Scalable software distribution systems

TDT4750 Computer systems administration, depth study

Tollef Fog Heen (tfheen@idi.ntnu.no)

24th November 2004

Acknowledgements

I'd like to thank my counselor Anders Christensen for a lot of help and guidance with the project. Without him, I would have gotten lost. I also wish to thank my girlfriend, Karianne, without whose support I wouldn't have finished on time.

Trondheim, November 2004
Tollef Fog Heen

Contents

Dissertation text

The text should describe scalability problems in the current software distribution systems for Unix-like operation system. The goal is to suggest ways they can be improved.

Chapter 1

Introduction

Computers are becoming more and more complex, more and more interconnected and have an ever-increasing amount of software installed and configured. In this paper, I will research one way to handle the problem of managing the installed software and growing number of hosts.

This text requires a good knowledge of UNIX-based operating systems and concepts. The text has been kept short while still being informative. The intended audience is developers of packaging systems and others with similar interests. The choice of English rather than Norwegian is due to the small size of the community and based on a goal that this article will help software distribution systems handle scalability problems better.

1.1 Delimitations

Making a generalized solution for all systems under the sun is, if possible, extremely hard. Also, rather than making a "one-size-fits-none" kind of solution, I will concentrate on a solution for UNIX and UNIX-like operating systems (like Linux). Those are similar enough that that they will be treated as variants rather than different operating systems. Other paradigms like Microsoft Windows and VMS will need other kinds of solutions, which will not be researched here.

Chapter 2

Problem definition

Most organizations, be it your family or your workplace, start small. They have few people who require little in the way of explicit communication and procedures. Today, most organizations use computers extensively, and those need maintainence. As long as you have a small number of hosts, maintaining each as a single entity, troubleshooting, debugging, installing and removing software on each machine by itself, is not too much work.

As the organization grows, so will the number of computers, and at some point, maintaining all the diverging hosts will become a nightmare, the users will be unhappy (since the hosts are broken in various ways: printing works on one, the word processor on another, and so on). Or, huge amounts of resources are spent on maintaining the hosts.

2.1 Packaging systems

What is a package system? It is a tool which helps the user[1] to:

1. Keep system consistent across system boundaries such as hosts or architectures.
2. Install and upgrade software.
3. Distribute software and configurations in a consistent, predictable manner.
4. Keep track of installed software.
5. Make sure the installed software actually works.

All of this is about handling complexity and scalability, and this is the hard problem at hand. Distributing a piece of software (as a tarball), and

[1]The user might not always be the end-user, it can also be the local system administrator or a software distributor.

unpacking it somewhere is easy. However, making it interact correctly with other software and being able to remove or upgrade it later, without fear of breaking other software makes the problem a lot harder.

Some of the dimensions in which we want scalability are:

1. Autonomous sites
2. Release levels
3. Regional differences (time zones, default language)
4. Package choices (conflicting packages, dependencies)

Chapter 3

Philosophy

When making something, one always faces choices on how to do it, weighing the implications for and against the different design choices.

3.1 Good versus bad design

One can argue whether good and bad is relevant when it comes to design. To a certain degree and in a certain context, it is. A quick hacked-together solution is not good design if you want something that lasts and is easy to for others to extend. As a software distribution system will be used by and worked on by many different people. Those will of course wary in both roles (installing and maintaining software, building packages, changing the distribution system itself) as well as skill. This seems obvious, but when one looks at the different systems out there, it is clear that many people wrote code first and thought about the problem later, rather than the other way around.

It is also important that the different design choices support each other. If not, the user will be confused, the software will be slow and full of errors.

3.2 Who is the user

Users are different and have different goals, usually dictated by their role. A system administrator is usually not interested in working on the packaging system itself. He is more interested in making or installing pre-made packages. To a packager, installing pre-made packages is of little interest, he will be interested in making good packages with as little hassle as possible. A new user wants the system to be easy to get into while an experienced user will value work-saving features. Note that there isn't a necessary conflict between those goals.

3.2.1 Operations

The number of operations a software distribution tool must support is surprisingly small. Those operations aren't atomic, but they are operations needed in a full-blown software distribution system. The different options and operations listed here are mostly intended as thought-starters rather than being a comprehensive list of all the choices available to the designer of a software distribution system.

Figure 3.1: The lifecycle of a package

1. *Production* Without the ability to produce packages, the rest of the system is unusable. When deciding what the production step should look like, there are some options available:

   - *Pure (pristine) source or patched source?* The trend is certainly towards having pristine tarballs from the distribution source, as this makes it easy to check the integrity of those using tools such as md5sum or sha1sum and using separate patches on top of this.
   - *Form of build scripts.* Build scripts can be abstract, saying "put this file there". Abstract build scripts don't have commands in them, their function is decided by their name or position in the build tree. Non-abstract, or real scripts can also be used. Examples are shell scripts or makefiles. Those list the commands to be executed.

2. *Distribution* can be in source or binary forms, it can be over a network connection or on CD or other physical media. The former is decided by the software distribution system, the latter is usually not. There are wide support for both source- and binary based distributions and they give different abilities:

   - Source-based distribution means you can turn off support you don't need, which can decrease the size needed by the system. In most modern systems, this will not make any real difference, but some people think it's very much worth it.

- Binary distribution has the advantage of much easier reproducability (if you install the same set of binary packages, you will always end up with the same setup.)

Note that source-based or binary-based is orthagonal to free software/open source, a source based distribution can ship pre-compiled objects which are linked together depending on the configuration. (An example of this is when booting the Solaris kernel, it relinks itself.)

3. *Initial installation* of packages. One of the most important choices when it comes to how to install packages, is the choice of complexity. The simplest solution is to have a cpio or tarball which is just extracted onto the system. It becomes a bit more complex if you add pre- and post-installation and -removal scripts. Even more complex if you add classes of files, so the system administrator can say "I don't want documentation installed", or "Those architecture-independent files can be accessed via NFS from that server".

4. *Maintainence* including querying capabilities and upgrading of configuration files. In some cases, another setting is needed or the syntax has changed. In those cases, you want to preserve the local modification while including the upgrade. In some cases, it's not possible to automatically upgrade and the administrator must be told of the problem so he can resolve it by hand.

5. *Configuration of the software distribution system* All systems have a number of built-in default values, here called implicit configuration. In addition, one usually has some kind of configuration file or tree where those default values are overridden. Striking a balance on what should be configurable and what shouldn't be is important, both for speed, stability and user friendliness.

   In addition to the amount of configuration is the form of configuration. It will be explored a little bit more in the next section, as part of the user interface. It is important to note that there is a grey area between configuration and customization of the distribution system, if the configuration language has the needed expressive power.

   Too much implicit configuration can be a problem, since the administrator then doesn't understand how (or has to learn how) his configuration interoperates with the default configuration. On the other side, the administrator is usually interested in turning the minimum number of knobs.

3.2.2 User experience paradigms

Designing user interfaces is a very hard task. An interface that is simple and easy-to-use for one person might be useless for another person, since the latter will not have the same connotations connected to the metaphors in use. This is especially true for configuration languages. Some can be in the form of production rules (like Makefiles). This is a very beautiful and very simple concept, but it can take a while to get used to it. Other languages will be a simple key = value form.

Many packages will need to have support scripts of some sort, for deciding on values to put into configuration files, call other utilities to notify them of the installed package and integrate the package properly into the rest of the system. Most packagers aren't really programmers, they just write those support scripts as a by-product of the packaging. This means they need a fairly simple way of programming. The simplest is the so-called "standard execution point" or procedural programming. For bigger and more complex packages, one needs support for structured programming.

3.3 Operational paradigms

There are several ways to go about in order to reach a certain goal. Push isn't better than pull, or the other way around. They are different and give you different abilities, strengths, weaknesses, and options on how to design the rest of the system. The goal is to choose a good set whose sum is as close as possible to where one wants to end.

Some important paradigms are:

1. *Push/pull*

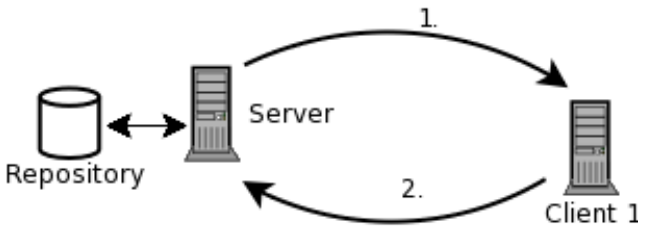

Figure 3.2: Pushing

Push means a central server knows the current state of each of the clients and pushes out the needed information to the clients. One gets fewer coordination points and just one machine to do backups of, but at the same time, the system becomes vulnerable with a single point of failure.

The server not only decides what changes should be pushed, but also at what time they should be pushed, which could be an inconvenient time for the client. In practice, the push server also needs to be able to log in as the administrative user on each of the clients, making push a possible security issue, even more so if the clients are in separate administrative domains.

Multi-level hierarchies are also harder to implement using push, as the central host will have a hard time knowing the state of all the clients due to having the state pushed through (possibly several) intermediaries.

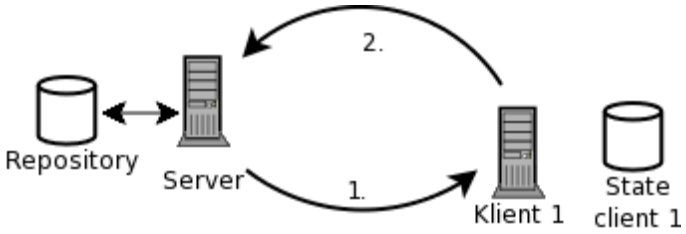

Figure 3.3: Pulling

Pull means the clients pull down whatever information they need from a server, they need a database of information to a lot lesser degree than in a push system, as the client can look up information runtime. A pull-based system is harder to get an overview of, as the state information is spread over all the hosts. It is also harder to know if a host is down or not updating properly, as it will just stop doing what it is supposed to do.

Most systems will not be one of the other, even systems like rdist, which are nominally push, will look to see if they actually need to push a newer version of the file, rather than storing all the state on the central server.

2. *what to push/pull: operations or wanted state?*

   One can transfer either a set of operations or the wanted state one wants the system to get to.

   Pushing (or pulling) a set of operations is a lot simpler, both on the client which just has to execute the operations and on the one deciding what operations to send. However, when pushing operations, a brute-force approach is often chosen, so instead of doing the minimum amount of work, a lot of runtime computer power is wasted in order to save human work.

   Pushing state to the client means more work goes into figuring out

how to get from the current state to the wanted state, but less work goes into actually getting there.

3. *autonomous units, immune systems*

   A fairly recent trend in configuration management systems, spearheaded by cfengine by Mark Burgess is the thought of the computer system as an autonomous system with an immune system composed of agents trying to ensure the system is in a wanted state. This is mostly used with the wanted-state approach listed above.

   Immune systems sounds like a very nice idea, the problem here is specifying what is the correct state and how to react to "infections". It is also a hard task for the programmer of the system.

4. *tool structure*

   Different operating systems have different styles in which most of the tools are made. Some have large tools which can do everything, some have small tools which only do one thing, but do it really well. The tools interoperate through different methods: pipes and object invocation (COM, CORBA) are among the options.

Operating system	Tool size	Interoperation method
UNIX	Small	pipes
VMS	Large	N/A[1]
Windows	Large	object invocation

5. *Tool size*

   Smaller toolsize facilitates debugging and lower memory demands at the cost of more processes and often speed.

   Bigger tools can be better integrated, will often be faster in total, due to said integration but will consume more memory, as all the functions are loaded even if one just uses a subset.

6. *Interoperation method*

   Pipes are a very simple concept, but they seem to scale amazingly well. They are flexible, since one can add another process in the middle of a pipeline. Pipes are usually combined with small tools, but they can work fine with large tools as well. Among the disadvantages, we have brittleness (it's hard to detect if one of the steps in the pipeline fails) and limited expressability as flow control is fairly limited.

   Object invocation seems better for gluing applications together and embedding applications in other applications rather than having applications work together. It makes complex interactions between different modules fairly painless at the cost of higher general complexity.

[1] In VMS, a switch would be added, or one would use temporary files.

3.4 Programming interfaces

Since no tool is complete by itself, most tools will be used together with other tools. It will be embedded and called from other tools and extended with new functionality. Some extensions are best done by plugins or extending the program, while others are best done wrapping the software in another program. This mean the different ways should all be supported, not just one.

Some of the programming interfaces available to extend a system are:

1. *Plugins*

   Support for plugins means custom code can decide how to handle items such as file types, like the example on "don't install documentation" can be changed into "put all documentation into this other (web-accessible) location".

   It can also be used for adding support for other distribution forms, other formats and so on. Generally, plugins are used for extending the abilities of a program rather than changing how the internal work flows.

2. *Hooks*

   Hooks are scripts, programs or library functions that are activated at predefined points during the execution of another function or program.

   Examples of this would be a package wanting to know when another package is installed, since it would then need to reconfigure itself (imagine an emacs add-on wanting notification about a new version of emacs being installed) or SELinux (Security Enhanced Linux) wanting to adjust the security contexts of newly installed files.

   Also, the ability to run a program once when a software package has completed installing would be a hook. An example would be regenerating the menu files for the installed window managers when a new program is installed.

3. *APIs*

   If the software distribution system is used as a part of another larger system, it's most useful to be able to drive the distribution system programatically with rich error reporting functions. It also means the system has a set of well-defined operations which are exposed to the rest of the world and the inner workings properly hidden inside the application.

3.5 Summary

As we can see, there are quite a number of design choices which affect the basics of the system quite heavily. If those are ill-chosen with regards to the wanted behavior of the system, the system will never be nice, smooth and well-working, but make the system act inconsistenly and surprisingly with missing functionality.

It is also important to note that some of the choices goes well with other choices while other are bad fits, so the set of options listed here is neither complete nor completely orthogonal.

As it goes for tools, "one size fits all" doesn't fit here either. Perfect tools are just a dream, but a combination of different tools fitting well together comes fairly close. Even if somebody made a perfect tool today, it would not fit the next time somebody came up with another requirement. Extensibility is therefore a keyword, and it needs to be built in from the start.

Chapter 4

Existing solutions

4.1 Introduction

Most tools available today fall into one of two classes, distribution-oriented or host/customization-oriented. Examples of the former are RPM and dpkg, of the latter, cfengine and rdist. Most distribution-oriented tools work with some sort of a database (binary or text-based) where they record all the software they have installed. They have no concept of software outside what they have installed themselves, which means locally compiled software can never satisfy needed dependencies for distribution-distributed software. The distribution-oriented tools are usually close-ended. Close-ended implies the list of operations which can be formed is closed.

Host/customization-oriented tools, on the other hand usually work with what's on the file system. Most of them are file-oriented and many have no concept of a "package", and if they do, it's mostly as a collection of files, not as a full-blown package with dependencies, pre- and post-installation and -removal scripts. In constrast to distribution-oriented tools, those tools are usually open-ended. Even though pre- and post-installation scripts may not be in the list of features supported by the tool, they can be provided by the packager.

4.2 Processes vs tools

This chapter is about existing tools, but even though good tools are needed, they are not enough. In order to have a good, well-scaling system, a lot of processes are needed. As shown in Figure 3.1, one needs to gather the software to be distributed, it needs to be customized, coherently packaged according to a policy and put in a repository. A subdistributor can then pick packages from different repositories, customize those and put them in their own repository. At any point in the tree, end-users can use the repository there as their repository. An infrastructure and processes for automatic

building of packages are needed if one has more than a very small number of architectures. It is therefore important not to focus too much on tools alone, but also make sure one has the needed infrastructure to fully utilize the tools.

4.3 Existing tools

There are a lot of existing tools; I will describe some of the most used ones.

4.3.1 dpkg

Dpkg is Debian's package manager and by itself, it is fairly low-level and is therefore normally used together with at least APT[1] and a frontend, like Synaptic, dselect, aptitude or apt-get.

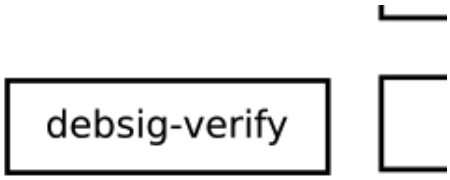

Figure 4.1: Debian's stack of user tools and how they work together

Debian packages are .ar archives (as usually used by static libraries (.a files)) with three components inside: a file containing the version number, a gzipped tarball with the control information (package name, dependencies, pre/post install/removal scripts) and finally a gzipped tarball with the files contained in the package. This makes sure the package can be handled using standard tools in case dpkg is unavailable or doesn't work (due to internal errors or missing libraries).

Dpkg consists of a lot of small programs. The end-user will usually just use "dpkg", which will then call out to dpkg-deb, dpkg-query and so on. Some of those, like dpkg-query are useful to have available for the advanced user, but won't be needed in most cases.

For developers, dpkg has a large selection of specialized utilities to help when building packages, most of them concentrate around building the package correctly and generating metadata used by the rest of the Debian infrastructure, but not by dpkg itself.

As the dpkg utilities are fairly low-level, there exists a number of helper utilities to help abstract away the details and help manage patches to the

[1] advanced packaging tool, developed for Debian/dpkg and later ported to RPM

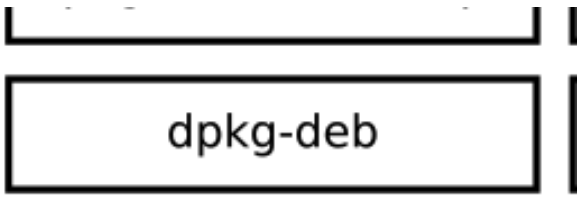

Figure 4.2: Debian's developer tools and how the work together

source code. Those are especially useful for larger packages, and they facilitates adherence to Debian's policy.

4.3.1.1 Characteristics

1. UNIX *philosophy.* As shown in the diagrams, dpkg consists of multiple smaller programs. The programs are cooperating through standard mechanisms such as pipes.

2. *Standard file formats (.ar, .gz, .tar).* This expands on the previous point. Rather than reinventing the wheel and making a new file format, standard, well-known file formats are used. This makes it possible to use standard tools as well, if that is wanted or needed.

3. *Text/RFC822-based control files.* Meta-information, such as package name, version, maintainer is stored in the format specified by RFC822 [1]

4. *Flexible with regards to source package layout* The only requirement is that the file `debian/rules` is a makefile. Many different tools are available to help build the package in a smooth fashion.

5. *Binary package distribution* The packages distributed are binary while a source package is usually also generated. The source package is used by autobuilders for other architectures.

6. *Modular base-line design.* Functionality considered standard in other packages are add-ons in dpkg, or handled at other levels, like fetching and signing of packages.

4.3.2 RPM

RPM originally meant "RedHat Package Manager", but was renamed to "RPM Package Manager" in the tradition of recursive acronyms [2]. It is used

[2]Recursive acronyms are quite common in the free software world, with examples such as GNU (GNU's not UNIX), PHP (PHP: Hypertext Preprocessor) and there's even an example

by a large number of Linux distributions: RedHat/Fedora, SuSE, Mandrake, Trustix and more. RPM by itself is a low-level tool, somewhat like dpkg.

It does not have a concept of a repository where to get dependencies from, even though it has built-in support for fetching packages via FTP and HTTP. This, combined with the fact that there wasn't any frontends with a repository concept made the built-in support for dependencies much less useful. The user had to fetch the dependencies himself as RPM told him about them. This hand-tracking of dependencies was coined "RPM hell". Later, it has gotten support tools like urpmi, APT, yum and up2date which handles the downloading and installation for the user.

RPM archives are segmented into two parts: one is a binary object with the control information. The second part is a cpio archive with the package's files. The file format means RPM and other rpmlib-using tools are the only tools able to handle RPM files.

The rpm command line tool is just one big program with its behaviour adjusted by command line flags. It is also used for building packages, specified by a "spec file". The spec file describes how the package should be unpacked, patched, built and installed. It also specifies version numbers, where to get the upstream source and any patches to apply. It also includes any scripts related to the installation and removal of the package. The spec file can also include macros, and a set of standard macros comes with RPM.

4.3.2.1 Characteristics

1. VMS-*alike philosophy.* See 4. rpm is one command line binary with its behavior dependent on switches. A more "UNIX-like" way is smaller programs talking to each other through pipes, with each small program tuned to the job.

2. *Efficient, non-standard file formats.* The format of the binary control information field is not in any standard format. cpio, while a standard format, is less common than tarballs. However, this gives possibilites for optimizations and relaxes requirements on not breaking compatibility with standard tools.

3. *Pristine source with stringent source layout.* The source package has a very well-specified layout. Spec files have a well-defined syntax, which makes it easy for people who know RPM, but not the specific package to make changes.

4. *Binary package distribution.* While RPM has support for source packages (called SRPMs), packages distributed to end computers contain the binaries to install.

---

of two mutually recursive acronyms: the HURD, which stands for "HIRD of UNIX-replacing Daemons", with HIRD standing for "HURD of Interfaces Representing Depth"

5. *Built-in auxilary utilities.* RPM has many utilities built-in, which are external in other package managers. Support for FTP and HTTP downloads, signed packages and verification of installed packages are examples.

6. *Good querying support.* RPM stores a fair amount of meta-data about the installed packages, like installation time, on what host the package was built. All this information is available for querying, with a user-specified format.

7. *Librarified.* rpmlib is a library other application than the rpm command line tool can link to in order to provide package management services, including query support. This makes it relativetly easy to build frontends with a richer set of semantics than what pipes and return codes make possible.

4.3.3 Freebsd Ports

FreeBSD, when installed, provides a small, generic "base system". It does not include much more than what's needed for bootstrapping applications. If you want to run a web server, have another shell and so on, you have to install a port, from "the ports collection". A similar system is used by NetBSD (pkgsrc) and OpenBSD. The gentoo Portage system is also inspired by ports.

Ports is a tree of (at this writing) almost 12000 applications. The on-disk format is a tree, organized first by section ("cad", "graphics", "web" and so on) then by application name.

The application directory contains a makefile, some meta-data and zero or more patches. Each makefile is really a small application which creates packages that are handled by the pkg* commands. Using a configuration file, the user building the package ("running the application") can decide whether to use or disable certain features, like IPv6 support.

As the packages are tarballs with meta-data files, they are very easily manipulated using standard UNIX tools. The meta-data files are all text files and even though they aren't according to RFC822 or other standard format, they are easy to read and understand.

For development, Ports provide a framework which all ports use. They don't necessarily have to, but in practice everybody does. Development consists of writing a makefile to download and build the sources. In many cases, some patches are needed as well, so development consists of writing those as well.

4.3.3.1 Characteristics

1. UNIX *philosophy.* Ports leverage existing tools heavily. Some tools

used are make, sed, wget, md5sum and gcc.

2. *Standard file formats.* As standard tools are used for building packages, using non-standard file formats would be unnecessary work.

3. *Text based control files.* UNIX has a tradition of having files as text files, unless it is absolutely necessary to have them as binaries. With the chosen tools, text-based control files comes naturally.

4. *Pristine source with fairly stringent source layout.* Like RPM, Ports has a pristine source-policy, with external patch files. This helps not losing patches when updating to a newer upstream version.

5. *Source based packages.* Most ports are distributed as source packages and built on the machine where they are to be installed, but there exists binary port repositories as well, and apart from customs, there's nothing stopping one from installing binary ports on machines.

4.4 Other tools

We have a plethora of other important tools, some of which could have been examined as the ones above have been. They will only be briefly described with an overview and an explanation of why they are not being examined further.

4.4.1 cfengine

From the cfengine website[2]:

> Cfengine, or the configuration engine is an autonomous agent and a middle to high level policy language for building expert systems which administrate and configure large computer networks. Cfengine uses the idea of classes and a primitive intelligence to define and automate the configuration and maintenance of system state, for small to huge configurations. Cfengine is designed to be a part of a computer immune system, and can be thought of as a gaming agent. It is ideal for cluster management and has been adopted for use all over the world in small and huge organizations alike.

Cfengine is not being examined further because it is not a closed-ended system. It is more of a language and an open-ended system for building a software distribution system, so comparing cfengine doesn't make much sense. Cfengine would be a strong candidate for making a new software distribution system in order not to start completely from scratch.

4.4.2 rdist

rdist[3] is a program which uses push file distribution and with rudimentary support for post-installation scripts. It is an open-ended system without a package concept, though one can be emulated. One way is grouping files per-package in different distfiles[3] and distributing a different set depending on what packages are to be installed on a host.

4.4.3 pkgadd (Solaris) / SAM (HP-UX) / SMIT (AIX)

Those are all tools for commercial UNIXes and are generally not too well documented in what kind of package formats they use and how to build packages for them. They do not come with source, which means examining them with the intent of extending them would be hard. They are also tied to their "native" UNIX and therefore has limited interest given that we want to scale across platforms.

4.4.4 stash

stash[4] is mainly used for managing a small number of packages in a person's home directory rather than a full system installation. It tries to autodetect the type of package (autoconf configure script, IMakefile, perl's Makefile.PL, python's setup.py and so on) but doesn't seem to have any support for automatic building of packages through some "receipe".

It is not being examined further because it is geared towards managing programs in a user's home directory rather than system-wide software management.

4.4.5 Depot

Depot[5] organizes related but different software packages into categories known as "collections". It doesn't have a dependency concept, so a collection has to be self-sustained, or the user will have to track dependencies himself. Depot is more of a method to organize software into manageable groups and doesn't have a mechanism to distribute packages. This makes it out-of-scope for the purposes of this paper.

4.4.6 Variations/forks of other tools

Those tools listed here are either plain forks of software listed above, or they are variations on the theme, and are therefore not examined in further detail.

[3] The files that describes what files are to be distributed and where to.

4.4.6.1 OpenPKG

OpenPKG uses a forked RPM, and concentrates a lot more on the distribution side and is more of a source distribution tree with source RPMs than a package tool.

4.4.6.2 Fink

Fink is a patched dpkg and APT for Mac OS X, with a separate package repository from Debian.

4.4.6.3 Portage

Portage is the BSD ports taken to a full system level. It is used by the Gentoo Linux distribution.

4.4.6.4 gnu Stow

Stow is a scaled-down version of depot, not requiring a database and not having a collection concept. There are a bunch of forks/reimplementations of it: StowES and XStow are two.

4.4.6.5 Store

Store was developed at what is now the Norwegian University of Science and Technology. It is inspired by depot, but the codebase is fully separate. It has not been developed for quite some time and is slowly being phased out.

4.5 Summary

The following is a summary of key points on the three package systems examined in detail.

Requirement	dpkg	RPM	BSD Ports
File formats	simple, standard	custom, binary	text-based, standard
Philosophy			
- push/pull	pull	pull	pull
- operations/state	both[4]	both[4]	both[4]
- tool structure	small tools	large, do-it-all tool	small tools

[4]Files are state, pre- and postinstallation/removal scripts are operations

Requirement	dpkg	RPM	BSD Ports
Operations support			
- pristine source	supported, but not compulsory	compulsory	compulsory
- patch system	none native, but externally supported	native	native
- recursive repository support	no[5]	no native repository support	no
- upgrade support[6]	yes	yes	yes[7]
- configuration file handling	postinst or asking administrator	rename new or old file	none[8]
- installation	yes	yes	yes
- acquistion	not natively[9]	yes	yes
- querying	yes	yes	yes
- verification	not natively	yes	no
- signed packages	not natively	yes	no
- multiple architecture support	no	yes	N/A
Programming interfaces			
- plugins	no	no	no
- hooks	no	yes[10]	no
- APIs	yes, command line	yes, library	yes, command line

RPM has good native support for a lot of extensions such as good querying support and support for signed packages. Dpkg does not have native support, but it's supported through external tools. RPM is librarified, which is a huge advantage if one wants to use it as a part of a bigger system. There has been some discussion about librarifying dpkg, but work has not started on it.

Scalability can be supported through recursive or hierarchical repositories where the computers do not have a custom configuration themselves, but get their configuration from the repository. Given a low enough cost for branching off a repository, having one repository per configuration is doable and efficient. Some repositories might be used by as little as a single

[5] repositories supported with APT, externally

[6] support for upgrading packages, separate from removal and reinstalling

[7] upgrading ports can easily break package dependencies, especially for library packages

[8] due to policy choices; the admin is told what to do and has to do the changes himself

[9] APT or dselect acquires packages

[10] called triggers

host (an example being a single computer with a scanner or DVD writer in a computer lab), but most repositories will be used by a larger number of hosts, such as a complete lab.

A design like this blurs the difference between multiple different configurations inside an organization and branching off another organization's repository for one's own internal one. Handling different release levels can be done with different repositories which can be partially overlapping (where the version in multiple repositories is the same) where only the file describing the repository shows a different subset. The design is inspired by Tom Lord's Arch[6].

None of the tools has native support for recursive repositories of any kind, but both dpkg and RPM can be wrapped by APT. With external tools, emulating recursive mirrors is possible to do.

Chapter 5

Composition / Evaluation

This chapter describes how to solve the problems as described in the problem definition. There are two "extreme" alternatives: write something from scratch or build on one of the existing solutions. Starting from scratch gives one a lot of possibilities, but it also means more work and a higher risk of errors. Starting from a know code base means one can start working on the needed features right away, but means one is tied to the current design, which might get in the way of wanted result.

Dpkg has been in development for over ten years, RPM a bit shorter, but it's also fairly old. If one were to develop a new tool, it would not be feasible to have equivalent functionality in significantly (one magnitude or more) less time. In addition, the existing tools have been used and thereby proven by a lot of users over time. A new tool would need time to be proven in the same way. Because of this, investigating the existing solutions rather than developing something from scratch is the most interesting approach.

5.1 Evaluation

Following is an evaluation of the different tools with regard to the requirements pointed out in the problem specification.

5.1.1 RPM

RPM doesn't support any repositories by itself, but it is well-supported by way of APT, yum or a number of similar tools. None of those support recursive repositories, but as described, that can be emulated. The repository tool would also be responsible for handling different release levels.

5.1.2 dpkg

Dpkg is well-integrated with existing tools such as APT. APT handles different releases and mixing of those easily. Dpkg also has rich semantics when

it comes to dependencies and conflicts. This makes it a good candidate for extending, even though it lacks in the programming interfaces department.

5.1.3 BSD Ports

BSD Ports lacks several important features, and while it could be a good basis to build on, one would use a fair amount of resources to build an infrastructure which matches RPM's or dpkg's. It has a strength that it is cross-platform due to only depending on standardized tools like make.

5.2 Ranking

None of the systems support repositories by themselves, but both RPM and dpkg does it through APT (or similar tools). RPM is already librarified and supports a fair amount of extra functionality internally, which need external tools for dpkg. Dpkg is more UNIX-like with small tools and interfaces with them. This means further development on dpkg is more likely to succeed, and there is developer support already for some of the needed changes.

This gives us the following ranking:

1. dpkg
2. RPM
3. BSD Ports

5.3 Needed changes

For dpkg to satisfy the requirements laid out in 2, the following changes, to dpkg itself, or related components, the following changes need to be implemented. Note that not all the requirements in the problem definition are touched, as dpkg is already a working software distribution system and therefore has several of the requirements fulfilled already.

- *Keep system consistent across system boundaries such as hosts or architectures.* Currently, dpkg has no support for this at all. It is realized through combinations of tools such as pkgsync[7] with rdist[3] or cfengine[2]. Those tools would need to be extended, or procedures such as putting the pkgsync configuration in a file which is installed by pkgsync itself would have to be developed. This fulfills 1 in 2.1.
- *Recursive repository support* will make managing autonomous sites, regional differences a lot easier. This would be handled by APT, not dpkg itself. A way to do it would be to have "meta-repositories" which just pointed to other repositories for most of the files (in order to save

disk space on the repository). This solves the problems surrounding 2autonomous sites" and "regional differences (time zones, default language)" in 2.1.

- *Support for multiple architectures.* Dpkg would have to be extended to know about what architectures the installation supports, either natively or through emulation. The syntax of the control file fields in dpkg would also have to be extended, as they are currently unable to differ between architecture-dependent and architecture-independent dependencies. Some proposals on how to solve this has already been made, see [8] [9].

  While this isn't part of the problem definition, it is supported by other other packaging systems. It also solves some problems related to moving from one architecture to another.

- *plugin and hooks support.* The dpkg maintainer has already talked about where he sees dpkg moving[10], including triggers and libdpkg, mostly in terms of what features are desireable rather than how they should be implemented. This solves scalability problems related to dependencies in "Package choices" in 2.1.

Chapter 6

Further work

When investigating possible solution, one excludes a certain amount of possible solutions and there are interesting paths one does not explore.

- Explore the relation between tools and processes. Tools are useless if there are no processes and procedures that choose how they are to be used.
- Explore open-ended tools further. Open-ended tools have mostly been skipped in this article, but they are certainly worth a closer look as a basis for writing a tool from scratch.

References

[1] David H. Crocker. Rfc 822 - standard for the format of arpa internet text messages.

[2] Cfengine web site. http://www.cfengine.org/.

[3] Rdist web site. http://www.magnicomp.com/rdist/.

[4] Stash web site. http://www.wyrick.org/source/perl/stash/.

[5] Depot web site. http://andrew2.andrew.cmu.edu/depot/.

[6] Gnu arch web site. http://www.gnu.org/software/gnu-arch/.

[7] Steinar H. Gunderson. pkgsync. http://packages.debian.org/unstable/admin/pkgsync.

[8] Matt Taggart. Multiple architecture problem and proposed solution. http://www.linuxbase.org/ taggart/multiarch.html, 06 2004.

[9] Tollef Fog Heen. How to handle multiarch on x86-64 (and other platforms). http://err.no/debian/amd64-multiarch-3, 05 2004.

[10] Scott James Remnant. Where next for dpkg? http://www.netsplit.com/blog/tech/debian/dpkg/where_next_for_dpkg_.html.