Config Management Proposal

Nicolas Lara

July 1, 2007

1 Motivation

Config management is a very common task, it can also be a repetitive tedious task (doing lots of changes that could be automated by hand) or a complicated task (especially for non-linux-experts). That is why package managers should take care of it.

A good configuration management system should follow some basic rules:

- It should be not intrusive. There shouldn't be any un-authorized change in configurations that could break the system. If there are non-trivial changes the config management system should either prompt the user or provide a way to go back to the previous configuration.
- It should allow to have both mandatory changes and optional changes. Mandatory changes are those that are needed for the application to work correctly. Optional changes are those that depend on the user preference and can be left for the user to decide.
- It should be easy to use for packagers. No intensive programming should be needed to modify configuration files, only the knowledge of what needs to be changes, added, deleted or left un-touched
- It should be general purpose.
- It should be optional. Allowing packages to be installed without config management.

This proposal for configuration files management is divided in three sections: (i) version control for configuration files, (ii) configuration files generation/modification/migration (the packager's perspective) and (iii) backend and frontend description (the programmer's perspective and the user's perspective)

2 Version Control

As I said in the motivation, the system should be not intrusive. I believe prompting the user for every change is both annoying (like Windows Vista) and unnecessary. To avoid prompting the user for changes every time, it is necessary to have some type of version control so when either the configuration management system or the user make a mistake in a config file it can be undone by a simple "revert" call.

I propose to use subversion with a local repository as the versioning system because it is both powerful and easy to manage. For security and user-friendliness I propose to wrap the usual commit, log, add and revert methods (among others) and provide a configuration version control program with both CLI and GUI.

As a downside of using subversion (or other versioning control system) to manage /etc one could cite the new need to add files to the repository or commit changes after modification. To solve this problem I propose integrating the inotify module to the default kernel so the kernel would "notify" our versioning wrapper of the changes on the directory and a allow it to commit the changes. In the case where the user modify the files by hand, comments might be left at the begining of the file enclosed but some still-to-define syntax in order to allow our config management system to read the comment, use it as a commit message and delete it from the file.

Another possible problem is the amount of disk space the repository will take. To solve this problem our new config version control system could allow the users to clean the older revisions when a configuration is known to be stable or when the disk space is running out.

This part of the proposal could be implemented independently from the others and haves many advantages by itself.

References

- inotify: `http://en.wikipedia.org/wiki/Inotify`
- inotify: `http://edoceo.com/creo/inotify/`
- pyinotify: `http://pyinotify.sourceforge.net/`

3 Config API

To allow packages to contain config migration information there should be a config migration API. I believe that simply using python to modify configuration files is not enough because: (i) it is not generic, there are many ways to read/add/delete/change files in python; (ii) it's messy, every packager can use the language as they please creating long hard-to-maintain code which could lead to complications for the next release of the package; (iii) it's repetitive, for every change in configs packagers would need to deal with the same simple problems (i.e.: finding the conflicting line and removing it without touching the rest of the file, changing some syntax into another, ...) all over again; (iv) it's hard(er), many changes should be as simple as stating what the change is.

For this reasons I believe there is the need of an API to migrate files. The use of this API should be optional and packages could simply leave configuration to the user or config scripts (every package is different).

The config migration system should implement the following functions:

find

find(regexp, file)

Returns a list of dictionaries containing the position in wich the regular expresion "regexp" was recognized and the strings to wich named variables were unified. For example:

```
>>> l = find(r'DAY = (<day>\w+);', file)
>>> l
[{'position': 2, 'day': 'Monday'}, {position': 17, 'day': 'Thursday'}]
>>> l = find(r'DAY = (<day>\d+);', file)
>>> l
[{'position': 3, 'day': 2}, {position': 18, 'day': 5}]
```

add_line

add_line(line, file, position=False)

Appends the string "line" to the file "file". If position is not false the line is added as the $position^{th}$ line in the file.

del_line

del_line(regexp, file)

Removes the lines matching the regular expression "regexp" from the file "file". Returns a list containing the lines that have been removed (and their positions?).

change_line

change_line(regexp, str, file)

Changes the lines matching the regular expression "regexp" from the file "file" for the string "str". This string might contain the named variables from the regular expression.

change_line_into_file

change_line_into_file(regexp, str, file1, file2, remove=True)

Changes the lines matching the regular expression "regexp" from the file "file1" for the string "str" in the file "file2". This string might contain the named variables from the regular expression. If remove is not True the line is left in the original file, otherwise it is removed.

parse_separated_values

parse_separated_values(regexp, separator, file1)

Returns a list of the values separated by the string "separator" in the file "file" that match the regular expression "regexp"

```
>>> l = parse_separated_values("USE=\"(P<sep>\w+)\"", ", ", file)
>>> l
["apache", "cups", "alsa"]
```

modify_separated_values

modify_separated_values(regexp, separator, file1, function)

Modifies the values separated by the string “separator” in the file “file” that match the regular expression “regexp” according to the function “function”

```
>>> modify_separated_values("USE=\"(P<sep>\w+)\"", ", ", file, upper)
```

change_separator

change_separator(regexp, separator, file1, new_separator)

Modifies the values separated by the string “separator” in the file “file” that match the regular expression “regexp” changing “separator” for “new_separator”

```
>>> change_separator("USE=\"(P<sep>\w+)\"", ", ", file, "|")
```

This are the main functions that I consider are needed in order to simply modify configuration files. As I might be missing something I propose the addition of an extra (advanced) function to allow the packagers to create their own context free grammar and modify parts of the matching. This function is more complicated and might take an article of its own to design. That is why I leave it to be added later.

The last function to be implemented is the function that will allow the changes to be executed.

provide

provide(config_function, type='mandatory')

This is the core function that will allow the changes to be executed. The objective of this function is to either call the config_function and execute the changes or insert the function into the config-pending database.

```
>>> def execute_on_startup():
... add_line("/usr/sbin/some_program 1>&2", "/etc/conf.d/local.start")
...
>>> def make_fullscreen():
... """
... Selecting this option will make the program execute
... in fullscreen mode
... """
... add_line("FULLSCREEN = True", "/etc/some_config_file")
...
>>> provide(execute_on_startup)
>>> provide(make_fullscreen, type='optional-noargs')
```

4 Perspectives

4.1 Modules / Interaction

The backend for the config system is proposed to work as follows:

- **iNotify Module**
  This is the module that will monitor the changes in the files. It should be started as a service and continue to work while the system is up. The iNotify module will recive a system call whenever a file is created, modified or deleted. When this happens it should read the file to detect if a comment was added at the beginning to remove it and invoke the subversion module to commit the changes with the comment as a log message.

- **Subversion Manager Module**
  This module should work with the python bindings for subversion. It is the one in charge of the local svn repository in which the changes of the configuration file will be stored.

- **Config Management Module**
  This is the core module of the system. This module is the one that will provide the functions to be used in the packages. The functions provided to the packagers should behave as stated in the previous section.

- **Pending-Config Module**
  This module consists mainly of a database that will store the configurations that are left for the user and the current status of his/her decisions. I propose to implement it using Berkley DB and its python bindings. When the user makes a choice about the configuration the module should call the function that came with the package using the users decision as a parameter for the function (in the case it corresponds).

- **Interface Module**
  The interface module should take care of the CLI and GUI and allow the user to control the different aspects of config management as adding and

removing version control from files and directories, start/stop watching files for changes, change the current version and take decisions on optional or user dependent configurations.

Finally it will not be necessary to modify PiSi in any way since the management could be done in comar's post-install method just by calling the config management module.

4.2 User Intarface

From the end-users perspective there should be nearly no interface since the process should be mostly transparent for them. Even though I propose to allow the user to configure the packages (for non-mandatory, non-trivial changes). Some config options depend solely on the users opinion and needs. This changes should be left for the user to decide. The user interface for this changes should be simply a list of messages explaining the feature to be added and a selection option for the user. The user should be informed by the program when there are decisions to be made by him. The backend for user choice should be managed by the pending-config module and the code provided by the package. The user interface should also allow the user to clean the version repository when a stable config state is reached and to switch between old, new or parallel configurations.