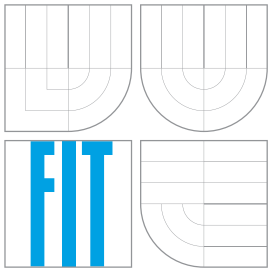# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NATIVNÍ PODPORA DEB BALÍČKŮ PRO SPACEWALK
NATIVE SUPPORT FOR DEB PACKAGES IN SPACEWALK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE   Bc. LUKÁŠ ĎURFINA
AUTHOR

VEDOUCÍ PRÁCE   Ing. RADEK KOČÍ, Ph.D.
SUPERVISOR

BRNO 2010

Abstrakt

Systém Spacewalk je určený pre správu linuxových operačných systémov používajúcich balíčkový systém RPM. Cieľom práce je rozšírenie systému Spacewalk o podporu balíčkového systému DEB, ktorý je spojený s distribúciou linuxového operačného systému Debian. Výsledok je natívna podpora spravovania systému Debian pomocou Spacewalku, čo zahŕňa jeho registráciu, distribúciu konfiguračných súborov, vzdialené spúšťanie skriptov a správu DEB balíkov.

Abstract

The system Spacewalk is a managment tool for the linux operating systems based on RPM package manager. The aim of thesis is adding support to Spacewalk for DEB package managment system, which is connected with Debian, a distribution of linux operating system. The result is native support of managing Debian system by the Spacewalk, what includes a registration of system, distribution of configuration files, remote scripts running and managment of DEB packages.

Klíčová slova

Spacewalk, balíčkové systémy, Debian, RPM, DEB

Keywords

Spacewalk, package managment systems, Debian, RPM, DEB

Citace

Lukáš Ďurfina: Native Support for DEB Packages in Spacewalk, diplomová práce, Brno, FIT VUT v Brně, 2010

Native Support for DEB Packages in Spacewalk

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Radka Kočího, Ph.D. a Mgr. Miroslava Suchého. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

....
Lukáš Ďurfina
May 13, 2010

Poděkování

Ďakujem svojmu vedúcemu práce Ing. Radkovi Kočímu, Ph.D. a môjmu konzultantovi z firmy RedHat Mgr. Miroslavovi Suchému za odbornú pomoc a podporu pri vytváraní práce.

Contents

Chapter 1

Introduction

Managing and controlling the larger number of the operating systems is not easy task. The Spacewalk is tool, which makes these activities much more simple. It stores hardware and software information about the systems, it can handle virtual systems, deploy files, run scripts or supply the software packages. The Spacewalk is free and it comes from Red Hat, what ensures its continual development.

The Spacewalk naturally supports Red Hat Enterprise Linux, CentOS and Fedora systems, in other words, the systems based on RPM packages. The aim of my thesis is to add support of the Debian operating system, based on DEB packages. This support can open a gate for supporting the all debian based systems such as Ubuntu and other derivations.

The thesis analyses the Spacewalk system, designs functional Debian client tools, and makes changes on the server side to provide the base functions for Debian systems. The aim is to add a possibility to register debian systems to Spacewalk, deploy configuration files and run scripts on these systems, also enable Spacewalk to manage DEB packages, what takes in receiving them from client and provides them to the other clients.

This paper describes the Spacewalk, the format of DEB and RPM files and differences between them. In the chapters 3 and 4 the client tools are analysed and their changes are designed there. The chapter 5 and 6 analyse the server code and introduce its implementation. Realized tests are described in the chapter 7. In the end there is discussion about the results and future work is proposed.

Chapter 2

Description of technologies

The main used technologies are Spacewalk and deb packaging system. In next paragraphs they will be introduced and important parts for this work will be described in more details.

2.1 Spacewalk

Spacewalk [15] is open-source management software for Linux systems. It is realesed under GPLv2 and is derived from Red Hat Network Satellite, which gives customers added values such as: supporting for 5 years, supported upgrade from an arbitrary previous version, direct synchronization from rhn.redhat.com, telephone support or the higher prirority of fixing the bugs reported from customers. The differences between Spacewalk and Satellite are introduces in the table 2.1.

Spacewalk provides these important services:

- systems inventory - you can review software and hardware information

- install, update or delete software packages on system

- collect and distribute custom software packages into manageable groups

- provision, system kickstarts

- manage and deploy the configuration files

- system monitoring

- manage virtual XEN and KVM guests - start, stop, configure and monitor

- efficient distributing of content across various geographical locations

At this time Fedora, RHEL and CentOS systems are officially supported by the Spacewalk. It is given by fact, that Spacewalk comes from Red Hat. However it supports RPM packages, so it is possible to use it on other rpm based distributions, it should be easygoing on OpenSuse and the same way for Mandriva, but there packages have to be compiled by the users from source.

Spacewalk provides the user web interface for whole available actions, this is a great feature, because you have an access to your systems from every place connected to a same network as spacewalk server and from any operating system. This interface is accesible through HTTPS protocol.

	Spacewalk	**Satellite**
Primary benefits	the latest technology released early and often	stable and supported
Feature selection and integration	Red Hat and developer community	Red Hat
Development model	open source	open source
Architectures	i386, x86_64	i386, x86_64, s390, s390x
Managed systems	Fedora, CentOS	Red Hat Enterprise Linux
Red Hat support options	none (community supported)	many, including 24x7 premium with unlimited incidents
Content stream	manual import	direct via Red Hat Network
Release interval	1-3 months	6-9 months
Testers	Community	Red Hat
Maintenance and updates	Community-driven	Available via Red Hat Network

Table 2.1: The differences between Spacewalk and Satellite [16]

Registered users are divided by the assigned roles. These roles point out the operations, which can be done by user. Division of the roles:

- Administrative roles

  - Spacewalk administrator
  - Organization administrator

- Roles

  - Channel administrator
  - System group administrator
  - Activation Key administator
  - Configuration administator
  - Monitoring administrator

A Spacewalk administrator is the person with the highest rights. He can create new organizations managed by the Spacewalk and also he can add or manage an organization administrators. "The organization administrator is a user role with the highest level of control over an organization's account. Members of this role can add other users, systems, and system groups to the organization as well as remove them. A organization must have at least one organization administrator. A Channel Administrator is a user role with full access to channel management capabilities. Users with this role are capable of creating channels, assigning packages to channels, cloning channels, and deleting channels."[13] The capabilities of the other administrators are the same, but there are linked with the the concrete function like monitoring or configuration.

Spacewalk consists of the several components[13], this is the listing of them and short description:

- Database - Spacewalk supports the Oracle database, but there is ongoing work on a support of PostrgeSQL. The database can be installed on the same machine as Spacewalk or on the separate machine.

- Spacewalk - core „business logic" and entry point for yum, the package manager running on client systems. The Spacewalk also includes an Apache HTTP Server (serving XML-RPC requests).

- Web interface - advanced system, system group, user, and channel management interface.

- RPM Repository - package repository for RPM packages.

- Management Tools

    - Database and file system synchronization tools
    - RPM importing tools
    - Channel maintenance tools (Web-based)
    - Errata management tools (Web-based)
    - User management tools (Web-based)
    - Client system and system grouping tools (Web-based)

When a client requests updates, the Spacewalk queries its database, authenticates the client system by the client ID, identifies the updated packages available for the client system, and sends the requested RPMs back to the client system. Depending upon the client's preferences, the package may also be installed. If the packages are installed, the client system sends an updated package profile to the Spacewalk and it is updated in the database. And finally those packages are removed from the list of outdated packages for the client.

2.1.1 An overview of the capabilities

Each Spacewalk capability contains the set of functions, for the better imagination of the Spacewalk power and value for its users we can take a closer look on the each capability. The information is taken out of [14].

Management

Spacewalk Management is based upon the concept of an organization. Each Management-level Spacewalk Administrator has the ability to establish users who have administration privileges to system groups. An Organization Administrator has overall control over each organization with the ability to add and remove systems and users. When users other than the Spacewalk Administrator log into the Spacewalk web interface, they see only the systems they have permission to administer.

With each Management subscription, you receive these functionality:

- Package Profile Comparison - compare the package set on a system with the package sets of similar systems

- Search Systems - search through systems based on a number of criteria: packages, networking information, even hardware asset tags

- System Grouping - web servers, database servers, workstations and other workload-focused systems may be grouped so that each set can be administered in common ways

- Multiple Administrators - administrators may be given rights to particular system groups, easing the burden of system management over very large organizations

- System Set Manager - it is possible to apply actions to sets of systems instead of single systems, work with members of a predefined system group, or work with an ad-hoc collection of systems. Install a single software package to each, subscribe the systems to a new channel, or apply all Errata to them with a single action

- Batch Processing - compiling a list of outdated packages for a thousand systems would take days for a dedicated sysadmin. Spacewalk Management service can do it for you in seconds

Provisioning

Like Management, Provisioning is based upon an organization. It takes this concept a step further by enabling customers with Provisioning entitlements to kickstart, reconfigure, track, and revert systems on the fly. Provisioning provides:

- Kickstarting - systems with Provisioning entitlements may be re-installed with a whole host of options established in kickstart profiles. Options include everything from the type of bootloader and time zone to packages included/excluded and IP address ranges allowed. Even GPG and SSL keys can be pre-configured.

- Client Configuration - Spacewalk users may use it to manage the configuration files on Provisioning-entitled systems. Users can upload files to custom configurations channels on the Spacewalk, verify local configuration files against those stored on the Spacewalk, and deploy files from the Spacewalk.

- Snapshot Rollbacks - users have the ability to revert the package profile and settings of systems. Spacewalk users can also roll back local configurations files. This is possible because snapshots are captured whenever an action takes place on a system. These snapshots identify groups, channels, packages, and configuration files.

- Custom System Information - users may identify any type of information they choose about their registered systems. This differs from System Profile information, which is generated automatically, and the Notes, which are unrestricted, in that the Custom System Information allows you to develop specific keys of your choosing and assign searchable values for that key to each Provisioning-entitled system. For instance, this feature allows you to identify the cubicle in which each system is located and search through all registered systems according to their cubicle.

Monitoring

Monitoring allows an organization to install probes that can immediately detect failures and identify performance degradation before it becomes critical. Used properly, the Monitoring entitlement can provide insight into the applications, services, and devices on each system. Monitoring provides:

- Probes - dozens of probes can be run against each system. These range from simple ping checks to custom remote programs designed to return valuable data.

- Notification - alerts can be sent to email and pager addresses with contact methods identified by you when a probe changes state. Each probe notification can be sent to a different method, or address.

- Central Status - the results of all probes are summarized in a single Probe Status page, with the systems affected broken down by state.

- Reporting - by selecting a probe and identifying the particular metric and a range of time, you can generate graphs and event logs depicting precisely how the probe has performed. This can be instrumental in predicting and preventing costly system failures.

- Probe Suites - groups of probes may be assigned to a system or set of systems at once rather than individually. This allows administrators to be certain that similar systems are monitored in the same way and saves time configuring individual probes.

- Notification Filters - probe notifications may be redirected to another recipient, halted, or sent to an additional recipient for a specified time based on probe criteria, notification method, scout or organization.

2.1.2 Spacewalk architecture

Spacewalk is implemented in the several languages and has a classic three-tier architecture [10]. The presentation tier consists of both a web UI, command line clients, and XML-RPC clients (which can in turn be command line or even full blown web applications). Behind the presentation tier lies the logic tier, which in Spacewalk is spread across the four languages: Java, perl, python, and PL/SQL. In reality, there is only a small bit of code that overlaps between the languages, each language is usually used for separated purposes. Finally, the last tier, the data tier, is backed by an Oracle database and work for PostgreSql support has started. Simple diagram of Spacewalk architecture is shown on diagram 2.1.

Web UI

The web UI consisted entirely of perl running through an Apache web server from the first versions of Spacewalk. Several years ago, a Java migration was started and most of the commonly used features of the application have been migrated. In general, the perl pages are only modified to fix bugs or to support the new features in the Java side. Any new web UI development is being done in Java.

Frontend API

One of the most sought-after features of Spacewalk has been its XML-RPC API. Many users want to write automated scripts to perform repetitive tasks, usually tasks that are available via the web UI. While a web UI is useful for performing a few tasks on either one or more servers, sometimes, there is no replacement for a good script. The frontend API attempts to expose as much of the web UI functionality as possible through XML-RPC. The frontend API is written completely in Java and runs in Tomcat, in conjunction with
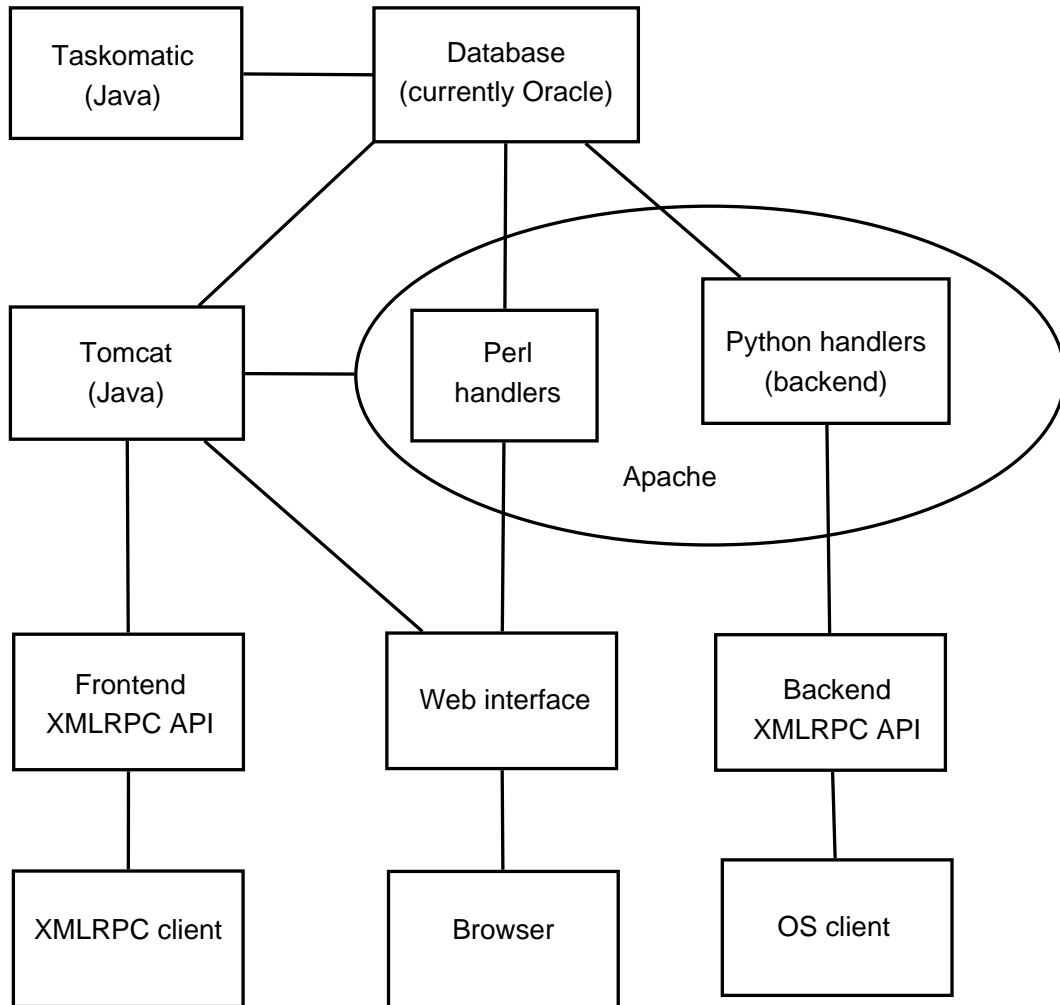
Figure 2.1: Spacewalk Architecture [11]

web UI, within the web application. Because of this, the manager layer is shared between them.

Backend

The backend provides a set of APIs that the different client utilities (rhn_register, up2date, yum) can connect to. They are solely used by the client utilities and they are not documented, but I will write some informatation about them in analysis of client tools.

Taskomatic

Taskomatic is a daemon whose job is to perform long running tasks that are scheduled to run asynchronously, such as clean up the sessions table, or send out email notifications for new errata. Taskomatic is written in Java which allows it to take advantage of the same manager layer as the rest of the Java tier. It runs as a daemon with the help from

tanukiwrapper[1], what is special java service wrapper.

Search server

One of the most important things of any application which supports large amounts of data is finding that data. Typically, an application will show pages containing lists of items which one must page through to find the item you are looking for. While Spacewalk has the page lists, it also has a search feature which allows one to find the system, package or errata quickly as opposed to paging through hundreds of items on a list. Spacewalk uses a standalone search server that run as a daemon, also with the help of tanukiwrapper. The search server is written in Java.

Java part design

Java version implements standard three tier web application model. It made the sizable code, what are showing unnecessarily on same places, but there was a speed as a very good argument to do that.

- User Interface is created by Struts[2] as a controller and JSP[3] as the view. The Struts servlet gets all web requests and then each request is directed to correct java code. Therefore there is no business logic done in this presentation layer. All data values from web forms are encapsulated by UI logic in Data Transfer Object (DTO) and forwarded to the appropriate manager class in business tier. After processing DTO are returned to UI layer and result is presented by JSP. Same way as Struts part JSP does not implement any business logic. It has only one aim, take the objects associated with the requests and format that information correctly for the UI. Next technology for UI layer is taglibs, it is used for centralization common display tasks, for example: displaying a list of data.

- Business logic tier is implemented as a set of static methods, which perform all security checks, and then make calls to the correct business objects to perform the work. This tier is slightly muddied today, because it is much heavier weight than it should be. The problem is the amount of logic that is embedded in stored procedures. Because so much of this logic is in the database, the manager classes have to make calls directly to the DB instead of utilizing business objects to do that work. Manager classes are only allowed to call directly into their related Factory class. For example, the UserManager can call the UserFactory, but it can not call the ServerFactory. If the UserManager wishes to retrieve server information, it must call through the ServerManager to do so. This ensures that no matter what is happening, security checks are performed, because the Managers take care of all security checks.

- Persistence layer is implemented by Hibernate, which allows to persist standard Java objects. There is created a standard HibernateFactory class, which implements all of persistence logic. Each major business object class has its own factory, which extends the HibernateFactory. For example, the User object has a UserFactory class, which knows how to persist users, addresses, emailAddresses and other user info.

---

[1] http://wrapper.tanukisoftware.org/
[2] http://struts.apache.org/
[3] JavaServer Pages

Python part design

Python part is divided into 2 parts:

- Client part - it is the set of client side libraries that run on the managed system. The client code periodically checks in with the server, which sends instructions to the client for executing, for example: rhn_check, rhn_register or yum-rhn-plugin. There is the only one exception, it is rhnsd, which is written in C.

- Server part - it exposes an XML-RPC interface which the clients interact with. It encapsulates a good portion of RHN Satellite's business rules and validation. In order to service up this XML-RPC interface a hand coded python object mapping of RHN's database schema was developed. Schema on 2.2 shows, how server backend is placed in architecture.
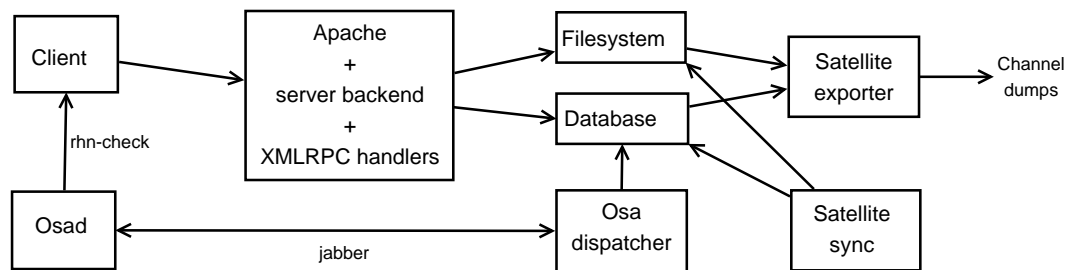
Figure 2.2: Spacewalk server backend architecture

2.1.3 Software channels

A software channel is a list of RPM packages grouped by use. Channels are used to choose packages to be installed on a system. There are two types of software channels: base channels and child channels[14]. Channels can be further broken down by their relevance to your systems, including All Channels, Popular Channels, My Channels, Shared Channels, and Retired channels.

Base channels

A base channel consists of a list of packages based on a specific architecture. For example, all of the packages for the x86 architecture make up a base channel. The list of packages for the Itanium or Alpha architecture make up a different base channel.

A system must be subscribed to one base channel only. This base channel is assigned automatically during registration based upon server system architecture or to channel, which is associated with the activation key used during the registration. In the case of public free channels, the action will succeed. In the case of private or protected base channels, this action will fail if the channel belongs to another organization or your organization does not fall into the trusted organizations.

Child channels

A child channel is a channel associated with a base channel that contains extra packages. For instance, an organization can create a child channel associated with Fedora 10 for the

x86 architecture that contains extra packages needed only for the organization, such as a custom engineering application.

A system can be subscribed to multiple child channels of its base channel. Only packages included in a system's subscribed channels can be installed or updated on that system. Therefore channel management is an important task and it has its own administrator role. This authority has the ability to create and manage organization custom channels.

2.1.4 Configuration

This section is about managing your configuration channels and files, whether they are centrally managed or limited to a single system. You must be the Configuration Administrator for having enabled the configuration actions. In addition, you must have at least one Provisioning entitlement.

„Centrally-managed files are those that are available to multiple systems; changes to a single file in a central configuration channel can affect many systems. In addition, there are local configuration channels. Each system with a Provisioning entitlement has a local configuration channel (also referred to as an override channel) and a Sandbox channel."[14]

Central configuration management allows user to deploy configuration files to multiple systems. Local configuration management allows user to specify overrides, or configuration files that are not changed by subscribing the system to a central channel. Central configuration channels must be created via the web interface. Local configuration channels are not created this way. They automatically exist for each system to which a Provisioning entitlement has been applied.

The adding files to the configuration channels can be done by three methods. Files can be uploaded, imported or created. The upload of file is made by browsing for the file on the local system. The importing of files works from other configuration channels, including any locally-managed channels. Finally, file can be created from scratch, that includes entering the ownership and permissions.

2.1.5 Monitoring

If the machine has Monitoring entitlements, that it enables to view the results of probes, which have been set to run against Monitoring-entitled systems and manage the configuration of monitoring infrastructure. The probes can have several differant statuses:

- Critical

- Warning

- Unknown

- Pending

- OK

Monitoring data and probe status information that was previously availble only through the web interface of the Spacewalk can now be exported as a CSV file. The exported data may include, but is not limited to probe status, all probes in a given state (OK, WARN, UNKNOWN, CRITICAL, PENDING) and a probe event history.

2.1.6 Similar systems as Spacewalk

The similar system as Spacewalk is Landscape, but it is aimed on Ubuntu systems. "Landscape is an easy-to-use systems management and monitoring service that enables you to manage multiple Ubuntu machines as easily as one through a simple Web-based interface. Landscape provides powerful, automated systems administration capabilities such as management, monitoring and provisioning of packages across multiple machines lowering your per-systems cost of management and administration. Landscape simplifies the complex task of monitoring and administrating multiple servers by enabling IT administrators to manage multiple machines through a single Web-based interface. At its most basic level, Landscape securely enables updates and provisioning of packages across multiple (standalone or virtual) machines. In addition, Landscape provides a host of additional monitoring, user control, process management, inventory control and support enhancement tools that can help increase your productivity immediately."[2]

Landscape is commercial product from company Canonical, so there is no possibility to use it free as Spacewak. It is distributed in two version

- Hosted edition - delivered over the internet as Software as a Service (SaaS)

- Dedicated server edition - is installed locally, giving complete control over the enviroment

Main features of Landscape are:

- System managment - allows creating groups of systems, managing the software packages, intergrating custom repositories.

- System monitoring - provides system info tool, maintaining a detailed hardware inventory or security audits.

- Cloud managment - configures, starts, stops and updates a private Ubuntu Enterprise Cloud.

2.2 Package management systems

"A package management system is a collection of tools to automate the process of installing, upgrading, configuring, and removing software packages from a computer. Distributions of Linux and other Unix-like systems typically consist of hundreds or even thousands of distinct software packages; in the former case a package management system is nice, in the latter case it is essential"[22]. Because it is almost not possible to handle all packages by human.

Packages are bundles of software and its metadata, metadata usually contains full name of software, short and long description, version and release number, vendor, checksums, list of dependencies on other packages, which is needed for correctly running of software. After installation is all that metadata stored in a local package database.

These systems are sometimes incorrectly called as installers. There are a lot of differences between package management systems and installers:

Package management system	Installer
part of operating system	each product has own installer
single installation database	handle installation info by own way
can verify and install all packages on system	works only with bundled product
single package management system vendor	multiple installer vendors
single package format	multiple installation formats

Main task of package management systems is maintaining all of the packages installed on the operating system and ensures their usability. This maintainance consists of these functions:

- installing and removing selected packages

- verifying correctness and completness of packages by checksums controls

- authenticating the origin of package by verifying digital signatures

- upgrading packages with lastest version from a software repository

- providing info about packages to users

- managing dependencies to ensure a package is installed with all packages it requires

Critical state is dependency hell, which have to be avoided by good package management system. It can be caused by requiring different versions of dynamicly linked libraries, which are widely used on linux based systems. Dependency hell is well known on windows systems as DLL hell, but it is known on linux systems too, for example RPM hell on systems with RPM package manager, but this problems are solved by intelligent wrappers as yum.

The next problem with package management systems connected with the upgrading of packages is upgrading of configuration files. Package management systems are usually based on file archiving utilities, so they used to only either owerwrite or retain configuration files, rather than applying rules to them. There are some exceptions for example kernel configuration, because error in these files can caused unusability of system. Main problem is change in format of the configuration files. For instance, if the old configuration file does not explicitly disable new options that should be disabled. Possible solutions is allowing configuration during installation or if it is desirable, install packages with the new default configuration and then owerwrite it.

2.2.1 APT

APT is an abbreviation for Advanced Packaging Tool. "APT is a free user interface that works with core libraries to handle the installation and removal of software on the Debian GNU/Linux distribution and its variants. APT simplifies the process of managing software on Unix-like computer systems by automating the retrieval, configuration and installation of software packages, either from binary files or by compiling source code."[18]

APT can work on more platforms. Originally, it was designed for Debian as dpkg front-end, but later the support for work with RPM was added via apt-rpm. APT is also available for Mac OS X, OpenSolaris, and currently it was ported to iPhone OS and there is work for porting it to other certain mobile operating systems.

There is no single apt program, because it is the set of tools and libraries. The main important programs are apt-get and apt-cache. Apt-get is command-line tool for installing, updating or deleting packages and it knows other usefull functions like cleaning or autore-moving.

2.2.2 YUM

"The Yellowdog Updater, Modified (YUM) is an open-source command-line package management utility for RPM compatible Linux operating systems and has been released under the GNU General Public License. It was developed by Seth Vidal and a group of volunteer programmers. Though yum has a command-line interface, several other tools provide graphical user interfaces to yum functionality."[23]

The predecessor of yum was YUP - Yellowdog Update, yum was created as its full rewrite. Various provided actions are called by commands in format yum command_name parameters. The most important commands are install, remove, info, list, clean or update. Yum repository is a simply directory, where can be optionally the next subdirectories, with meta information in XML format, which contains standard info about dependencies, file lists and similar. Yum can access the repository over ftp, http or a file URI.

YUM plugins

"Yum has a simple but powerful plugin architecture which allows external modules to add new features and/or modify Yum's behaviour. Yum plugins are Python modules (.py files) which are loaded when Yum starts. Plugins were created partially as a place to put functionality that was seen as either less common or undesirable for the main yum package. Functionality in plugins will generally not be moved or included in the core yum package."[12]

The architecture is similar to event-based architecture. There are a functions, called hooks, which corresponds to given slots. A slot is a point in yum's execution chain. When the point for exact slot is reached, all the hook functions, which were registered for that slot, are called. The registration of hooks is automatic, and it is made by plugin module according to the functions names. If the function name is in the format `slotname_hook`, the function is automatically registred as the hook function for that slot. All hook functions take one argument, a `conduit` instance. This is object, which provides methods and parameters for communication with the Yum. Conduit differs depending on the plugin slot.

The following slots exist and time of their execution:

- config - initialization of plugins

- postconfig - after yum config object is initialised

- init - start of yum initialization

- predownload - before start of downloading packages

- postdownload - after finishing package downloads

- prereposetup - before initialization of repository information

- postreposetup - after initialization of repository information

- exclude - after package exclusion or inclusion is processed

- preresolve - before package resolution

- postresolve - after package resolution

- pretrans - before update transaction

- posttrans - after update transaction

- close - on the yum exit

- clean - during clean up invoked by commands clean all or clean plugins

2.3 Package formats

There are different package formats for distributing software on various Linux systems. Basic kinds are binary and source packages. The next kinds are according to their architecture. In the next paragraphs the two most known package formats will be introduced.

2.3.1 DEB

"Debian packages are also used in distributions based on Debian, such as Ubuntu and others. Debian packages are standard Unix ar archives that include two gzipped, bzipped or lzmaed tar archives: one that holds the control information and another that contains the data"[20].

The file header of ar archive is as follows[19].

Field Offset from	Field Offset to	Field Name	Field Format
0	15	File name	ASCII
16	27	File modification timestamp	Decimal
28	33	Owner ID	Decimal
34	39	Group ID	Decimal
40	47	File mode	Octal
48	57	File size in bytes	Decimal
58	59	File magic	0x60 0x0A

Main tool for managing deb packages is dpkg, but it is easier to use higher level tools as apt, aptitude or synaptic, which use dpkg as the backend. These tools provide graphical interface so it is more confident for begginers, but they can solve much more things than dpkg for example installing dependencies.

Structure of the package

- debian-binary: file with deb format version number, currently it is 2.0

- data.tar.gz: or other type of archive according to used archiver. This archive contains installable files.

- control.tar.gz: contains files with meta-information:

  - control - base meta-information with package name, version, architecture, maintainer, size, dependencies and description
  - conffiles - list of the configuration files
  - md5sums - list of the md5 checksums for each file in data archive
  - preinst - script, which is run before package installation
  - postinst - script, which is run after package installation
  - prerm - script, which is run before package removing
  - postrm - script, which is run after package removing

Creating deb packages

Creating of deb packages is relatively easy and there is no need of special tools as you need for creating RPM packages, you can create package with only tools ar and tar[24], but it is comfortable to use tools, which make some tasks automatically.

Very good tool is debhelper, which is widely used. It contains various small tools like dh_make, which can help with creating a basic directory tree and required files for package creation. It takes information from console parameters and from enviroment variables. Name and email of packager should be set in enviroment variables DEBFULLNAME and DEBEMAIL. Third important information is type of license, which can be set by parameter. Other info will be asked by tool during the preparation of package. This tool is creating example files too, they are marked by extension .ex or .EX, this is usefull for begginers, else it is good to remove them[17].

Files for building package are created in directory debian, there will be listed and described:

- compat - version of debhelper used for building package, it is important for compatibility, because debhelper is still developing

- control - information with package name, version, architecture, maintainer, size, dependencies and description

- copyright - license of package, Debian is very strict on this issue

- docs - list of documentation

- dirs - list of the directories, where files from package will be stored

- changelog - list of changes between packages revisions

- rules - rules for make, which are needed for package compilation

- watch - script for tool uscan, which can control if package is up-to-date

- NEWS - important messages for users

- README.Debian - information about changes of debian package from original one for other package system

We can take a look on the very important file `control`, show the example of this file and then introduce the parameters used in it:

```
Source:  rhn-client-tools
Section:  admin
Priority:  extra
Maintainer:  Lukas Durfina <lukas.durfina@gmail.com>
Build-Depends:  python-all-dev, python-support (>= 0.8), debhelper (>= 7),
gettext, intltool, pychecker
Standards-Version:  3.7.3
Homepage:  http://rhn.redhat.com

Package:  rhn-client-tools
```

```
Architecture:  all
Essential:  no
Depends:  ${python:Depends},${shlibs:Depends}, ${misc:Depends}, gnupg, hal,
rhnlib (>= 2.1), python-dbus, python-ethtool, python-rpm
Provides:  ${python:Provides}
Description:  Red Hat Network Client Tools provides programs and libraries
    Red Hat Network Client Tools provides programs and libraries to allow
    your system to receive software updates from Red Hat Network.
```

Description of parameters [5]:

- Source - contains a name of the source package. The name consist only of lower case letters (a-z), digits (0-9), plus (+) and minus (-) signs, and periods (.). The name must be at least two characters long and must start with an alphanumeric character.

- Section - This field specifies an application area into which the package has been classified.

- Priority - Priority represents how important it is that the user have the package installed.

- Maintainer - The package maintainer's name and email address. The name should come first, then the email address inside angle brackets.

- Build-Depends - The Build-Depends field must be satisfied when any of the following targets is invoked: build, clean, binary, binary-arch, build-arch, build-indep and binary-indep.

- Standards-Version - The most recent version of the standards (the policy manual and associated texts) with which the package complies.

- Homepage - The URL of the web site for this package, preferably (when applicable) the site from which the original source can be obtained and any additional upstream documentation or information may be found. The content of this field is a simple URL without any surrounding characters.

- Package - The name of the binary package. Binary package names must follow the same syntax and restrictions as source package names.

- Architecture - Depending on context and the control file used, the Architecture field can include the following sets of values:

  - A unique single word identifying a Debian machine architecture
  - all, which indicates an architecture-independent package.
  - any, which indicates a package available for building on any architecture.
  - source, which indicates a source package.

- Essential - If set to yes then the package management system will refuse to remove the package (upgrading and replacing is still possible). The other possible value is no, which is the same as not having the field at all.

- Depends - This declares an absolute dependency. A package will not be configured unless all of the packages listed in its Depends field have been correctly configured.

- Provides - The field contains a list of provided packages, virtual packages can be listed too.

- Description - The field contains a description of the binary package, consisting of two parts, the synopsis or the short description, and the long description.

Debian repository

"A Debian repository is a set of Debian packages organized in a special directory tree which also contains a few additional files containing indexes and checksums of the packages. If user adds a repository to his `/etc/apt/sources.list` file, he can easily view and install all the packages available in it just like the packages contained in Debian.A repository can be both online and offline (for example on a CD-ROM), although the former is the more common case."[4]

A repository consists of at least one directory with some DEB packages in it, and special index files, there are two index files: `Packages.gz` for the binary packages, and `Sources.gz` for the source packages. According to the listing of the repository in sources.list, apt-get will fetch Packages.gz for the entry with the keyword deb (binary packages) and Sources.gz if the keyword is deb-src (source packages).

Packages.gz contains the name, version, size, the short and the long description, and the dependencies of each package, plus some additional information. All that information is listed (and used by) the Debian package managers such as aptitude. Sources.gz contains the name, version and the build dependencies (the packages needs for building) of each package (plus some other information). That information is used by apt-get source and similar tools. There's an optional Release file containing some informations about the repository.

A structure of the repository

```
(repository root)
|
|-conf
|
|-db
|
|-dists
| |
| |-etch
| | |-main
| | | |-binary-all
| | | |-binary-i386
| | | |-binary-...
| | | +-source
| | |-contrib
| | | |-binary-all
| | | |-binary-i386
| | | |-binary-...
```

```
| | | +-source
| | +-non-free
| |    |-binary-all
| |    |-binary-i386
| |    |-binary-...
| |    +-source
| |
| +-lenny
|    |-main
|    | |-binary-all
|    | |-binary-i386
|    | |-binary-...
|    | +-source
|    |-contrib
|    | |-binary-all
|    | |-binary-i386
|    | |-binary-...
|    | +-source
|    +-non-free
|       |-binary-all
|       |-binary-i386
|       |-binary-...
|       +-source
|
+pool
```

The `conf` directory contains a file `distributions`, which has general information about the repository content. Every item in this file has these parameters: origin, label, suite, codename, version, architectures, components and description. Directory `db` involves files with specific repository data in berkeley database format: checksums.db, content.cache.db, files.db, packages.db, references.db, relese.caches.db and version. In the `dists` directory are be listed the supported distributions of Debian. In our example we have there etch and lenny, but there also can be much more values like sid, sarge or higher level specifications like experimental, stable, unstable or testing. In the `pool` directory the packages are stored and the are separated in the subdirectory according to a first letter int their name.

There is a several tools for easier creating Debian repository[7]:

- dak (Debian Archive Kit) - packaging of the tools handling the official Debian repositories

- reprepro (formerly known as mirrorer) - local Debian package repository storing files in a directory `pool`.

- debpool - lightweight replacement for dak using a pool layout

- debarchiver - a simpler version of dak

- mini-dinstall - miniature version of dak

- apt-ftparchive - superset of dpkg-scanpackages and dpkg-scansources

- dpkg-scanpackages and dpkg-scansources - can not create Release nor Contents files

- mini-dak - partial and lightweight reimplementation of dak in shell script and with no database dependencies

- DebMarshal - maintain multiple snapshots from upstream distros, to permit staging

2.3.2 RPM

RPM was created by Red Hat for Red Hat Linux, but now it is used on many Linux distributions. RPM file format is also the baseline package format for Linux Standard Base, what is the standard managed by Linux Foundation.

RPM files has specific naming convention: name-version-release.architecture.rpm[1]. Name and version is of course name and version of packaged software, release is the number of times this version of the software has been packaged and architecture is a shorthand name describing the type of computer hardware the packaged software is meant to run on. It can be src or nosrc. The nosrc string means that the file contains only package building files, while the src string means the file contains the necessary package building files and the software's source code.

Every RPM package file can be divided into four distinct sections: lead, signature, header, archive. Package files are written to disk in network byte order. If required, RPM will automatically convert to host byte order when the package file is read.

The lead

The lead is the first part of RPM package file. It was used for storing information used internally by RPM in older versions of RPM. Now it is used for easy identification of RPM file by tools like file in unix. All the information contained in the lead has been duplicated or superseded by information contained in the header.

The lead starts with magic number, that identify the file as RPM package. Next two bytes indicate RPM file format version. After them there are next two bytes for determining binary or source package. Next two bytes store architecture, that the package was built for. If the package is source package, these bytes should be ignored. Next bytes contain name of the package and it is ended by null character. The next bytes represent the operating system for which this package was built. Translations between number and coresponding operating systems are written in file rpmrc, it is same for translations between numbers and architectures. The last bytes are identification for type of a signature.

The lead is an abandoned data structure, because it is not flexible, for example the name of package is limited by 66 chars. The solution of this problem is header section.

The signature

The signature section follows the lead in the RPM package file. It contains information that can be used to verify the integrity, and optionally, the authenticity of the part of the package file. Only the header and archive parts can be verified, because the data in the lead and header of signature are not included when the signature information is created. This is not a weakness of RPM design, because the lead is used only for easily indentification and the change of that part would cause only unsuccesfull identification of the package. The change to the signature header structure would make it impossible to verify the file's integrity and therefore it would be unusable.

The first bytes of the signature is the magic number for the start of the header structure. The next byte indicates the header structure's version. The next four bytes are reserved. After that the number of index entries is following. The next number indicates how many bytes of data are stored in the signature. Following the first 16 bytes is the index. Each of the index entriy in the header structure consists of four numbers: tag, type, offset and count.

The header

The header section contains all available information about the package. Entries such as the package's name, version, and file list, are contained in the header. Like the signature section, the header is in header structure format. Unlike the signature, which has only three possible tag types, the header has more than sixty different tags.

The easiest way to find the start of the header is to look for the second header structure by scanning for its magic number. The sixteen bytes, starting with the magic, are the header structures's header. The byte following the magic identifies this header structure format. Following the four reserved bytes, we find the count of entries stored in the header. The next four bytes tell us how many bytes of data is in the store. Next is the tag for the package name, and it finishes with data types values.

The archive

At the end after the header part is the archive, that holds the actual files that comprise the package. Before rpm version 4.7 the archive was compressed by GNU zip. The archive started with magic number for a gzipped file. The following byte was flag used by GNU zip to indicate compression method. The next byte stored strength of compression and the last byte was mark of the operating system under which the archive had been compressed. Now LZMA archive format is used. LZMA has higher compression and faster decompression, so downloading and installing of package is faster and it makes the lower data traffic on the net.

The creation of RPM package

For building RPM package the packager needs source of the program and `.spec` file. Source can be prepared as tar file or .src.rpm package. All other things are described in .spec file. There is an example of this file and explanation of its parts according to [8]:

```
Name:  python-debian
Version:  0.1.16
Release:  1
Summary:  Modules for Debian-related data formats
# debfile.py, arfile.py, debtags.py are release under GPL v3 or above
# everything else is GPLv2+
License:  GPLv2+ and GPLv3+
Group:  Development/Libraries
Source0:  http://ftp.debian.org/debian/pool/main/p/python-debian/python-debian_%{version}.tar.gz
URL: http://git.debian.org/?p=pkg-python-debian/python-debian.git
BuildRoot:  %(mktemp -ud %{_tmppath}/%{name}-%{version}-%{release}-XXXXXX)
BuildArch:  noarch
Requires:  python >= 2.4
```

```
BuildRequires:  python-devel, python-setuptools

%description
This package provides Python modules that abstract many formats of Debian
related files.  Currently handled are:
* Debtags information (debian.debtags module)
* debian/changelog (debian.changelog module)
* Packages files, pdiffs (debian.debian_support module)
* Control files of single or multiple RFC822-style paragraphs, e.g.
  debian/control, .changes, .dsc, Packages, Sources, Release, etc.
  (debian.deb822 module)
* Raw .deb and .ar files, with (read-only) access to contained
  files and meta-information

%prep
%setup -q

%build
%{__python} setup.py build

%install
rm -rf $RPM_BUILD_ROOT
%{__python} setup.py install --prefix=%{_prefix} --root=$RPM_BUILD_ROOT

%clean
rm -rf $RPM_BUILD_ROOT
%{__python} setup.py clean

%files
%defattr(-,root,root,-)
%dir %{python_sitelib}/debian
%dir %{python_sitelib}/debian_bundle
%{python_sitelib}/*.py*
%{python_sitelib}/debian/*.py*
%{python_sitelib}/debian_bundle/__init__.py*
%{python_sitelib}/python_debian*

%doc README README.changelog README.deb822 HISTORY.deb822 ACKNOWLEDGEMENTS

%changelog
* Thu Apr 22 2010 Lukas Durfina <lukas.durfina@gmail.com> 0.1.16-1
- Creation of package
```

Description of parameters:

- Name - package name

- Version - current version of software in package

- Release - mark of build revision

- Summary - short description, should be visible on one line in terminal

- License - license of application

- Group - type of application

- Source0 - path to source of packed application

- URL - URL to program in package

- BuildRoot - directory for program building

- BuildArch - aimed architecture

- Requires - required packages for correct using of program

- BuildRequires - required packages for building of package

- %description - expanded text about package

- %prep - creation of a build enviroment

- %build - steps for compiling of program

- %install - installation commands

- %clean - cleaning the build enviroment

- %files - the list of package files

- %doc - documentation files

- %changelog - log with changes

2.3.3 Differences between DEB and RPM

The short comparison of the formats DEB and RPM[3]:

Description	DEB	RPM
Security, authentication, and verification		
Signed packages (internal support for a GPG or PGP signature)	yes	yes
Checksums (for all files in the package)	yes	yes
Permissions, owners, sizes etc.	yes	yes
Usability by standard linux tools		
Recognizable by a tool file	yes	yes
Data unpackable by standard tools	yes	no [1]
Metadata accessible by standard tools	yes	no
Creatable by standard tools	yes	no
Metadata		
Name	yes	yes
Version	yes	yes
Description	yes	yes
Dependencies	yes	yes
Recommendations	yes	no
Suggestions	yes	no
Conflicts	yes	yes
Virtual packages and provides	yes	yes
Versioned dependencies and conflicts	yes	yes
Boolean package relationships [2]	yes	no
File dependencies [3]	no	yes
Copyright info	no [4]	yes
Assigning to a group	yes	yes
Priority	yes	no
Special files		
Config files	yes	yes
Documentation files	no	yes
Ghost files	no	yes
Package programs		
Binary programs allowed	yes	no
Pre-install program	yes	yes
Post-install program	yes	yes
Pre-remove program	yes	yes
Post-remove program	yes	yes
Verify program	no	yes
Triggers	yes	yes
Scalability		
No hard-coded limits	yes	yes
New metadata	yes	yes
New section	yes	no
Format version data	yes	yes

---

[1]rpm2cpio can do it, but it's not a standard tool, except on rpm-based systems.

[2]This means that a package can depend, conflict, etc on a package AND (another package OR a third package). Any boolean expression must be representable, no matter how complex.

[3]The package can require that some other package be installed that contains a given file (like /bin/sh)

[4]Copyright info is included in deb packages, but not in an easily extractable format.

2.3.4 ebuild

"An ebuild is a specialized bash script format created by the Gentoo Linux project for use in its Portage software management system, which automates compilation and installation procedures for software packages."[21]

The application is installed by command: emerge name_of_application, emerge is also installing all dependencies. The ebuild name format is usually following: name_of_application-version.ebuild

"Each version of an application or package in the Portage repository has a specific ebuild script written for it. The script is used by the emerge tool, also created by the Gentoo Linux project, to calculate any dependencies of the desired software installation, download the required files (and patch them, if necessary), configure the package (based on „USE flag" settings), compile, and perform a sandboxed installation (in /var/tmp/portage/[ebuild name]/image/ by default). Upon successful completion of these steps, the installed files are merged into the live system, outside the sandbox.

Although most ebuilds found in the Gentoo Portage repository are used to compile programs from source code, there are also ebuilds to install binary packages, ebuilds that install only documentation or data such as fonts, and basic ebuilds called „metabuilds" which solely trigger the installation of other ebuilds (such as the GNOME or KDE metabuilds)."[21]

Chapter 3

Analysis of the client tools

Client tools have to be installed on each machine, which should has been registered and managed by Spacewalk. There is more tools, which have the speficic purposes like registering, running scripts or distributing configurafion files. The description is based on the source code of tools [9].

3.1 rhnlib

Rhnlib is a collection of python modules used by the Spacewalk. It is the base package for all client packages. It provides a secured communication channel between server and client using SSL, therefore it requires pyOpenSSL. Rhnlib provides XML-RPC interface for sending and receiving messages from the server. These messages are processed by the other client tools.
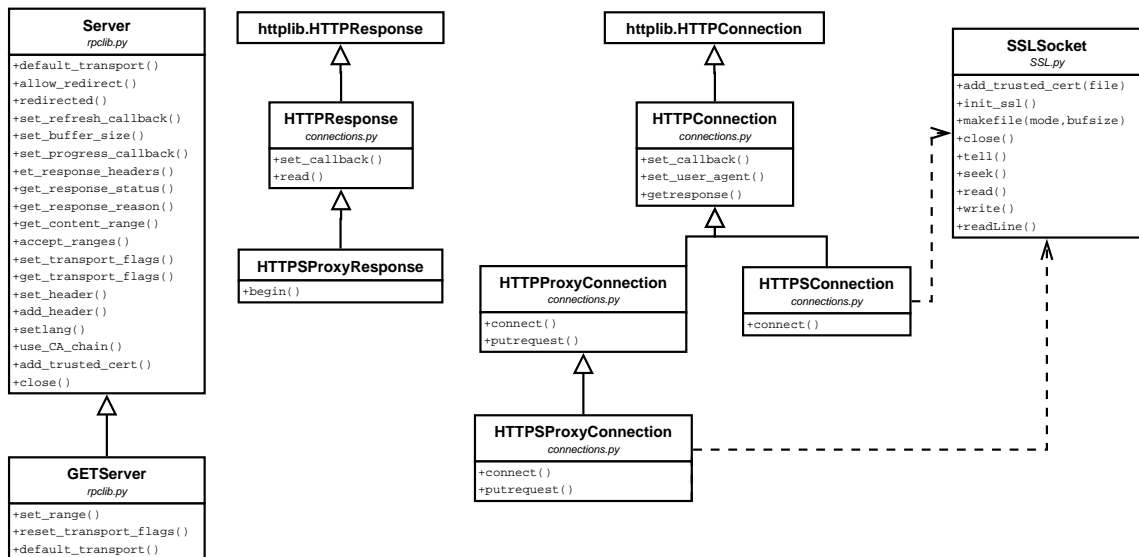
Figure 3.1: The most important classes of rhnlib

3.2   rhn-client-tools

Rhn-client-tools is the package with collection of tools and libraries for the managing of the client system. The tools:

- rhn_check - polls the Spacewalk server and receives queued actions for the system. After receiving the actions, rhn_check will then process them, and return the results to Spacewalk.

- rhn-profile-sync - connects to Spacewalk and refreshes the data stored for the system. It updates data about installed packages, machine hardware, and virtual guest instances.

- rhn_register - is a client program that registers the system to Spacewalk. It can run both in graphical and text modes.

- rhnreg_ks - is a client program for registering a system to Spacewalk. It is designed to be used in a non-interactive environment. All the information can be specified on the command line or stdin.
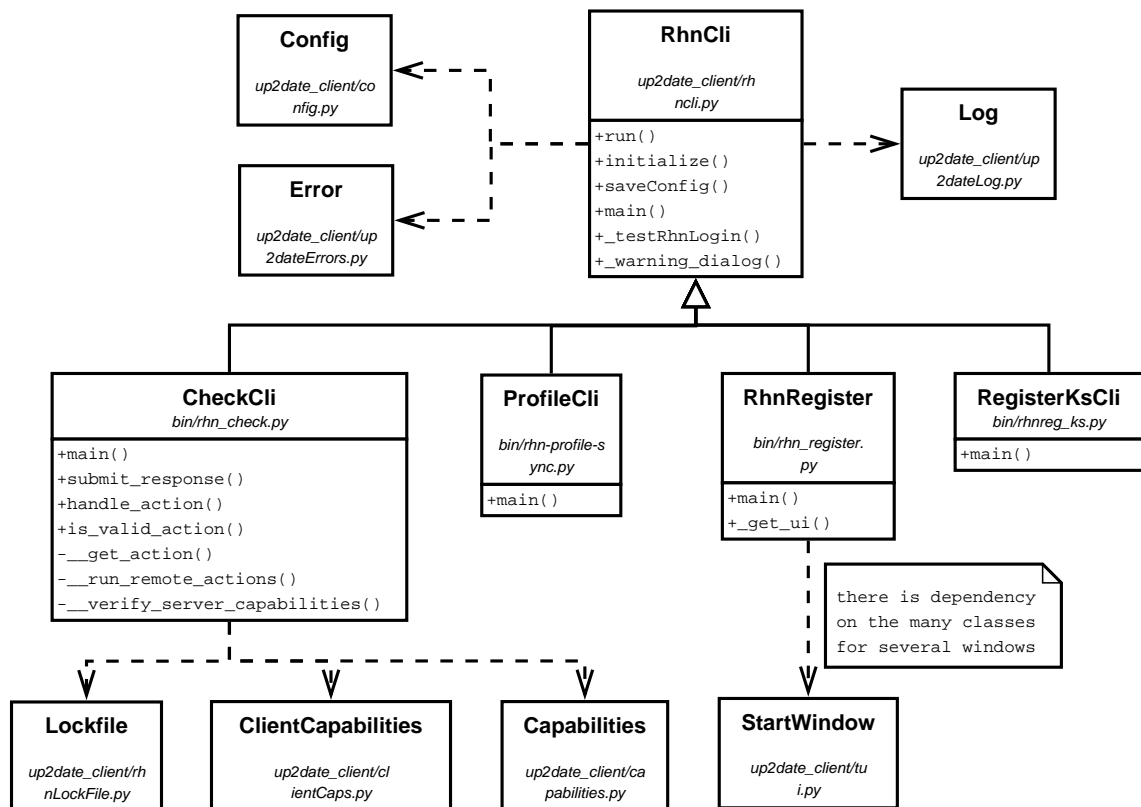
Figure 3.2: The class design of the client tools

All libraries from this package are supporting for tools, which were noted in a previous paragraph. Main capabilities are focused in hardware and packages information. For hardware control is used Hardware Abstraction Layer and linux system's tool `uname`. Library

provides complete information about cpu, this is parsed from file `/proc/cpuinfo`. The data about memory is found out from file `/proc/meminfo`, but this proces is different between kernel version 2.4 and 2.6, what is given by various formats. Functions `gethostname()` and `gethostbyname()` are providing a base data about networking.

Python has the modules transaction and rpm, which are used for getting information about packages. For each package is stored name, version, release, architecture and optionally epoch and cookie. The name is name of package, the version is version of packed content, the release is a number of this version build, the architecture describes an aimed architecture. If package is indepedent on architecture, it is marked as noarch. The epoch term allows package manager to replace old packages with a bit different names, for example packages, which are release candidates and have rc in name.

3.3 rhncfg

Rhncfg contains all libraries and functions for running configuration actions scheduled via Spacewalk. Main actions are distribution of the configuration files and running scripts. Each script can be run with the privileges of selected user and group, there can be set timeout and time, when running of that script will be allowed, this is done by Spacewalk web GUI.

Another actions are diff of the configuration files, uploading the files or mtime. Diff is needed by compare function of Spacewalk, which can compare all or selected managed files by Spacewalk. The uploading files is usefull, if the configuration files are created on the one managed systems, then they are upload to configuration channel on Spacewalk and after that they are distributed on all systems registered to this configuration channel.

This package provides three binaries:

- rhn-actions-control - tool for allowing or disabling features (deploying, running scripts, uploading, diff) and for getting report of actual status of these features settings. Configuration is stored in directory `/etc/sysconfig/rhn/allowed-actions/`, there are two subdirectories configfiles and script, in each subdirectory is file, which appoints allowed actions, for example if rhn-actions-control was run with a parameter –enable-all there is the file named all in each of these two subdirectories. Another names for this files are deploy, diff, upload, mtime_upload, these files can be combined, if there is not used file all. The design of this program is presented on the figure 3.3.

- rhncfg-client - supports all actions for configuration files: diff, deploying, verifying and listing. The architecture of the tool is shown on the figure 3.4

- rhncfg-manager - maintains parts connected with server part, like creating, listing, removing or downloading configuration channels. The architecture is designed by the same way as the rhncfg-client.

3.4 rhnsd

Rhnsd is a query daemon, which automatically queries the Spacewalk server by running the rhn_check tool, which gets scheduled actions and takes care about them. Rhnsd is written in C language, so it is not architecture independent like other client tools, which are created in python.
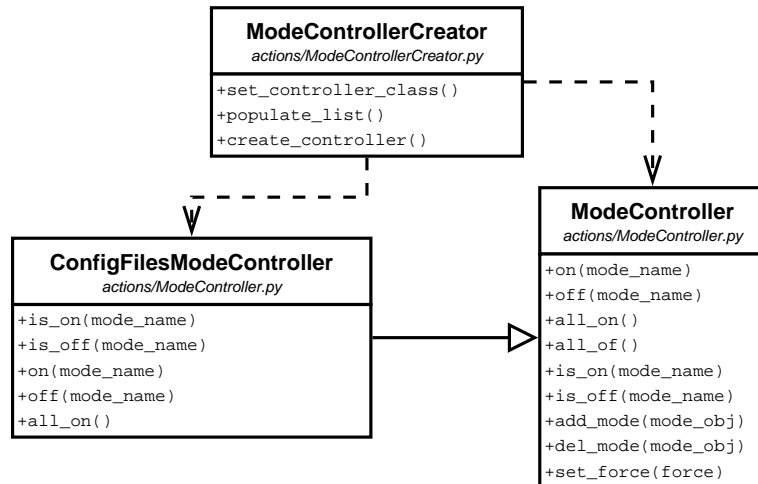
Figure 3.3: The design of classes for controlling configuration modes
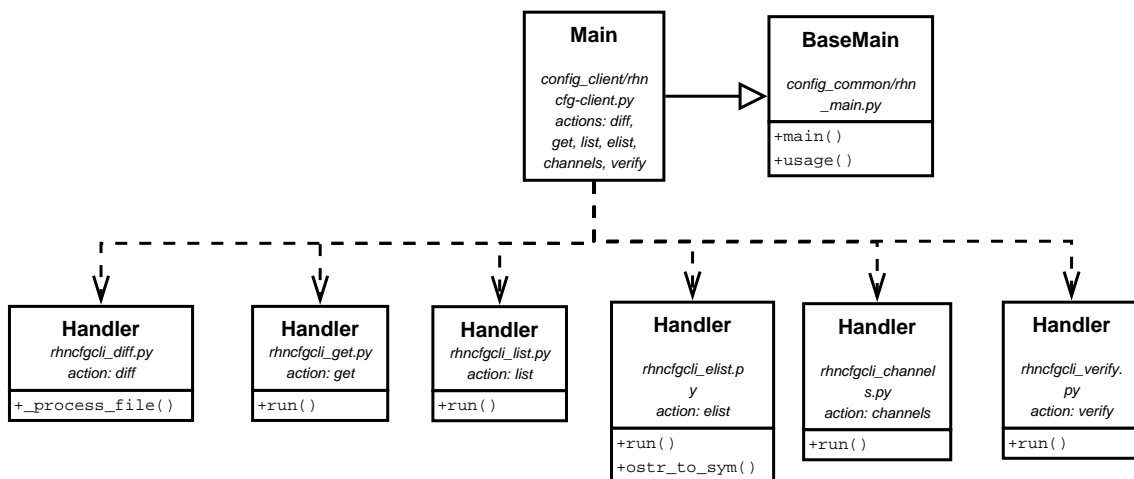
Figure 3.4: The class design of the rhncfg-client

This daemon has only the one configuration parameter, it is interval in minutes, which is stored in file `/etc/sysconfig/rhn/rhnsd` and it schedules how often rhn_check should be run. Due to using rhn_check this package depends on package rhn-client-tools.

Rhnsd checks if it is running with root privileges, other way it ends. After that it makes control, if the only one instance is running. With function `openlog()` is opened connection to the system logger and the log messages are written by `syslog()`. After setting the signals handlers is made final initialization and the the infinite while loop is started. There is run the rhn_check after waiting the interval.

3.5   rhnpush

Rhnpush uploads binary or source packages and their headers to the Spacewalk into speci-fied channel and allows several other channel management operations relevant to controlling

what packages are available per channel. The action can be done only by the channel administrator for selected channel, so the username and the password have to be provided.

Process of uploading package:

- Parse given console parameters by class OptionParser, which is a part of standard Python module optparse or optik

- Get configuration from configuration files by class ConfManager, which is included in rhnpush package

- Create instance of UploadClass with given parameters and configuration

- According to parsed options take an appropriate action (in our case the uploading of package)

- Set force flag, organization name, URL, channels, server and finally authenticate

- Ping the url for uploading packages and get an answer

- Check the answer of server for header X-RHN-Check-Package-Exists to know if the server supports the capability fo check existence of file on the server

  - If the capability is supported, the client will compute MD5 sum of the package and create a base information about package like name, version, release, epoch and architecture and add the information of an aimed software channel. This information is pushed to Spacewalk and the answer of an existence of this package is returned.

- If the package does not exist on the server or force flag is set, the package will be uploaded to Spacewalk. The header is created and it is sent with the data stream of the whole package. There are two methods of sending, which are selected according to the result of the ping from the previous step:

  - POST - uses http protocol, rhnpush identifies itself as agent rhn-package-upload and all headers have prefix X-RHN-Upload. There are these headers: Package-Name, Package-Version, Package-Release, Package-Arch, Packaging and File-MD5sum. The body of the http packet is filled with the package.

  - XML-RPC - standard XML-RPC call is used with given username and password or the identificator of open and autorized session. To this data is attached hash with information about package, target channel and packaging type. This call use the public Spacewalk API.

In the previous text some important classes was mentioned, the design and methods of these classes are presented on the figure 3.5. The packages can be uploaded into specific channels or generally to Spacewalk, and then they are marked as unmanaged and the channel administrators can move them into a concrete channel later. The condition of succesfull upload to channel is compatible archicture of package and channel. In the case of an incompatibility the package becomes unmanaged and is not subscribed into any channel.
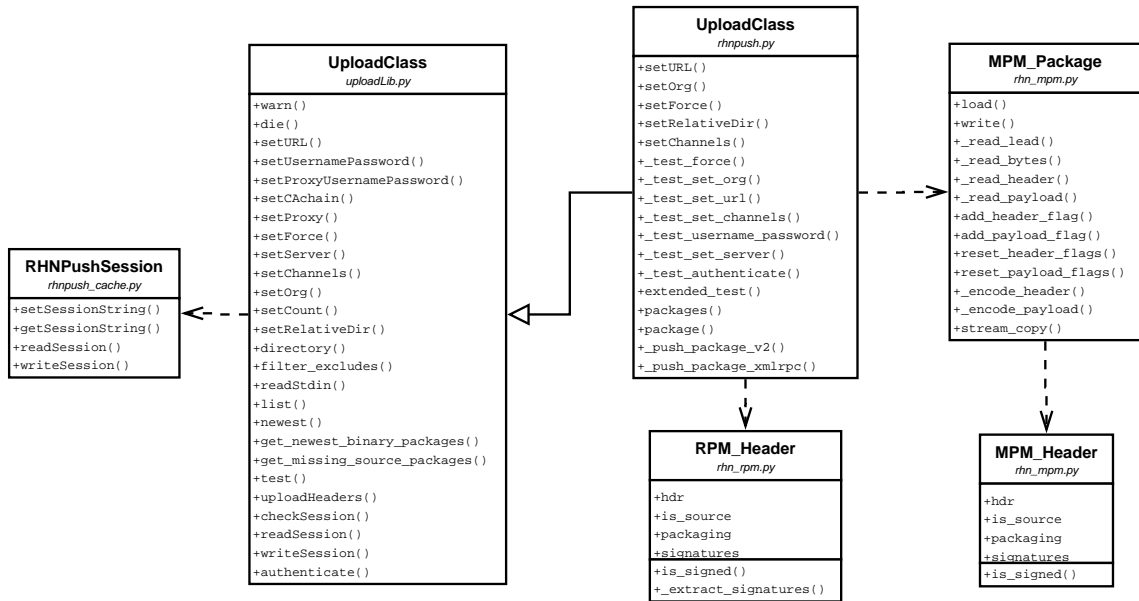
Figure 3.5: The design of the main classes from rhnpush

3.6 yum-rhn-plugin

Yum-rhn-plugin is a plugin for yum, the package management utility. It provides support for accessing Spacewalk to get software updates. Plugin uses standard yum plugin architecture. On the figure 3.6 the plugin classes are shown. In the following text there will be a description and an explanation of the plugin.
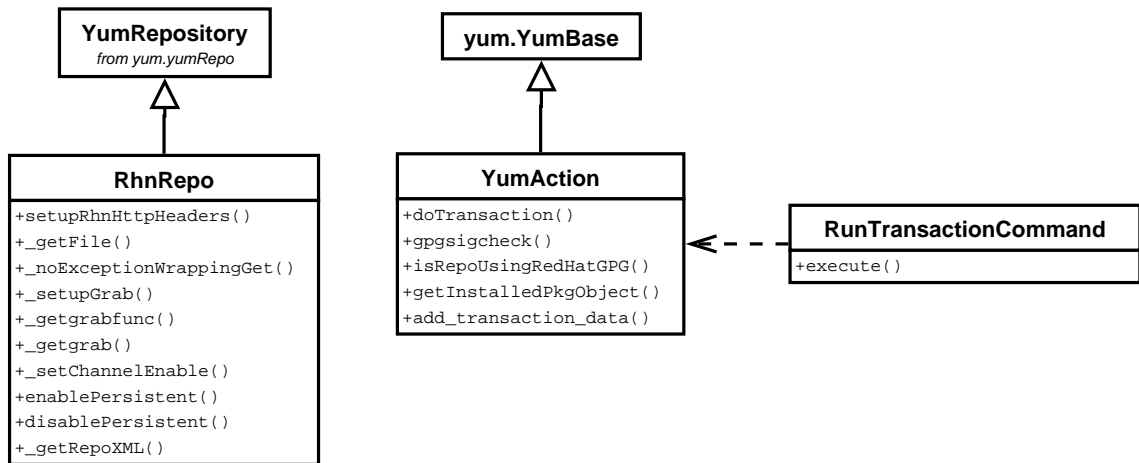
Figure 3.6: The classes used by yum-rhn-plugin

The plugin is formed by 2 source files: `rhnplugin.py`, which contains hook functions for implementation of yum plugin, and `packages.py`, which is implementing provided actions similar to other spacewalk client tools. There is a list of these actions, each action has the function with a same name:

- update - gets a list with packages, which have to be installed. From that list the transaction data are created, these data are represented by the python dictionary type and contain the array, where each package is stored with the requested action, what is „install" in that case. Then the transaction is run with these data. The transaction is an ability of yum, which can ensure that the failed action will be completly rollback and it will have no bad influence to the state of the system.

- remove - works the same way as the update action, but the important difference is the action, which is done with the packages in the received list. In that case the packages are removed from the system with all dependencies, which can not be installed on the system without the packages marked for deletion.

- refresh_list - gets an actual list of installed packages from the client. This action is called everytime, when some modification with packages is done by the other function.

- fullUpdate - calls the update command on the yum. It checks all available package updates and install them.

- checkNeedUpdate - checks if the locally installed package list changed. If there was a change, the update would be ensure by the refresh_list action.

- runTransaction - gets the transaction data as a parameter and it runs the transaction on a group of packages. This was historicaly meant as generic call, but is only called for a rollback. Therefore all *install* actions are changed to *rollback* actions, where dependencies and obsoletes will not be checked, because plugin assumes that state to which it is rolling back should be correct.

- verify - gets the list of packages, which have to be verified. The verification is composite from the control of the package, which has to be installed and in correct state.

- verifyAll - verifies all the packages on the client system.

Transactions use help class *RunTransactionCommand*, which depends on class *YumAction*, what is shown on figure 3.6, and that class calls inherited method `execute` from yumBase to accumulate transaction data, then by method `buildTransaction` the transaction is prepared, and if there is no error, the transaction is performed by softly reimplemented method `doTransaction`.

In the file `rhnplugin.py` is the communication with yum granted by 2 hook functions: `init_hook` and `posttrans_hook`. In init_hook the RHN channels are setup. Function logins into Spacewalk using backend API and gets a list of RHN channels from the server, then make a repo object (class RhnRepo) for each one. This list of repos is then added to yum's list of repos via the conduit (see 2.2.2). As we see, that code creates the ability to get the packages from our Spacewalk server without need to download packages from other extraneous repositories and also it makes easier way to manage and distribute our packages, which are not public. There is an exception for command clean, because for that command it is not neccesary to login to Spacewalk, so only dummy repositories are produced instead of channel's repositories.

The function posttrans_hook is called after transaction. The aimed of the action is update of spacewalk profile, exactly part with information about packages. This is done by rhnPackageInfo modul, which is provided by package rhn-client-tools, so it is clear, that

this plugin requires that package. From that package the modul for handling the software channels is also used.

The class *RhnRepo* is inherited from *YumRepository* and it is used for each software channel as it was mentioned above in the description about the init_hook. The class can login to Spacewalk and setup special headers for downloading requested package, then the package is normally downloaded from Spacewalk. The information about obtainable packages from channel join with the concrete instance of this class are included in repodata XML file, which is stored and get from url:

`http://spacewalk_address/XML-RPC/GET-REQ/channel-id/repodata/repomd.xml`

This file contains information about other files, which are needed for correct repository structure: `filelists.xml.gz`, `other.xml.gz` and `primary.xml.gz`, this files are downloaded by plugin in the next step and according to information from them the packages can be installed or updated from the repository. All these requests have to contain special headers with authorized session identification.

Chapter 4

Client tools for Debian

All the packages in the previous chapter have to be ported for Debian. The advantage is the same operating system type, the both RPM and DEB packages are for Linux, if there would be required porting for another operating system like Haiku, it could bring more problems. Due to that fact, the porting has included only resolving the problems caused by the differences between Debian and the systems based on Red Hat Enterprise Linux.

4.1 Design

The changes, that have to be done, were searched by the code reviews and the following compiling DEB packages on aimed platform for the tests. Each package needs the different alternations, which will be described in the next paragraphs.

All the packages with client tools depends on package rhpl, which is a library of python code used by programs in Red Hat Enterprise Linux and is no more developed. According to this knowledge I create patch, which removes dependencies on rhpl and uses standard python modules. This will make porting for Debian easier and it will help the original packages to be not dependent on no longer supported package. There are 2 libraries, which are used in client tools and will be replaced:

- rhpl.translate - module gettext is the perfect replacer, and now it is a lot more functional than rhpl.translate

- rhpl.ethtool - there is python package python-ethtool, which provides almost the same functions

rhnlib

Rhnlib consists of the few python files, so there have not been necessary any changes in the code, the important thing was correct compilation of the package. The compilation was made by class setup from distutils.core.

rhn-client-tools

The code from this package needs more changes, because it works with hardware, packages and system information. The hardware information seems to be correct, so now there are not designed any patches, but it could be changed in the the next development cycles after more detailed testing. The dependencies of this package are shown on the figure 4.1. There

are the both types of dependencies: Depends and Build-Depends, the differences between them are explained in section 2.3.1.
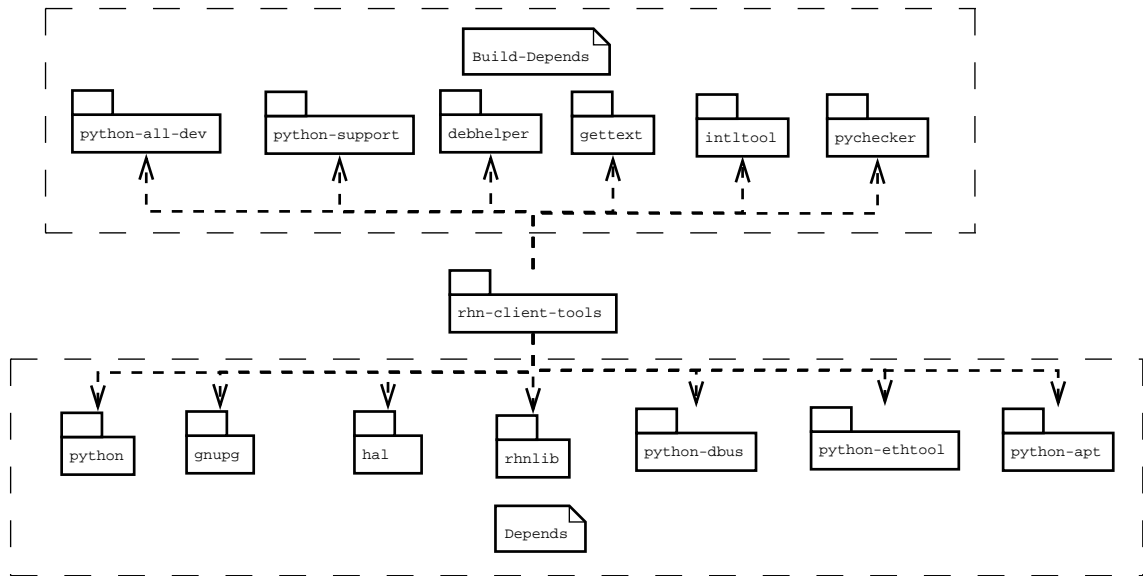
Figure 4.1: The dependencies of rhn-client-tools deb package

The system information is parsed from file in `/etc/issue` and from function `uname()`. The content of issue file on Debian Lenny is Debian GNU/Linux 5.0. From this data two parameters are get, the operating system release: Debian GNU/Linux and the version: 5.0. The next parameter is the release and the kernel version was choosen for this purpose, it is get by function `uname()` from the module os. The architecture of the system is get from this function too.

The most amount of work will require an elaboration with packages and info about them. On the RPM systems this process is done by modules RPM and transaction. On the Debian system it has to be replaced with some alternative for processing DEB packages. The good choice seems to be python module apt.

Class *apt.Cache* accesses all available packages. From this cache is possible to find out which packages are installed and get info about them. There are differences between RPM and DEB, DEB has not epoch and cookie, so this properties will be leaved empty. With the architecture and the name there is no problem, but DEB has only version and RPM has version and release. That would be solved by trying parse release number from version, if there is last number in version separated by char -, else it would be leaved as an empty string.

rhncfg

Rhncfg seems to be simple for porting same way as rhnlib. The process of deploying configuration files has no specific parts, that would require changes. The same situation is for running scripts, where standard linux functions are used. The changes will have to be make to Makefiles due to a little bit different compilation of the package, but this is needed for every package.

rhnsd

Rhnsd will need rework, because the package with daemon for the Debian system has a different structure. In the RPM package the script, which will run daemon is in the path `/etc/rc.d/init.d/` and after package installation the commnad `/sbin/chkconfig --add rhnsd` is run. In the DEB package that script is in the path `/etc/init.d` and after installation it is registerd to the operating system by command update-rc.d rhnsd defaults. Similar way it is removed from system registration, on RPM system by `/sbin/chkconfig --del rhnsd` and on the DEB system by `update-rc.d -f rhnsd remove`. More over on the DEB system there has to be added an option `force-reload` in the starting script.

On the RPM system there is file `functions` in the directory `/etc/rc.d/init.d`. This file contains functions, which are used by most or all shell scripts from the directory `/etc/init.d`. This file is used by rhnsd too, and because it is not part of the Debian, it will be neccesary to make package with this file and find an appropriate directory for it such as `/usr/share/rhn`.

rhnpush

First step is removing the files with classes for RPM packages, which are no more needed, concretely it was files: `rhn_mpm.py`, `rhn_rpm.py`, `rpm2mpm.py` and `solaris2mpm.py`. Then there has to be created file with class to do the same work with DEB package. The client needs to send the base information about the package to server, so the there are no so big requirements on the client side, because the detailed information about package is gained from it on the server.

For that purpose the file `rhn_deb.py` was created. The main class of that file is *deb_Header*, which gets the name, version and architecture of the package. RPM packages has also parameter release, which is required for uploading package. The solution with DEB packages is an attempt to find char „-" in the version parameter. If this char is found, the version is divided into two parts by this char. The first part is taken as version and the second part as release. If the char is not found, the char `X` is inserted as the release.

For getting information about DEB package the python modul apt [6] was used. It is used also in rhn-client-tools. The class *apt_inst.DebFile* takes filename of the package, this class provides attributes:

- control - the package version, as contained in debian-binary. - returns the TarFile object associated with the control.tar.gz member

- data - the TarFile object associated with the data.tar.gz,bz2,lzma member

- debian_binary - the package version, as contained in debian-binary

With the method control the information from control file is unpacked, but it is returned as a string. The better manipulation would be if the parameters can be accesible as an attributes. For that purspose *apt_pkg.TagSection* class is proposed. It takes the string as a parameter, parses it and produces the object, which provides helpful methods for example as:

- find_flag(key) - returns boolean value for the key. An example for such a field is Essential

- get(key, default=") - returns the value of the field key if available, else return default

37

- keys() - returns a list of available keys

Except this, the keys are accesible in the way: instance[key], so the package version can be easily get by instance['Version']. The next modification affects the package architecture value. The suffix `-deb` has to be added to string value obtained from control information. This modification reflects the updates on the server side and will be made more clearly in the following text about server changes.

In the sent headers the parameter X-RHN-Upload-Packaging, there was value `rpm` changed to `deb`, and also Content-Type was overwritten to application/x-deb.

The other supporting packages

These packages were created to make work with client tools easier or to support some functions of these tools.

rhn-setup

Rhn-setup is something like metapackage. It does not provides any data or funcionality. Its task is to install all the client tools, what is reached by insertion of these client tools packages in Depends field.

osad

This package provides OSA deamon agent. Agent receives commands over jabber protocol from Spacewalk Server and commands are instantly executed. This package effectively replace rhn_check command, which check in Spacewalk Server only in some period usually given by paramter interval for deamon rhnsd and also run automatically by rhnsd. This way of executing commnands on client is very worthwhile for commands, where time between submission and execution of command is critical, for example the security updates for packages or the restart of system.

Osad is written in python and is completly independent on platform, so the creation of the DEB package was the task without any bigger problems, just the correct names for debian packages in Depends field had to be find.

jabberpy

Jabberpy is simple python module, which is used by osad. It is a python module for the jabber instant messaging protocol. It deals with the XML parsing and socket code, leaving the programmer to concentrate on developing jabber based applications. It produces richly featured easy to use library for creating both jabber clients and servers.

Jabberpy is also system independent and can be use on Windows systems too, so the process of packaging was identical to osad package.

python-ethtool

Python-ethtool provides python bindings for the ethtool kernel interface, that allows querying and changing of ethernet card settings, such as speed, port, autonegotiation, and PCI locations. It is used by rhn-client-tools for getting network information. It is written in

language C, so the built package will be available only for the same architecture as the compiling machine has, what is in our case i386. This is difference to python moduls written directly in python, which are independent on architecture.

The package python-ethtool exists for Fedora and creating DEB package follows standard procedure noted in the next section.

rhn-functions

The linux systems based on RPM contains contain file **/etc/init.d/functions**, what is bash script and provides functions to be used by most or all shell scripts in the **/etc/init.d** directory. This file is missing on the Debian systems, but it is used for example by rhnsd. So the package rhn-functions was created for distribution of this file. There was a change in placing of file and was moved to **/usr/share/rhn/functions**, because it will be probably used only by our ported packages.

The file functions affords usefull functions for checking process id, getting information from fstab, killing or finding out the status of process, or running daemons.

4.2   Process of creating DEB package

We can take a look on the process of creating deb package. This procedure will not go into the details, but it should show main steps.

- get package.src.rpm

- extract payload from package by

  `rpm2cpio package.src.rpm | cpio -idmv --no-absolute-filenames`

- `mv package.tar.gz package.orig.tar.gz`

- `tar xfz package.orig.taz.gz`

- set name of packager by `export DEBFULLNAME=name`

- set email of packager by export `DEBEMAIL=e-mail`

- run `dh_make -c gpl` to create base structure of debian package

- remove example files by `rm debian*.ex debian*.EX`

- remake source files to make program functional on Debian

- edit files in directory debian, mainly `dirs`, `control` and `rules` files

- use `dpkg-buildpackage -rfakeroot` to build package

Chapter 5

Analysis of the server side

On the server side the changes had to be done for correct accepting and handling DEB packages. The first step was an editing a database schema. The next changes patched the backend code for parsing DEB packages and storing information about these packages in the database.

5.1 Database schema

The Spacewalk does not know the type of package `all`, because on the RPM systems there is used an equivalent `noarch`. Moreover there are differences on the supported platforms between systems from Red Hat and Debian, which implicates the another distinct types of packages.

The table with architectures has to extended with architecture DEB. The same way there has to be added channels for debian packages connected to an archicture type deb. The name for the DEB channels can be derived from RPM channels by adding `-deb` on the end of the name. In the table with the types of the packages the connection has to be created between deb architecture and types of packages for this architecture. To the server architectures the debian types have to be added with similar name to rpm types, for example from `i386-redhat-linux` will be deduced `i386-debian-linux` with difference in the connected architecture type.

Description of the tables:

- rhnArchType - The list of the supported architectures. The deb architecture has to be added to this table.

- rhnChannelArch - The table with channels for specific processor architectures such as intel or alpha. These software channels are connected to the base architecture (rhnArchType) and that fact influences the packages, which can be added to these channels.

- rhnPackageArch - The supported package's architectures are inserted in this table. They are connected to rhnArchType by the same way as it is done in rhnChannelArch.

- rhnServerArch - The list of supported machines types. The client machine sends one label from that list, when the registration to the Spacewalk is proccesing. These servers are connected to the their architecture from rhnArchType too.

- rhnChannelPackageArchCompat - This table stored the connection between channels and packages, which can be added to channels. They have to have the same architecture.

- rhnServerPackageArchCompat - This table stores the connection between servers and packages, which can be installed on that servers. The same architecture of server and package is necessary.

- rhnServerChannelArchCompat - The connections between servers and channels are inserted into this table. The server can be subscribed into the software channel only if it the connection between them is stored in that table.

- rhnServerServerGroupArchCompat - The table stores the entitlements for servers. The enabled Spacewalk's actions like managment or provisioning for servers are given by that table.
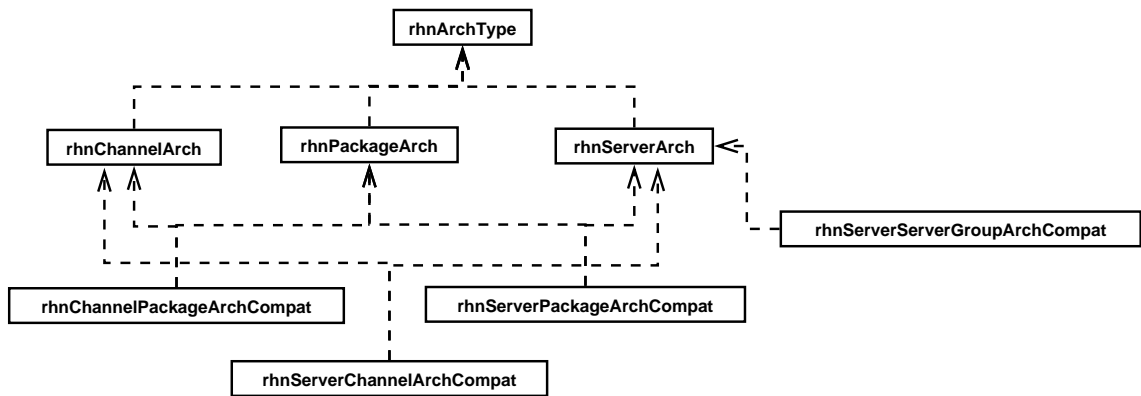
Figure 5.1: The connection between sql tables

The dependencies between database tables are shown on the figure 5.1. For easier creating the connections, there are several stored procedures:

- LOOKUP_ARCH_TYPE(arch) - search architecture type id according to given label (examples: deb, rpm)

- LOOKUP_SERVER_ARCH(server) - search server architecture type id according to given server label (examples: i386-redhat-linux, amd64-debian-linux)

- LOOKUP_CHANNEL_ARCH(server) - search channel architecture type id according to given channel label (examples: channel-ia64, channel-alpha-deb)

- LOOKUP_PACKAGE_ARCH(package) - search package architecture type id according to given package label (examples: noarch, all-deb)

- LOOKUP_SG_TYPE(server_group) - search server group type id according to given label (examples: sw_mgr_entitled, provisioning_entitled)

5.2 Backend

There are two different parts of backend, which have to be modified. The first part is code, that receives the package from rhnpush, parses and stores the package on filesystem and inserts its header information to database. The second part has the goal to supply these packages for all clients, which have authorization to download these packages. This authorization is declared by subscription to the software channel, where the package is saved.

5.2.1 Receiving and storing the package

This part can parse and store RPM package, the way how it is done will be introduced in the following text and based on that knowledge there should be designed the changes for doing these actions with DEB packages.

The request for uploading package from rhnpush is received by Apache and it is redirected to the appropriate handler, what is in this case handler APP, this handler recognizes, that the package should be uploaded and redirect the process of uploading to handler PACKAGE-PUSH.

This handler stores package in the temporary file and then calls the function to parse that package and this function returns the object with header information, and object represent the payload stream. From header data the type of checksum is found out and that checksum method is run on the payload. The result is compared to the value, that was received in the HTTP headers, it they are different, that error about mismatching information is returned back to the client.

In the next step the path for storing package on the filesystem is computed. This file path is checked for existence and if there exists file with the same name, the error about the fact, that file already exists, is returned. In other case the file is saved to that path, the path seems like that /var/satellite/redhat/1/b12/python2.6/2.6.4-6/i386-deb/b12350499229636b605a607429d0c340/python2.6-2.6.4-6.i386-deb.deb where we can see, that the files are stored to directories named by the first three letters of checksum, the next subdirectories are named according to package name, version, architecture and the final directory, where the package is stored is named according to checksum value.

The following step is creation of the object, that represents the package. It contains all the information from header such as name, version, dependencies, files in package, etc. and some other computed values like the size of package. That object is added to batch, which is given to the instance of Oracle backend. This instance handles storing the information in all tables, where it has to be inserted. In the end the transactions are commited.

Design for accepting DEB package

The upper analysis implies the two main parts for adjustments. It is parsing of received package. The first step will be naming of temporary files to know, that it is DEB package, this naming can be based on the packaging type, which is sent in the HTTP headers.

The second part of modification will be consists from the fabrication of the class for the DEB package representation. It will be similar to the class for RPM package, because we want to have these classes compatible at the high level, because then there will not be required the changes in the Oracle backend class, which converts information from that classes to the database.

5.2.2 Providing package to client

RPM packages are provided by repository for the package manager yum. This repository consists of XML files with packages metadata and it is generated by Taskomatic. Taskomatic is run automatically, when there is some change in the software channel. Taskomatic has to be extended for capability to generate files for APT repository, what would ensure access to packages by Debian systems. The design for this feature is shown on the figure 5.2. The APT repository requires existence of file `Packages.gz`, which contains information about all packages in the repository.

```
┌──────────────────────────────────────┐          ┌──────────────────────────────────────┐
│           RepositoryWriter            │          │            DebPackageWriter           │
├──────────────────────────────────────┤          ├──────────────────────────────────────┤
│ -pathPrefix: String                   │          │ -filenamePackages: String             │
│ -mountPoint: String                   │ - - - -▷ │ +DebPackageWriter(prefix:String)      │
│ +RepositoryWriter()                   │          │ +addPackage(pkgDto:PackageDto): void  │
│ +writeRepomdFiles(channel:Channel)    │          │ +generatePackagesGz(): void           │
│ -writeDebRepoFiles(channel:Channel)   │          │ -addPackageDepData(): void            │
└──────────────────────────────────────┘          └──────────────────────────────────────┘
```
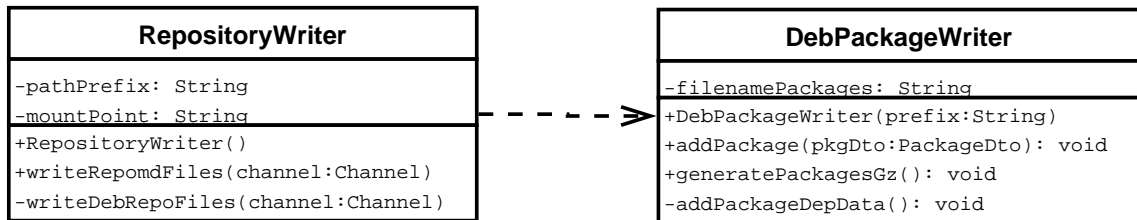
Figure 5.2: The design of modification for Taskomatic

The next step is accesing the file Packages.gz. Repository files for RPM channels are obtainable through XML-RPC request, which is controlled by the backend code for HTTP headers with an authorization data. On RPM systems this data is sent by yum-rhn-plugin. There is a problem, that APT on Debian systems does not provide similar plugin architecture as yum, so it is not easy to add these authorization steps for getting Packages.gz from APT repository.

Chapter 6

The implementation of the server side changes

The implementation can be divided into 3 logical parts:

- database
- accepting DEB package(s) from client
- providing DEB package(s) to clients

In the following sections these modifications will be introduced into more details.

6.1 Database

According to analysis of database schema there were added these values to tables. The architecture deb was added to supported architecture types:

```
insert into rhnArchType (id, label, name) values
    (rhn_archtype_id_seq.nextval, 'deb', 'DEB');
```

The channels names was were adapted by addind suffix `-deb` as it can be seen on this:

```
insert into rhnChannelArch (id, label, name, arch_type_id) values
    (rhn_channel_arch_id_seq.nextval, 'channel-ia64-deb',
    'IA-64 Debian', lookup_arch_type('deb'));
```

The same way of naming was used for package's architectures. There are some architectures like ARM, which are not supported by RPM, so the name has not to be necessary extended by the suffix, but for better recognitation the suffix was used.

```
insert into rhnPackageArch (id, label, name, arch_type_id) values
    (rhn_package_arch_id_seq.nextval, 'arm-deb', 'arm-deb',
    lookup_arch_type('deb'));
```

The server architecture name was created easily by using label debian instead of redhat, but only for the architectures, which are supported by Debian.

```
insert into rhnServerArch (id, label, name, arch_type_id) values
    (rhn_server_arch_id_seq.nextval, 'alpha-debian-linux', 'alpha Debian',
    lookup_arch_type('deb'));
```

New channels for DEB packages were connected with the packages, which can be uploaded into these channels.

```
insert into rhnChannelPackageArchCompat (channel_arch_id, package_arch_id)
    values (LOOKUP_CHANNEL_ARCH('channel-amd64-deb'),
            LOOKUP_PACKAGE_ARCH('amd64-deb'));
```

The same procedure was done for the server types.

```
insert into rhnServerPackageArchCompat (server_arch_id,
  package_arch_id, preference) values
    (LOOKUP_SERVER_ARCH('ia64-debian-linux'),
     LOOKUP_PACKAGE_ARCH('all-deb'), 1000);
```

When the system is registering to the software channel, the control if the channel has correct packages for that system has to be done. For that purpose the channels and server types were connected.

```
insert into rhnServerChannelArchCompat (server_arch_id, channel_arch_id)
    values (LOOKUP_SERVER_ARCH('powerpc-debian-linux'),
            LOOKUP_CHANNEL_ARCH('channel-powerpc-deb'));
```

Finnaly there were established the capabilities, which can be done with the systems.

```
insert into rhnServerServerGroupArchCompat (server_arch_id,
    server_group_type) values
    (lookup_server_arch('s390-debian-linux'),
     lookup_sg_type('sw_mgr_entitled'));
```

6.2 Accepting DEB package

After accepting HTTP request the package is stored as temporary file. We need to recognize, that it is DEB package, what is done by checking HTTP header X-RHN-Upload-Packaging and setting parameter packaging for function, which stores the package in directory for temporary files. This parameter was added to this function too, the new declaration of function is: `def write_temp_file(req, buffer_size, packaging=None)`, also then the temporary file has to be created with the name, which satisfies rule, that it ends with `.deb`. For that purpose was used the method `tempfile.NamedTemporaryFile(suffix=suffix)`, which takes the suffix set to `.deb`.

The next step is parsing header of DEB package. The class for the header information was named *deb_Header* and its code is in the new file `rhn_deb.py`. The parsing of DEB package is not easy process and it would be redudant work to program python modul for it, if there exists another one. For Debian the python-debian package is available. As part of this work this package was rebuild as RPM package and then it was simply connected with class *deb_Header* to gain needed data about package. The structure of the classes from that package is shown on figure 6.2.

As it was said before, the class for Oracle backend is required. For that intention a class *debBinaryPackage* was developed. It inherits from *rpmBinaryPackage* and the diagram of class is evinced on 6.1

The attribute `tagMap` contains the list of parameters, that are stored in the database. These parameters are accesible as fields of the debBinaryPackage instance. The values are
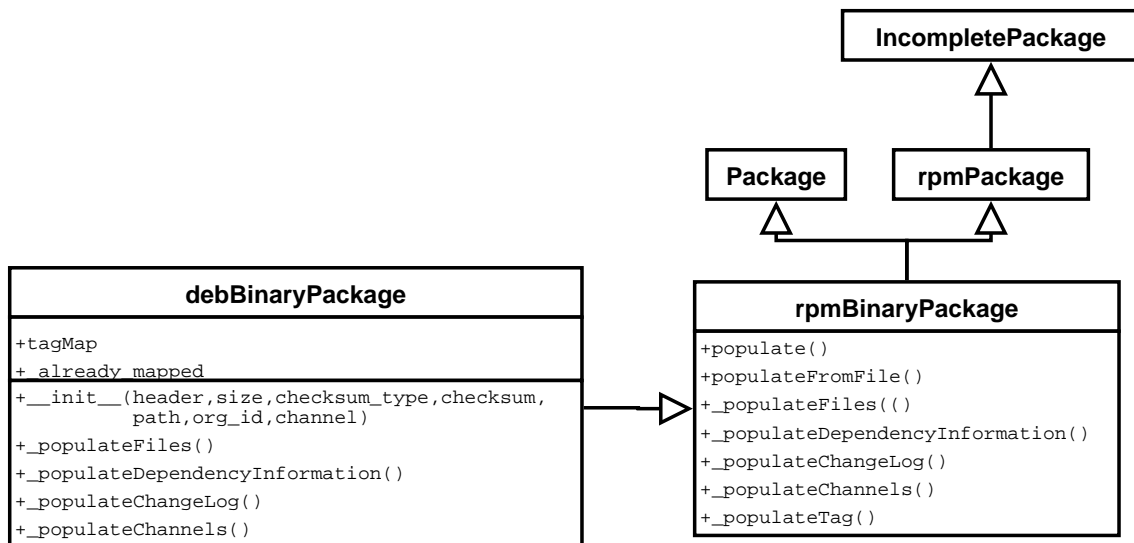
Figure 6.1: The class diagram for debBinaryPackage

checked, if the value is empty, then None is assigned or if it is string in unicode, then it is encode in UTF-8.

6.3 Providing DEB package

Implementation of providing package was not succesfull at all. The repository is generated correctly, but now there is no way how to access it. As it was said before, we need add capability for APT to send authentication data to Spacewalk. The problem is creating plugin for APT, which could modify HTTP request and add headers into it, concretly these headers have to be added:

- X-RHN-Server-Id - ID of registered client system

- X-RHN-Auth-User-Id - ID of user, which has active open session

The creation of this plugin can be theme of some other next work on that project. It is possible to create extension to APT, but it more difficult than plugin for YUM, because APT does not have standard plugin interface, but there can be found some extension written in Python[1], what is the good presumption for communication with other Spacewalk client tools, which for instance contain modul for creating active session for user.

The generating of APT repository was easier task and it was implemented by adding code to Taskomatic. Taskomatic is written in Java and the package, which had to be modified, was `com.redhat.rhn.taskomatic.task.repomd`. After every change on the software channel Taskomatic runs the task for updating repository data and this task is handled by that package. Writing repository files is done by class *RepositoryWriter*, exactly in the method `writeRepomdFiles()`, which gets the channel reference. There was inserted check, if the channel is for DEB packages, the method `generateDebRepository()` is called. This method takes iterator over packages in the channel and information about all packages are

---

[1] `http://git.debian.org/?p=collab-maint/apt-listchanges.git;a=summary`

put into repository files (Packages and Packages.gz). This method uses class *DebPackage-Writer*. The overview of this cooperation is shown on figure 5.2.

This class writes file Packages by using method `addPackage`, which adds these tags about package:

- Package - name of package

- Version - is composited from version and release values stored in database

- Architecture - supported architecture, the value ends with `-deb`, so this suffix has to be removed before writing it into file

- Maintainter - name and email of maintainer

- Provides, Depends, Conflicts, Replaces - these values are added by own method `addPackageDepData()`

- Filename - path to package, Apache has to have permission to read this file, but this is solved by the python code, which stores package on the filesystem

- Size - size of package, it is checked after download by apt (same as checksum)

- MD5sum, SHA1, SHA256 - one of this checksum is written to file (this tag is required). The checksum type is selected according to checksum stored in database, currently it is only MD5, but in the next versions of the Spacewalk new SHA checksum will be added.

- Section - package group

- Description - text information about the package

After this section the new line is inserted and the section with next package can be written. When all packages are proccesed, Packages.gz has to be created by method `generatePackagesGz()`, which reads file Packages, and using *GZIPOutputStream* produces the result.
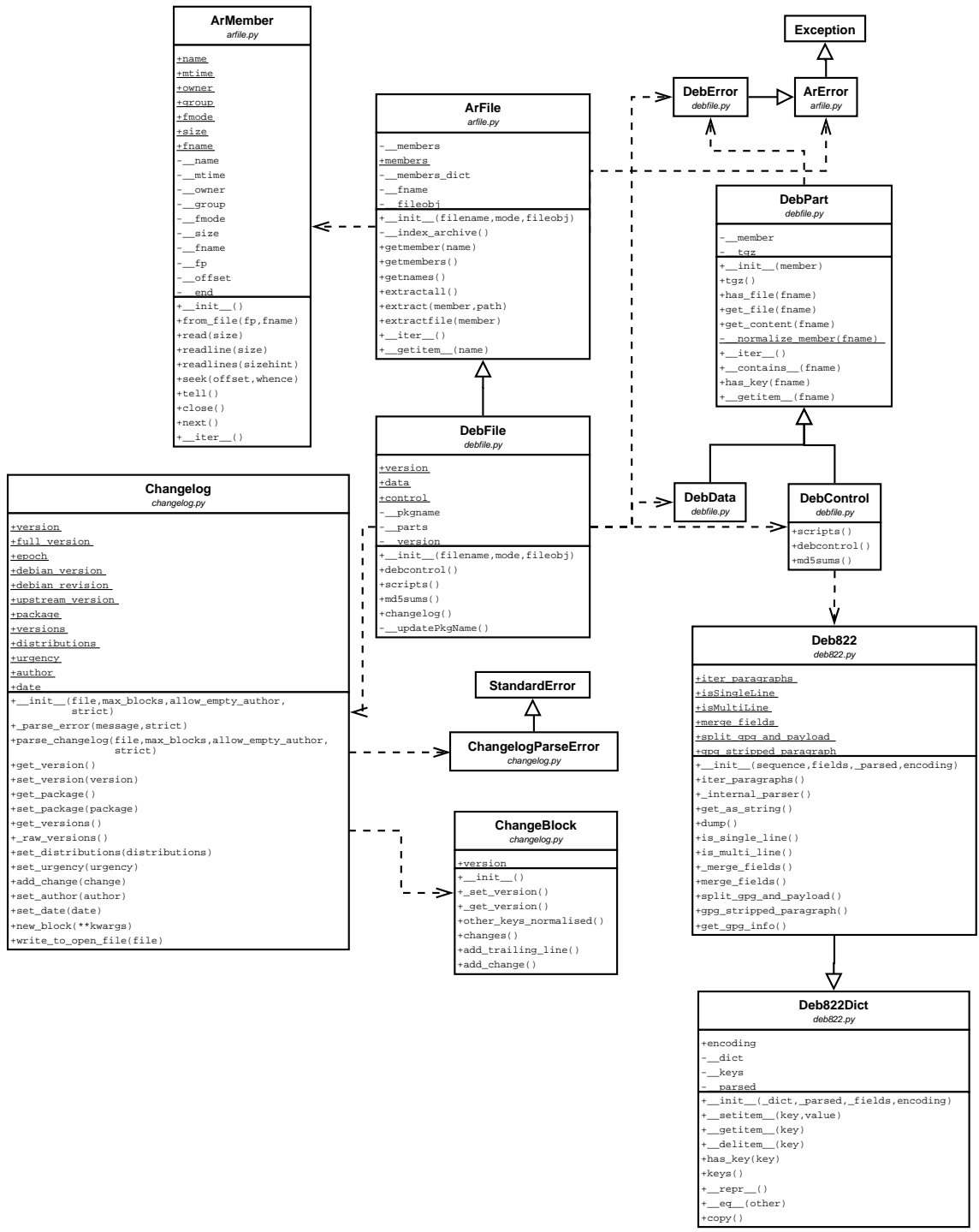
Figure 6.2: The class diagram for classes in python-debian

Chapter 7

The testing

In this chapter we will test all important and ported features. The schema of the tests will be approximatelly running some client tool and showing the printscreen of Spacewalk web interface to see the result.

7.1 Registration of client to Spacewalk

The first step is registration of client system to Spacewalk server. This can be done by rhnreg_ks or by rhn_register. We will show the case with rhnreg_ks. The package rhn-client-tools has to be installed.

```
rhnreg_ks --profilename Debian1 --serverUrl=http://192.168.122.81/XML-RPC
--activationkey=1-debian
```

The explanation of parameters:

- profilename - the name of system, which will be showed as system name in Spacewalk

- serverUrl - the URL of Spacewalk, where client will be registered

- activationkey - is used to register system, system registered with an activation key will inherit the characteristics defined by that key (for example subscribed software channel) and it makes the registration more easier, because no username and password is required, when the key is used.

The result of registration is visible on Spacewalk web interface and is shown on the following schemas B.1, B.2 and B.3.

7.2 The configuration file distribution

The next test will try the distribution of configuration file to registered system. The package rhncfg has to be installed. The configuration file can be created directly in Spacewalk or can be uploaded, we will create our file with web interface as it is shown on B.4. The file is automaticcaly pushed into Sandbox (B.5), because it is marked as under development. To enable distribution of this file it has to be copied into central or system channel. Central channel will provide this configuration file to all client systems registered to this channel and system channel will provide it only for the our system. We will choose system channel.

In the tab for deploying files we will select our file and confirm distribution of this file by clicking on the button Deploy Files B.6. On the next screen the time of deployment can be choose, we will choose a option as soon as possible (B.7). This is the finish of the process on Spacewalk side and we can move to client and check the result.

On the client we will run command rhn_check (with parameter -vvv to see debug output) to take actions and run them. The output from console:

```
debianhost:/home/lukas# ls /root/
debianhost:/home/lukas# rhn_check -vvv
updateLoginInfo() login info
D: login(forceUpdate=True) invoked
logging into up2date server
D: writeCachedLogin() invoked
D: Wrote pickled loginInfo at  12725442.82  with expiration of  12729042.82
seconds successfully retrieved authentication token from up2date server
D: logininfo: {'X-RHN-Server-Id': 1000010022, 'X-RHN-Auth-Server-Time':
'1272535443.61', 'X-RHN-Auth': '7RwMZ9oWPw6OIXO3d9haSA==',
'X-RHN-Auth-Channels': [['debian', '20100429105127', '1', '1']],
'X-RHN-Auth-User-Id': '', 'X-RHN-Auth-Expire-Offset': '3600.0'}
D: handle_action actionid = 59, version = 2
D: do_call configfiles.deploy ({'files': [{'config_channel': '1000010022',
'username': 'root', 'encoding': 'base64', 'checksum':
'7b53845be14ba0bdeb54361d3fc5690', 'filetype': 'file', 'delim_start': '{|',
'file_contents': 'I3Rlc3QgY29uZmlndXJhdGlvbmaWxlCnRpbWU9MTAwMJ1Zz1vZmY=\n',
'groupname': 'root', 'delim_end': '|}', 'selinux_ctx': '', 'filemode': 644,
'checksum_type': 'md5', 'path': '/root/test.conf', 'revision': 1}]},)
configfiles.deploy
D: Sending back response (0, 'Files successfully deployed', {})
debianhost:/home/lukas# ls /root/
test.conf
debianhost:/home/lukas# cat /root/test.conf
#test configuration file
time=1000
debug=off
```

We can see, that the file was successfully deployed and is stored correctly on the given place.

7.3   Running of the script

Running of the script requires package rhncfg and also it has to be enabled by configuration file in local rhn configuration directory, it is enabled if file `allowed-actions/script/run` exists. The script has to be written before the running. In the tab Remote Command in the web interface we will create script (B.8).

The same situaiton on client side as it was for the previous test, we will run command rhn_check (with parameter -vvv) to take up script and to run it. We got the following output to the console, there are common parts with the previous console output, so these parts were removed:

```
debianhost:/home/lukas# cat /etc/group | grep spacewalk-test
debianhost:/home/lukas# rhn_check -vvv
D: logininfo: {'X-RHN-Server-Id': 1000010022, ....
D: handle_action actionid = 60, version = 2
D: do_call script.run (60, {'username': 'root', 'groupname': 'root', 'now':
'2010-04-29 13:04:49', 'timeout': 60,
'script': '#!/bin/sh\naddgroup spacewalk-test'})
script.run
D: Sending back response (0, 'Script executed',
{'output': 'QWRkaW5nIGdyb3VwIGBzcGFjZXdhbGstdGVzdCcgKEdJRCAxMDAyKSAuLi4KRG9
uZS4K\n', 'base64enc': 1, 'process_end': '2010-04-29 13:04:49',
'return_code': 0, 'process_start': '2010-04-29 13:04:49'})
debianhost:/home/lukas# cat /etc/group | grep spacewalk-test
spacewalk-test:x:1002:
```

7.4   Removing the package from client

We can select the packages to be removed from package list of client as we see on B.9. We will select one package and then confirm the removal of this package by clicking on the Remove Packages button and on the following form the time of action is set. As it before we choose the option as soon as possible. The package rhn-client-tools is required for this action on the client.

We will run rhn check again to undertake the action and see the result cleaned from loging info:

```
debianhost:/home/lukas# aptitude search abiword
i   abiword
debianhost:/home/lukas# rhn_check -vvv
D: handle_action actionid = 61, version = 2
D: do_call packages.remove ([['abiword', '2.8.2', '2', '', 'all-deb']],)
packages.remove
D: Called remove_packages [['abiword', '2.8.2', '2', '', 'all-deb']]
Reading package lists... Done
Building dependency tree
Reading state information... Done
Building data structures... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
Building data structures... Done
<Package: name:'abiword' id:118>
(Reading database ... 122006 files and directories currently installed.)
Removing abiword ...
D: Sending back response (0, 'remove_packages OK', {})
D: do_call packages.checkNeedUpdate ('rhnsd=1',)
packages.checkNeedUpdate
D: Called packages.checkNeedUpdate
Updating package profile
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
Building data structures... Done
D: local action status:  (0, 'package list refreshed', {})
debianhost:/home/lukas# aptitude search abiword
c   abiword
```

We can see that the package was succesfully removed by apt. It is documented by the mark from aptitude search, which changed from i (the package is installed) to c (the package was deleted, but the its configuration files remain on the system). The following action was updating the software profile on Spacewalk with actual package list. On figure B.10 we see, that the package is no more in the list.

7.5   Uploading DEB package to Spacewalk

For the uploading package the rhnpush tool has to be used, so the rhnpush package is required. We will run rhnpush command on the client and control the result. We will upload two packages in the software channel debian. This channel is for i386 architecture, so we will try upload one package of architecture all and one i386 package. There are two parameters given for rhnpush it is --server, what is url address of Spacewalk server and -c, which tells the name of software channel, where packages should be pushed. The rhnpush asks for the username and password of the administrator for this software channel, this step can be skipped if there were another package upload by rhnpush in short time ago and the created session is still active.

```
debianhost:/home/lukas# rhnpush -vvvv --server=192.168.122.81 -c debian
acidrip_0.14-0.3_all.deb python2.6_2.6.4-6_i386.deb
Connecting to http://192.168.122.81/APP
Red Hat Network username: lukas
Red Hat Network password:
url is http://192.168.122.81/PACKAGE-PUSH
Result codes: 200 OK
Computing md5sum and package Info .This may take sometime ...
Package acidrip_0.14-0.3_all.deb Not Found on RHN Server -- Uploading
Uploading package acidrip_0.14-0.3_all.deb
Using POST request
Package python2.6_2.6.4-6_i386.deb Not Found on RHN Server -- Uploading
Uploading package python2.6_2.6.4-6_i386.deb
Using POST request
```

If we try to push the packages again, we can see, that the username and password are not asked again and the uploading is skipped.

```
debianhost:/home/lukas# rhnpush -vvvv --server=192.168.122.81 -c debian
acidrip_0.14-0.3_all.deb python2.6_2.6.4-6_i386.deb
Connecting to http://192.168.122.81/APP
url is http://192.168.122.81/PACKAGE-PUSH
Result codes: 200 OK
```

```
Computing md5sum and package Info .This may take sometime ...
Package acidrip_0.14-0.3_all.deb already exists on the RHN Server--
Skipping Upload....
Package python2.6_2.6.4-6_i386.deb already exists on the RHN Server--
Skipping Upload....
```

In the Spacewalk there were added two packages in the debian channel (B.11) and the details of package are available too (B.12).

Chapter 8

Conclusion

The thesis succesfully reached the given aim to add support for Debian operating system to Spacewalk, which by now supports only systems based on RPM. The client tools for main functionality were ported and packed for Debian system. These systems can register to the Spacewalk, hardware, software and network information about them is sent to server. The configuration files and scripts can be distributed to them and scripts can be also executed.

The Spacewalk can manage DEB packages in the software channels same way as it does with RPM packages. In the next work there can be added a feature to synchronize the channel with extern repository with DEB packages. The Spacewalk generates repository with the DEB packages, but there is a problem with the access to repository, because APT is not able to log in to Spacewalk. The future work should include a creation of client tool or plugin for APT, which will be able to authorize and get packages.

The chapter with tests shows the details how the tools work and the results of their actions are presented by the screenshots from the Spacewalk's web interface or by console output from clients.

The thesis brings usable tools and patches for Spacewalk to provide native support for DEB packages, except the problem with accessing the repository there are no other limitations and users can use these tools same way as they use them on RPM platforms. This work can be very helpfull for users, who use Spacewalk to handle considerable number of RPM systems and also have some Debian systems, which have to be administer manually or by not so complex tools.

Bibliography

[1] Edward Bailey. *Maximum RPM*. 2000. ISBN: 1-888172-78-9.
`http://rpm.org/max-rpm/`.[Online; visited 22.2.2010].

[2] Canonical Ltd. Landscape. `http://www.canonical.com/projects/landscape`.
[Online; visited 23.2.2010].

[3] Joey Hess. Comparing linux/unix binary package formats.
`http://kitenet.net/~joey/pkg-comp/pkg-comp.xml`. [Online; visited 12.4.2010].

[4] Aaron Isotton. Debian repository howto.
`http://www.debian.org/doc/manuals/repository-howto/repository-howto`.
[Online; visited 16.4.2010].

[5] Ian Jackson and Christian Schwarz. Control files and their fields.
`http://www.debian.org/doc/debian-policy/ch-controlfields.html`. [Online;
visited 12.4.2010].

[6] Julian Andres Klode. Python apt library.
`http://apt.alioth.debian.org/python-apt-doc/library/index.html`. [Online;
visited 27.4.2010].

[7] WWW pages. Howtosetupadebianrepository.
`http://wiki.debian.org/HowToSetupADebianRepository`. [Online; visited
16.4.2010].

[8] WWW pages. Packaging:guidelines.
`https://fedoraproject.org/wiki/Packaging/Guidelines`. [Online; visited
23.4.2010].

[9] WWW pages. The spacewalk source code.
`http://git.fedoraproject.org/git/?p=spacewalk.git`. [Online; visited
22.2.2010].

[10] WWW pages. Spacewalk wiki. `https://fedorahosted.org/spacewalk/wiki`.
[Online; visited 22.2.2010].

[11] WWW pages. Spacewalk wiki architecture.
`https://fedorahosted.org/spacewalk/wiki/Architecture`. [Online; visited
22.2.2010].

[12] WWW pages. Writing yum plugins.
`http://yum.baseurl.org/wiki/WritingYumPlugins`. [Online; visited 19.4.2010].

[13] Red Hat, Inc. Red hat network satellite 5.3.0 - installation guide.
`http://www.redhat.com/docs/en-US/Red_Hat_Network_Satellite/5.3/`
`Installation_Guide/html/index.html`. [Online; visited 15.4.2010].

[14] Red Hat, Inc. Red hat network satellite 5.3.0 - reference guide.
`http://www.redhat.com/docs/en-US/Red_Hat_Network_Satellite/5.3/`
`Reference_Guide/html/index.html`. [Online; visited 15.4.2010].

[15] Red Hat, Inc. Spacewalk. `http://redhat.com/spacewalk/`. [Online; visited
22.2.2010].

[16] Red Hat, Inc. Spacewalk - frequently asked questions.
`http://www.redhat.com/spacewalk/faq.html`. [Online; visited 15.4.2010].

[17] Josip Rodin, Osamu Aoki, Craig Small, Raphael Hertzog, Jaldhar Vyas, and Will
Lowe. Debian new maintainers' guide. `http://www.debian.org/doc/maint-guide/`.
[Online; visited 22.2.2010].

[18] Wikipedia. Advanced packaging tool.
`http://en.wikipedia.org/wiki/Advanced_Packaging_Tool`. [Online; visited
19.4.2010].

[19] Wikipedia. ar (unix). `http://en.wikipedia.org/wiki/Ar_(Unix)`. [Online; visited
12.4.2010].

[20] Wikipedia. deb (file format). `http://en.wikipedia.org/wiki/Deb_(file_format)`.
[Online; visited 22.2.2010].

[21] Wikipedia. ebuild. `http://en.wikipedia.org/wiki/Ebuild`. [Online; visited
19.4.2010].

[22] Wikipedia. Package management system.
`http://en.wikipedia.org/wiki/Package_management_system`. [Online; visited
22.2.2010].

[23] Wikipedia. Yellowdog updater, modified.
`http://en.wikipedia.org/wiki/Yellowdog_Updater,_Modified`. [Online; visited
19.4.2010].

[24] Michal Čihař. Seriál: Balíčky pro debian.
`http://www.abclinuxu.cz/serialy/balicky-pro-debian`. [Online; visited
22.2.2010].

Appendix A

Content of CD

- packages
    - DEB
    - RPM
- packages_source
    - DEB
    - RPM
- patches_for_Spacewalk
- thesis_source_latex

Appendix B

Screenshots

Figure B.1: The system overview

Figure B.2: The system details

Figure B.3: The list of installed packages

Figure B.4: The configuration file creation

Figure B.5: Sandbox

Figure B.6: Deploying of file

Figure B.7: The scheduling of deployment

Figure B.8: Creating and scheduling of remote command

Figure B.9: Selecting the package for removing

Figure B.10: The package was succesfully removed and it is not in the list

Figure B.11: The list of packages in the software channels

Figure B.12: The package details