died. The reason? we have a fixed shared memory area at `0x7fff000`[2], but the eServer kernel was compiled with the 2GB option meaning the shared libraries were actually occupying this area—"It's just Linux—Sure!".

We solved these issues by producing a special "for demonstration purposes only" package for the Caldera road-show. This seemed to be leading us unwittingly in the direction of one package per distribution. However, with the advent of RedHat 7.0 even the embryonic single package per distribution strategy was starting to fail: Some of the applications changed names or plain just wouldn't run with the new version of `glibc`.

This traditional UNIX flavour approach (produce one package for each distinct distribution[3]) wasn't flying well with our Sales Department either: "I just want to sell them one CD, I don't want to give them the third degree about what flavour of Linux they're running—I didn't even know Linux had flavours". And from the engineering perspective, the alternative of a single package containing all distributions looks like a release and a Quality Assurance (QA) nightmare (how do we add a new distribution quickly to our monolith? do we have to test on all distributions when we make a change specific to one?).

Even worse: at the moment the distributions we support all use the RedHat Package Manager (rpm). However, there are distributions (like Debian) that don't use this. How would we get our packages to install on these systems?

1.2 Examples of the Differences Among Linux Distributions

This represents an illustrative rather than a complete list of the differences

1. The Kernel: each distribution vendor rolls up their own patches, so if you require a specific patch, it may be in one vendor's 2.2.14, but not in another vendor's until their version 2.2.16.
2. Drivers: some companies that produce adapter cards (particularly with Fibre Channel) supply their own Linux drivers, but these are not part of the standard kernel (although they may be included in distribution specific kernels). In order to get the correct and tested driver for the product, we must either direct the user to download it from the vendor's website (and possibly compile it) or supply it ourselves.
3. Package names may be different (e.g. RedHat has `nfs-utils` whereas SuSE has `nfsutils`.
4. The init subsystem may be different (e.g. `/etc/rc.d/init.d` on RedHat but `/etc/init.d` on SuSE). (Note that the adoption of the Linux Standard Base will eliminate this issue.)
5. Device names (e.g. `/dev/raw/raw`*n* on RedHat but `/dev/raw`*n* on SuSE).

We must abstract each of these features from the product and construct it in such a way that future but currently unrecognised differences may also be handled without necessitating re-delivery of the entire package.

1.3 The Goals

At this point, we tried to find a way out of what had now become distribution hell by establishing clearly what our requirements were:

1. Be able to add support for a new distribution simply and without changing any of the released packages,
2. only have to run QA on the new distribution and not have to perform regression on any others, and the Marketing one:
3. be able to release on a designated distribution within 60 days of being told to.
4. minimise the costs associated with deploying and maintaining our product on the different distributions.

These goals are deceptive, since the requirement to be able to release, item 3, morphed into be able to release without requiring customers to do any upgrades to their systems, but be tolerant if they had. Now we have to be able to upgrade some distributions to the minimum working versions of software (and kernel) as part of our installation.

2 The Solution

The solution has its origin in the original marketing statement: "Of course it will work, it's just Linux". In theory, underneath all of the value add and differentiation, the essence of the operating system (the kernel, the libraries and the core applications) is effectively all from the same source. Therefore, we theorised, it should be possible to separate our product into two pieces, one large piece which is independent of the distribution (called the core) and a

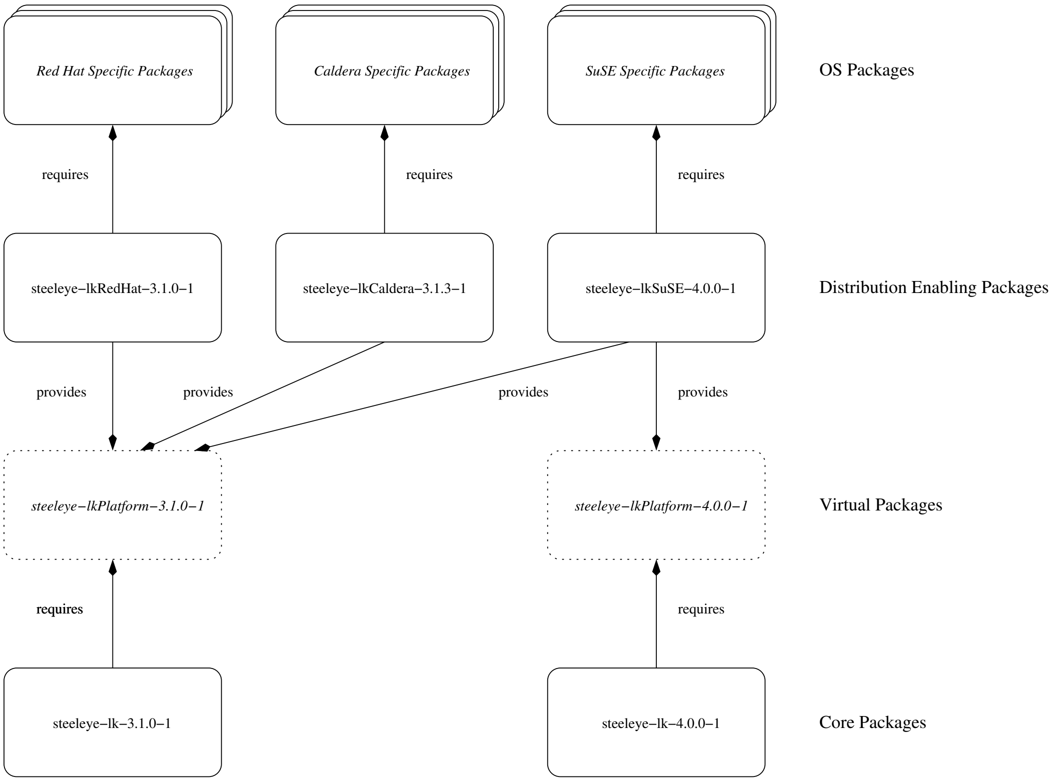

Figure 1: Distribution Enabling Package Scheme

much smaller one containing the distribution dependent components. Then, to support a new distribution, we would be able to produce a new distribution dependent component and deliver it along with our unaltered core package. Because the core and all of the other distribution dependent components are unchanged, to release on a new distribution we would only need to develop the distribution dependent component and then QA the whole thing on the one new distribution.

2.1 Implementing the Solution

The most appealing implementation would be to use the virtual package feature of `rpm`, so we have a single package for each distribution containing the distribution dependent components for that distribution. These distribution packages would all provide the same virtual package, which the distribution independent packages would require (see Figure 1). Unfortunately, `rpm` is not quite flexible enough to be used in exactly this way.

The basic issues are:

- Inability to modify one package from within the installation of another (prevents the installation of required upgrades and patches).
- No interaction allowed (it is unacceptable for us just to modify the installed operating system without explaining to a user what we want to do and asking permission to proceed).

In the end, we decided to create a `setup` script for each distribution to handle the initial setup tasks:

1. Recognise the distribution or exit 1 (all other error exits start from 2 up).
2. Analyse the configuration and compute the upgrades required.
3. Inform the user what needs to be done and request permission to proceed.
4. *Only* if the user permits us to install the required upgrades, install the distribution dependent component providing the virtual package.

Using step 1, we can now place a collection of these on a CD in individual directories and have a top level setup script which invokes each of the subdirectory setups in turn until it finds one that doesn't exit 1. Adding support for a new Linux distribution consists of simply adding an additional subdirectory, including the necessary setup script and distribution `rpm` package, to the CD.

One of the final benefits to this approach is that we can also bundle software that isn't distributed with the target OS on the CD. This came in very handy for us since our GUI is written in Java, but in 2000 only Caldera had a bundled JVM.

Finally, it is necessary to adhere to a set of rules that ensure general portability of the core components. The essentials of these are:

1. Never use absolute paths in scripts. Always use commands by their non-path qualified name so that the `PATH` variable may be updated per distribution to find the correct command.
2. In compiled programs, always use POSIX APIs found in the standard `glibc` if possible rather than non-standard APIs which are found in libraries not present on all distributions.
3. Never "hardcode" fixed parameters. Always load them from some type of defaults file so they may be modified in the field (or by a distribution enabling script) if necessary.
4. Resist the "latest greatest" craze. Beware of trying to code an application for the latest whiz bang feature since most distributions will pick this up more tardily than you anticipate. Unless absolutely essential (and you can work out how to deliver the functionality yourself) always code to APIs which exist on distributions today.

2.2 Distribution Specific Paths

Some utilities can have different names as well as non-standard locations for some daemons (e.g., `nfsd` vs `rpc.nfsd`). We fix this by putting the full absolute paths in the code but always put these paths through a mapping function which is fed from a distribution dependent map file. By default the mapper is the identity, but if it finds an entry in the map file, it will return the distribution specific absolute name. At the application design phase, we must identify all utilities that are likely to vary between distributions and abstract them in this fashion.

2.3 Abstracting the Init System

Since LifeKeeper protects applications, which may themselves have a presence in the init system, we must be able to manipulate other applications' init parameters at a basic level. We find there are four functions we need to abstract:

- `initstart`, `initstop`, `initenable` and `initdisable`.

For shell scripts, we make the distribution specific component provide the implementation (as separately callable commands). For example `initstop apache` is implemented as `/etc/rc.d/init.d/httpd stop` on RedHat, but as `/etc/init.d/apache stop` on SuSE. Note that even the service name has to be translated. As another example, `initdisable apache` becomes `/sbin/chkconfig httpd off` on RedHat but `/sbin/insserv -r apache` on SuSE.

2.4 Packaging the Solution

The packaging for the product consists of an Installation/Support CD and a Core Product CD.

The Installation/Support CD contains the aforementioned distribution setup scripts and packages, as well as any supporting packages and patches that may need to be installed on the system prior to core product installation.

The packaging dependencies are shown in figure 1. The diagram illustrates the OS dependent packages for three different operating sytems (RedHat, Caldera and SuSE) and also shows how we can choose to revoke or maintain backward compatibility as we revise the distribution dependent components: the SuSE package provides the virtual package required by both the 3.01 and 4.0 versions of our product. This use of multiple virtual packages provides us with extremely fine grained tuning of our operating system support.

Additionally, the distribution `rpm` packages themselves have requirements on essential system packages. The `rpm`s will fail to install if the customer's system does not contain these packages. The customer must then install the necessary packages before proceeding.

Once the Installation/Support CD has been successfully installed, it can be assumed that the system is prepared to run the core product. The core product `rpm` packages can then be installed from the Core Product CD.

3 Comparison with the Linux Standard Base

The Linux Standard Base[6] (LSB) is a set of system and application requirements. It defines a common interface between the operating system and applications with the goal to "increase compatibility among Linux distributions and enable software applications to run on any compliant Linux system." LifeKeeper's distribution enabling scheme also attempts to cope with the differences in the system interfaces of the various Linux distributions. However, since we cannot force standards on the distribution vendors, our distribution enabling scheme contains a compatibility layer that provides the (currently lacking) common interface between our application (and those under LifeKeeper's control) and the underlying operating systems. As the Linux Standard Base increases in scope and is increasingly adopted by Linux distribution vendors, our distribution enabling layer should diminish proportionally. Differences in system interfaces will either be eliminated or, at the very least, the compatibility layer (which, at present, must be contained within the applications themselves) will be pushed into the underlying operating system.

Adoption of the LSB is one very important step toward reducing the growing fragmentation among the Linux distributions. If this fragmentation is not addressed, Linux may succumb to the same fate that commercial Unix did. Therefore, adherence to the LSB and other standards (such as POSIX), while burdensome to software vendors, is a necessary evil. Without a common system interface, application vendors must bear the increased burden of scoping out and dealing with differences in underlying operating systems. This results in increased time to market and increased cost for application vendors, and may inhibit the porting of an application altogether.

While the LSB, in its current form, does address many of the needs of application vendors, there are also some areas where the goals of the LSB and our distribution enabling scheme diverge.

3.1 Areas in which the Linux Standard Base Benefits Us

The following items are examples of areas in which adoption of the LSB would allow for the elimination of various compatibility layers that are currently necessary in our product.

- Filesystem Hierarchy Standard (FHS) - Standard filesystem locations would eliminate the need for pathname abstraction mechanisms.
- Standardisation of Init Systems - Standard run level definitions, as well as standard startup script locations, actions, and expected behaviours would reduce the need for an `init` abstraction layer.
- Standardised Packaging Mechanism - The LSB recommends that the Red Hat Package Manager (rpm) be available. This would allow application vendors to provide a single package format that would install on any Linux distribution.

In addition to the above direct benefits, there are also a few indirect benefits of the LSB.

Dealing with bugs and feature changes in external software is generally a challenging task. (With open source, the detection and solution of external bugs is less difficult, but still a problem, nonetheless). Adherence to the LSB ensures that some degree of backward compatibility is maintained for those interfaces defined by the standard. Also, the test suite for the Linux Standard Base is an additional tool to help reduce the introduction of bugs (and other incompatibilities) to system interfaces.

3.2 Areas that the Linux Standard Base Does Not Currently Address

Because the LSB does not (yet) address several areas of difference among the Linux distributions, the task of designing and implementing an LSB-Compliant application is still a difficult one. The following items are some examples of areas that, if added to the LSB specification, would hopefully ease the burden on application vendors:

- Consistent Package Naming and Versioning Scheme - This would allow package dependencies to work correctly across distributions (e.g., the `nfsutils` package on SuSE is `nfs-utils` on other distributions).
- Internationalisation and Localisation - While there are several standards that define consistent library interfaces for localisation and internationalisation, each distribution has a different location and method for specifying localisation policies.
- System Administration - Each distribution has its own set of system configuration files and directories (e.g., `/etc/config.d`, `/etc/sysconfig`, `/etc/rc.config`). There are also distribution specific tools for modifying these files (e.g., `turbo*config`, `SuSEconfig`, `linuxconf`). These differences place an undue burden on vendors who wish to release their software on several distributions, since programmers, testers, and support engineers must learn how to correctly administer each distribution using system administration tools that are unique to that distribution.

3.3 Problem Areas Outside the Scope of Standards

In addition to those areas that can be (or are already) addressed in various standards, there are some areas in which the application vendor has the responsibility for ensuring that compatibility with the system interface has been maintained. In particular, dealing with version (and the accompanying feature) changes in system software is an ongoing task.

Some version differences that must be dealt with in Linux are

- kernel versions 2.2 and 2.4 - Between major releases of the kernel, several interfaces have changed and several subsystems have been (at least) partially rewritten. This requires both application software changes and updates to existing kernel drivers and patches.
- Linux distribution versions (for instance, Red Hat 6.2 and Red Hat 7.1) - There are different installation and system administration utilities, and different default system service and security settings, which must be learned.
- glibc versions 2.1 and 2.2 - Because backward compatibility has been maintained, there is not much change required in our product to deal with these two library versions (apart from upgrading "known bad" versions of the libraries).

4 Analysis and Conclusion

4.1 The Customer Experience

Customers who purchased the initial product (which was specific to RedHat 6.1 only and was available on one CD) did grumble about having a 2 CD set when they upgraded to the next release of the product. However, those who had installed the product themselves (rather than having us send out a Sales Engineer to do it) thought that the 2 CD inconvenience was outweighed by having all necessary upgrades and required packages available on the CD. Additionally, they thought having a script that could work out the missing packages and dependencies and install them in the correct order was an extremely valuable function.

Later customers have made no comment at all (either good or bad) on the 2 CD distribution enabling system. Perhaps this can be considered to be praise for a utility designed to facilitate installation—that it performs its job so quietly and efficiently that users don't really notice it.

4.2 What we got Right

The separation of the core product and the distribution setup and packaging onto separate media has afforded us greater flexibility in our product release cycle. We are able to introduce support for new Linux distributions quickly and asynchronously to the core product release schedule. In addition, modifications to the core product due to distribution differences are rare.

As an example, we recently added TurboLinux 6.5 to our list of supported operating systems. Since TurboLinux is derived from the RedHat distribution, most of the distribution enabling was pretty much plain sailing. The only slight problem was caused by our NFS Application Recovery Kit (used to provide High Availability NFS in a cluster). It turns out that on TurboLinux the RPC portmapper isn't started by default, so the NFS recovery kit was unable to start the NFS daemons. To get around this problem, the list of required daemons was abstracted (which required modifications to the kit) and `portmap` was added to the list of required daemons.

4.3 What We'll Do Better Next Time

In drawing the dividing line between the distribution independent core and the distribution dependent components, we erred on the side of including too much in the core package. Since LifeKeeper essentially provides a High Availability harness for an operating system and its applications, it is much more closely bound to the way the applications it is protecting are bundled with the operating system than a normal program. For this reason, the lack of abstraction of our application control interfaces in the core product has caused considerable problems for us. Modifications to our Application Recovery Kits (the pieces that control and monitor applications under LifeKeeper protection) due to differences between Linux distributions is still fairly common.

In the initial release, the init system abstractions described in section 2.3 were not in place. This caused us quite a bit of anguish over daemons which are very difficult to start and stop except by using the init scripts. We solved this problem on an interim basis by using our file mapping functions described in section 2.2 to translate the location of the init scripts and just assumed they could be called as *`script`* `start | stop` in all distributions. (With the adoption of the Linux Standard Base, we will no longer have to assume that the `start` and `stop` actions are valid, as they will be required for all init scripts).

4.4 Suggested Enhancements to `rpm`

in section 2.1 we outlined the problems trying to support different distributions with `rpm`. In this section we outline a list of enhancements that could be made to `rpm` to improve its utility in handling multiple distributions.

- Dynamic file to package mapping. It would be extremely useful to have `rpm` install certain distribution specific files to an innocuous location (e.g. `/tmp`) and later relocate or delete them as appropriate in the `%post` section. The remaining files should show up in the `rpm` inventory in their new location.
- Dynamic dependencies. Since distribution dependencies are not always known at package build time, it would be useful to be able to modify them in a special section of the `%pre` script and have the system act on all the dependencies including the newly added ones. In addition, the ability to specify conditional or boolean logic in package dependencies would be useful. For example, a package could depend on one package name and version on one distribution, while depending on a different package on another distribution. Or perhaps a package could depend on one of two possible required packages.
- Scripted Interaction. Could do with an acceptable input method (which would be standard for the graphical install tools as well) which would allow the user to enter responses to a series of questions. The responses should be capable of taking default input from a file (so that the install may be customised for non-interactive non-default installs).
- Multiple file ownership. It would be helpful if the `rpm` model permitted more than one package to 'own' a file. Often we get into the situation where we need to deliver a particular

file, which may also be delivered by another, optional, package.

- Ability to install missing dependencies. In addition to having `rpm` list all the required dependencies, it would be useful to be able to have it install the missing ones from a pool of available package files.

4.5 Future Enhancements to Distribution Enabling

The current implementation is pretty closely tied to `rpm`. However, there are a number of Linux distributions which don't use `rpm` to manage their packages.

The classic example of this is Debian, which uses its own (and completely different) package manager. The best way to support this is probably to use the standard distribution enabling setup and provide a debian package which has all the debian dependencies as package requirements. However, as part of the installation of this package, we would also install a skeleton `rpm` system which provides the distribution enabling virtual package. Then installation of the distribution independent application packages via `rpm` would be able to proceed properly, even on the Debian platform.

4.6 Conclusion

Using the Distribution Enabling scheme, the costs associated with releasing software on multiple Linux distributions can be managed so as to make them much less than the costs of supporting a corresponding set of releases for different flavours of UNIX. Unfortunately, this is still higher than the cost of targeting a single OS. However, we hope that this scheme will persuade the software vendors already considering a Linux release that the costs may not be as great as they initially anticipated.

4.7 Postscript

Although the product itself is proprietary, the distribution enabling model described herein is primarily script based and readers wishing to explore details of the implementation more fully are invited to download a demonstration copy of LifeKeeper from `http://www.steeleye.com/`.

Notes

[1] Linux is a trademark of Linus Torvalds. LifeKeeper is a trademark of SteelEye Technology, Inc. All other trademarks are the property of their respective owners.

[2] The shared memory segment must be fixed at the same virtual address for all LifeKeeper processes (since it contains internal pointer references) but that address may be varied globally. Therefore, we make all processes read the location of the shared memory segment from a file before start up and alter the value (to `0x5fff000`) for Caldera.

[3] This is primarily accepted wisdom, perhaps the best recent example is from the US DOJ vs Microsoft court case[4] where Microsoft argues …*The uniformity of Windows (in contrast to the many different flavors of the UNIX operating system) is one of the primary benefits of the operating system, leading to the availability of a vast range of compatible hardware and software products*

References

[1] SteelEye Technology, Inc. *LifeKeeper for Linux* The steeleye website is currently being redone—will provide the URL when it is finally fixed

[2] Jeremy White, Marcus Meissner, Alexandre Julliard *The Wine Project, An Open Source Implementation of the Windows API* Comdex 99 `http://www.winehq.org/Talks/comdex99/img0.htm`

[3] `http://www.dosemu.org/`

[4] Microsoft Corporation, Inc. *Microsoft's Supplemental Memorandum in Support of MSJ.* Re: DOJ v. Microsoft, Case No. 98-1232, 1233. Date: September 14, 1998. `http://www.techlawjournal.com/courts/dojvmsft2/80914msft.htm`

[5] Edward C. Bailey *Maximum RPM: Taking the Red Hat Package Manager to the Limit* 17 February 1997

[6] The Linux Standard Base `http://www.linuxbase.org/`