FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Konfigurationsmanagement mit Libelektra

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering - Internet Computing

eingereicht von

BSc. Bernhard Denner
Matrikelnummer 0626746

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dipl.-Ing. Dr.techn. Franz Puntigam
Mitwirkung: Univ.Ass.Dipl.-Ing. Dr.techn. Markus Raab

Wien, 16. Jänner 2018

Bernhard Denner	Franz Puntigam

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Configuration Management with Libelektra

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering - Internet Computing

by

BSc. Bernhard Denner
Registration Number 0626746

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.Dipl.-Ing. Dr.techn. Franz Puntigam
Assistance: Univ.Ass.Dipl.-Ing. Dr.techn. Markus Raab

Vienna, 16th January, 2018

Bernhard Denner	Franz Puntigam

Erklärung zur Verfassung der Arbeit

BSc. Bernhard Denner
Patzenthal 7, 2153 Stronsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. Jänner 2018

Bernhard Denner

Acknowledgements

I would like to express my very great appreciation to my main supervisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam and my second supervisor Dipl.-Ing. Dr.techn. Markus Raab for guiding me through this journey of writing and conducting this thesis and being very responsive to any of my questions. Without them this work would not have been possible.

I am particularly grateful for the assistance given by Dr. Stefan Hanenberg from University of Duisburg-Essen for the valuable discussions and guide lines while designing and conducting the user study and during analysis of the user study results.

I would like to offer my special thanks to all participants of my user study: Andreas Klebinger, David Fischak, Florian Nuding, Kurt Micheli, Lukas Naske, Anton Hößl, Andrey Lalin, Diana Vysoká, Marlene Hartmann, René Schwaiger, Sebastian Bachmann, Jakob Kogler, Armin Wurzinger, Christopher Cadek, Kurt Micheli and Sissi Wang. I would particularly like to thank Kurt Micheli for preparing the lab environment. In addition, I would like to thank Amina Hasimbegovic and Simon Treiber for assisting in organizing the user study sessions.

I received generous support from David Schmitt, who also conducted the user study and gave me very valuable feedback on my Puppet module.

I thank the Libelektra community for valuable feedback and support, especially Armin Wurzinger for writing the "`xerces`" XML storage plugin.

Last but not least, I would like to express the deepest appreciation to my wife Birgit for her support and patience.

Kurzfassung

Durch die ständig zunehmende Anzahl von Computersystemen, werden Konfigurationsmanagementwerkzeuge bei Systemadministratoren immer beliebter. Diese Werkzeuge erlauben Administratoren das Beschreiben von Computersystemen auf einem abstrakten Niveau, wobei das Konfigurationsmanagementwerkzeug diesen beschriebenen Zustand auf dem Zielsystem umsetzt. Üblicherweise verwalten diese Werkzeuge Konfigurationseinstellungen auf dem Niveau von ganzen Konfigurationsdateien, nicht auf der Ebene einzelner Konfigurationseinstellungen. Ein Problem dieser Vorgangsweise ist, dass bereits bestehende Standardkonfigurationen überschrieben werden. Weiters müssen Administratoren selbst sicherstellen, dass die Konfigurationsdateien syntaktisch korrekt sind. Wir haben in dieser Arbeit gezeigt, dass Konfigurationsmanagmentwerkzeuge durch die Verwendung von generischen Konfigurationsbibliotheken verbessert werden können. Dafür haben wir das Konfigurationsmanagementwerkzeug Puppet um die Funktionalitäten der Konfigurationsbibliothek Libelektra erweitert, wodurch Konfigurationänderungen auf der Ebene von einzelnen Konfigurationseinstellungen beschrieben werden können. Um zu zeigen, dass dieser Ansatz einem Systemadministrator einen messbaren Vorteil bringt, haben wir unsere Lösung in drei Schritten evaluiert. Zuerst haben wir das Laufzeitverhalten und die Robustheit der Syntaxvalidierung unserer Methode mit anderen Verfahren zur Konfigurationsdateienmanipulation von Puppet verglichen. In einem zweiten Evaluierungsschritt haben wir eine Fallstudie mit unserer Lösung durchgeführt, um zu zeigen, dass unser Ansatz auch für reale Einsatzzwecke verwendbar ist. Dabei haben wir unsere Erweiterung zum Konfigurieren eines automatischen Integrationssystems, bestehend aus 5 Computern und Konfigurationsdateien in 8 verschiedenen Formaten, verwendet. Um zu demonstrieren, dass unsere Lösung eine Produktivitätssteigerung beim Entwickeln von Puppetcode bewirken kann, haben wir in einem dritten Evaluierungsschritt eine Nutzerstudie durchgeführt. Dabei hatten 14 Teilnehmer vier Puppetprogrammieraufgaben in bis zu drei Varianten zu lösen, wobei wir die Durchführungszeiten der einzelnen Varianten miteinander verglichen. Mit der Fallstudie konnten wir zeigen, dass unser Ansatz auch für reale Einsatzzwecke verwendet werden kann. Weiters konnten wir mit der Nutzerstudie zeigen, dass unsere Lösung im Vergleich zu anderen generischen Verfahren zur Konfigurationsdateienmanipulation einen signifikanten Produktivitätszuwachs bringt.

Abstract

With the ever growing number of computer systems, configuration management tools are getting more popular. When using a configuration management tool, a system administrator describes the managed computer system on an abstract level and the configuration management tool configures the target system accordingly. Configuration management tools usually manage configuration settings on the level of configuration files and not on the level of individual configuration settings. One of the problems of this approach is that already preconfigured default configuration settings will be overwritten. Further, system administrators have to ensure a correct configuration file syntax themselves. In this thesis we have shown, that these weaknesses of configuration management tools can be improved by utilizing the methods of a general purpose configuration framework such as Libelektra. We implemented an extension for the configuration management tool Puppet, which allows us to treat configuration settings as first class citizens. In order to show that our approach has measurable advantages over existing Puppet concepts, we first compared our solution with other Puppet configuration file manipulation methods in terms of robustness of syntax validation and runtime performance. Second, we have conducted a case study to demonstrate that our solution is ready for real-world usage by managing a continuous integration system consisting of 5 computers and configuration files in 8 different formats. In a third evaluation step we have carried out a user study to show that our solution is able to improve the Puppet code development productivity. We asked a group of 14 participants to solve four different Puppet programming tasks in up to three different variants and compared the times required to solve the tasks. The case study demonstrated that our solution is ready for real-world scenarios and the results of the user study showed that our solution increases the development productivity significantly compared to other general purpose configuration manipulation strategies.

Contents

CHAPTER 1

Introduction

With the advent of cloud computing and virtualisation, the number of computer systems a system administrator has to handle increased dramatically. Therefore, system administration is becoming an increasingly complex task. As one can imagine installing software and managing the configuration of hundreds of systems proves to be a challenging task [NIB11; Tan+15; OGP03]. Administrators have to react on changes to the environment, fix security issues and keep software up-to-date. Doing this manually, machine by machine is almost impossible [Spi12].

Therefore, configuration management (CM) tools have been developed to assist administrators in managing their systems. Popular CM-tools in these days are *CFEngine* [CFE16a], *Chef* [Che16b], *Ansible* [Red16] and *Puppet* [Lab16e]. Despite their differences, they all have one common approach. An administrator defines a desired system state in form of text files. Such states can be expressed as software packages to be installed, configuration files with defined content or system services to be running. This desired system state is applied on as many computer systems as required, in a reproducible way.

One of the central tasks of configuration management is the manipulation of configuration files. Because of different requirements, many CM-tools offer different methods to perform changes, often with different approaches such as partially modifying configuration files or defining the complete content at once. Each method has something to commend it, being simplicity and ease of use or having complete control.

Plain text files are used as configuration store by most of the software tools on UNIX and Linux systems. Over time, different formats for such configuration files have evolved and often it seems that each tool uses its own syntax for its files [ŚB18]. Currently, it is common to manage the complete content of configuration files, with the help of configuration management tools.

However, despite its simplicity, this method has some drawbacks. Since many software packages install configuration files including reasonable default settings, these settings

have to be transferred to the management tool in some form. Otherwise, these default settings would be overwritten by the manage-complete-content approach of CM-tools. When managing a diverse infrastructure with different OS distributions, the problem becomes even worse. Each operating system comes with partially different default settings, which may have to be preserved.

A second issue comes with missing validation. Illegal configuration parameter settings that violate some syntax rules are common causes for misconfigured systems [Yin+11; KUC08]. Since the CM-tool is not aware of the syntax the configuration file uses, it does not detect any syntactical errors. As a consequence, this problem is detected at a very late stage. Usually the system administrator will discover these problems at the managed system itself. At this point in time there might already have been service outages.

In this thesis we will improve on these two problems, by utilizing the open-source library *Libelektra*. Libelektra is a general purpose configuration library for all kind of applications. It manages configuration settings within one hierarchically structured configuration space [Raa10; Raa16a]. Libelektra allows us to map different configuration files with different formats in its own configuration space. This approach enables an administrator to manage a huge variation of configuration files accessible by a single key-value database.

Our hypothesis is that combining CM-tools with Libelektra provides a measurable advantage for system administrators. Desired system configurations can be specified in form of key-value pairs, instead of defining the concrete content of configuration files. This also enables configuration management on a per-setting granularity, whereas default values do not have to be touched at all. Additionally, system administrators do not have to care about the syntax of configuration files, as the syntactic rules are automatically ensured by appropriate modules of Libelektra.

1.1 Aim of this Thesis

As introduced above, this thesis will tackle problems introduced by configuration file handling with CM-tools. The principal part of this work compares different approaches of configuration file handling with the CM-tool Puppet, to prospect the pros and cons for different usage scenarios of each method. In contrast to the existing methods, an appropriate extension module for the CM-tool Puppet was developed, which utilizes the mechanisms of Libelektra and its extensions. This allows us to directly compare our proposed solution with already existing configuration file handling solutions.

Puppet is a popular CM-tool in these days. It was chosen, since the declarative approach of Puppet for defining system aspects is very well suited for specifying system configuration settings in the form of key-value pairs. The developed Puppet module shall enable an administrator to cover all sorts of configuration file handling, by defining the required settings in form of key-value pairs instead specifying the configuration file content itself. This concept requires a deep integration with the features of Libelektra, such as key-value

manipulations, key meta-data handling and managing the Libelektra configuration space itself.

When designing the Libelektra Puppet module, a strong focus was put on API (Application Programming Interface) usability. An administrator shall be able to define the desired system configuration with simple but powerful concepts, while offering the full flexibility of Libelektra.

The aim of this thesis can be summarized by the following two research questions:

Research Question 1. *When using the proposed solution in a real-world scenario, is it possible to manage all required configuration files with the methods of the proposed solution? Are there any trade-offs necessary?*

Research Question 2. *Which approach of configuration file manipulation provides the best results if compared in terms of usability and maintainability? Does the proposed solution have an advantage over existing methods?*

1.2 Methodological Approach

In order to reach the desired goal of this work, we explored two competing general purpose configuration frameworks. The most suitable was chosen for our proposed solution of a key-value based configuration file manipulation method for Puppet. Before the actual implementation of the proposed solution could be started, we had to develop Ruby bindings for the Libelektra library.

To show that our approach solves the described problems of configuration file handling, the method proposed by this thesis was evaluated against existing Puppet concepts for defining configurations settings.

The first evaluation step compares the proposed solution with existing Puppet file manipulation strategies in terms of robustness of syntax validation and runtime performance.

The second evaluation step aims to show, if the proposed solution is usable in a real-world scenario. This step is guided by the Research Question 1.

The last evaluation step compares the proposed solution with existing Puppet file manipulation methods in terms of usability and maintainability. This was done by conducting a user study which tries to answer Research Question 2.

1.3 Structure of this Thesis

Chapter 2 gives the reader some background knowledge, required to understand the topics and concepts of this thesis. This chapter covers required concepts of the CM-tool Puppet and Libelektra. Further, we describe existing file manipulation methods of Puppet.

Chapter 3 describes why we have decided to use Libelektra for the proposed solution. Additionally, we present the architecture of our solution and how its components fit together.

Implementation details of our proposed solution are described in Chapter 4.

The first evaluation step, the feature comparison, is presented by Chapter 5.

A detailed description of the second evaluation step can be found in Chapter 6. At the beginning of this chapter the used real-world scenario system is described. Afterwards we present relevant implementation details of the case study.

The methodological approach and the results for the third evaluation step, the user study is presented in Chapter 7. First we give some details on the experiment design and the experiment setup. Afterwards, we present collected information about the user study participants together with all measured experiment results. These results are further analysed with statistical methods to show if there are significant differences between the used Puppet file manipulation methods. Finally we conclude this chapter by answering our research question for this evaluation step.

Finally, this thesis is concluded by Chapter 8.

CHAPTER 2

Background

2.1 Configuration Management Tools

Configuration management tools are being used by system administrators for a long time, however it is sometimes stated to be one of the most controversial aspects of system administration [BC06].

One of the first theoretical work on configuration management tools was done by Burgess. In his work [Bur95], he introduces *CFEngine* [CFE16a]. The aim of this tool is to provide the user a central management instrument for different types of computer systems within a network. *CFEngine* enables its users to define many aspects of UNIX based systems in one central file in a high-level manner by a newly introduced configuration language. This configuration language is used as a form of abstraction mechanism, which hides variations between different operating systems and allows the user to think in a more abstract way. Desired properties of managed systems are defined in a declarative manner, as well as relationships and dependences between these properties. *CFEngine* uses the concept of bundles and classes to group system aspects and provide a form of modularization. A *CFEngine* configuration program is then used on the target systems to automatically enforce the defined aspects and properties. In Burgess later works [Bur03] and [BC06], he further extended his theoretical concepts for *CFEngine*, which later got integrated in newer versions.

Puppet [Lab16e], grew out of dissatisfaction with *CFEngine* [Tsa10], however some of the primary concepts of these two tools are very similar. Puppet also allows the user to define desired system properties in a central repository. Therefore, its own declarative domain specific language (DSL) is used to model system properties (called *resources*), order-relations and dependencies between these resources. Similar to *CFEngine promises*, *resources* are the main abstraction mechanism in Puppet, hiding differences between operating systems. Multiple *resource* declaration can be grouped by classes and modules,

which helps to organize Puppet source code, written in Puppets own DSL. An *agent* running on the target system, called *managed node*, ensures all defined resources are in the desired state, otherwise certain actions will be initiated to enforce this state, e.g. installing a package, creating user accounts or updating the content of files [Kru+13; Tur07].

Beside *CFEngine* and *Puppet* a bunch of other CM-tools have emerged in the industry, such as *Chef* [Che16b], *Ansible* [Red16] or *Salt* [Sal16], to name just a few.

2.2 CM-Tool Puppet

Puppet is a Ruby-based CM-tool and can be used to manage configuration on UNIX, Linux and Microsoft Windows platforms. Puppet can be used either in standalone or in client-server mode and allows us to use it for very simple configuration management tasks but also for large and complex infrastructures [Kru+13].

2.2.1 Resources and Types

As already stated above, the central element in Puppet for defining certain aspects of a system, is the *resource*. Each individual characteristic of a computer system is treated as a resource, e.g. each single file, each package, user, directory but also parts of files or settings of configuration files. Puppet distinguishes between *managed* and *unmanaged* resources. Managed resources are described in the user provided Puppet code. These resources are managed by Puppet and the desired states (file permissions or content, package state ...) are enforced. All other resources not defined by the Puppet code are called unmanaged resources and are ignored by Puppet [Kru+13; Tur07].

Each resource is clearly defined by its *resource type*, such as "`file`", "`package`", "`user`", a type-unique resource identifier, such as file or package name, and a variable amount of properties, which define the desired resource state. An example is shown by Listing 2.1, which includes two resource declarations. The first declaration uses the resource type "`file`" with the unique resource ID "`/etc/resolv.conf`" and defines five parameter. Therefore, this resource declaration describes a desired state for the file "`/etc/resolv.conf`" with the specified properties. The second resource declaration is of type "`package`" with the unique resource ID "`firefox`". This resource declaration instructs the Puppet agent to ensure the package "`firefox`" is installed.

Resource types are implemented in Ruby and form the primary abstraction mechanism in Puppet, to hide differences between different operating systems. Thus, a Puppet user will always install packages with the resource type "`package`", being on Ubuntu, Fedora, Solaris, MacOS or even Windows. Puppet comes with many built-in resource types to manage the many important aspects of computer systems. More specialized resource types can be implemented as part of a new Puppet module. Many of such specialized modules are publicly available on Puppet-Forge [Lab16d], a collaboration and sharing platform for Puppet modules.

```
# manage a single file
file { "/etc/resolv.conf":
  ensure  => "file",
  owner   => "root",
  group   => "root",
  mode    => "0644",
  content => "nameserver 8.8.8.8"
}

# install a package
package { "firefox":
  ensure  => "installed"
}
```

Listing 2.1: Resource declaration examples

2.2.2 Classes, Node Definitions and Defined Types

Another central language construct in Puppets DSL are *classes*. Classes are used to group resource declarations and help to modularize Puppet code. Classes can also have parameters, which allows us to separate code and data (the concrete configuration values) [Lab16c]. Listing 2.2 show the class "`dns`" with a parameter "`searchlist`" and a parameter "`nameserver`" with a default value. The content of the managed file is derived by the two parameter, as both are used within the string passed to the attribute "`content`".

```
class dns(
  $searchlist,
  $nameserver = "8.8.8.8"
) {
  file { "/etc/resolv.conf":
    content => "search ${searchlist}\nnameserver ${nameserver}"
  }
}
```

Listing 2.2: Example class

Classes should not be mixed with the class concept known by object oriented languages. A Puppet class is not instantiated to an object, instead they are *declared* by other classes or node definitions. This can be done in two ways [Lab16c]:

- *resource-like* declaration: This declares a particular class and allows us to override its parameters. A resource-like declaration require that a given class is only declared once to avoid conflicting parameter values.

- *include-like* declaration: This declares a class without specifying parameter values. An include-like class declaration can be used several times.

Once a class is declared, Puppet uses the following methods to resolve all class parameter values in the given order [Lab16c]:

1. Use the overridden value from the class declaration, if declared by a resource-like declaration.
2. Request a value from an external data source, usually Hiera, a hierarchically organized key-value data store.
3. Use the default value.

If for one parameter no value can be found, Puppet will abort the compilation.

Node definitions can be seen as some sort of entry point. A node definition contains a set of class declarations or resource declarations a particular system should be managed with [Lab16c].

Defined types are blocks of Puppet code that can be evaluated several times with different parameter values. A defined type can be used in the same way as normal resource types. Therefore, a defined type always has an implicit parameter "`name`", which corresponds to the unique resource ID, specified within the resource declaration [Lab16c].

Beside these Puppet specific language constructs, the Puppet DSL contains very common elements, such as variables, conditional statements and function calls [Lab16c].

2.2.3 Puppet Source Code Structure

Puppet source code is usually organized in three main parts:

- A set of Puppet modules, either self written or community provided modules (see [Lab16d]).
- A site manifest, typically containing node definitions only. This is the main starting point for the Puppet compilation phase.
- Hiera data store, containing parameter values for classes.

This structure allows us to write *reusable* modules, which do not contain any concrete values, such as IP-addresses, domain names or passwords. Instead, this data can be clearly separated from the actual Puppet code and is located in the Hiera key-value store.

2.2.4 Operation Phases and Catalog

Puppet operates in two main phases when managing a particular node:

- *compile phase*: compile the Puppet source files together with data pulled from Hiera and node information to a *catalog*.
- *apply phase*: apply the compiled *catalog* on the managed node.

The Puppet client, also called Puppet *agent*, running on a particular node initiates a *configuration run* and sends a collection of node information (*facts*) to the server (Puppet *master*). The master starts the compile phase and compiles the site manifest and the sources from all modules together with all facts for this particular node to a *catalog* dedicated for the managed node. This catalog contains a collection of resources and associated parameter values that should be applied on the managed node. Once the agent receives the catalog, it starts the apply phase. During this phase the agent checks the current state for each resource within the catalog and if the actual state differs from the desired state, the agent applies actions on the managed node to enforce this desired state. The information which action has to be performed is implemented by each resource type. Build-in resource type implementations are contained in the Puppet agent itself, custom resource type implementations are provided by the Puppet modules [Kru+13].

Therefore, in a client-server setup, the two phases are executed on different systems. The compile phase is executed on the master system, the apply phase runs on the managed node system. In standalone mode, both phases are executed together on the managed node.

If the client-server setup is used, this architecture allows us to keep all Puppet source files in one place. The client system does not have access to all Puppet source files and Hiera data files. Instead the client receives only the compiled catalog, containing information relevant for this managed node only. This is an important aspect for big infrastructures since this reduces the management overhead.

2.3 Configuration Files

Software configuration values can be stored in different forms. Plain text files are very common for UNIX based applications, while on other platforms, such as Microsoft Windows OS family, often a central configuration database is used. Both forms have their advantages and disadvantages. Central configuration databases enable information querying and manipulation through a standardized API, while plain text files requires parsing its content based on a special syntax of the configuration format. Over time, different formats for such configuration files have evolved, and often it seems each tool uses its own syntax for its files. However, there are programming libraries available for a lot of the commonly used file formats, such as INI. Often, high-level programming languages ship such libraries already as part of their runtime core.

When manipulating plain text configuration files, especially with CM-tools in an automatic way, we have to consider certain aspects. We have to respect and observe the format and syntax rules of the configuration file, otherwise the configured software might refuse reading the file. When syncing configuration files between different computer systems, we might overwrite some system-individual settings like hostnames or IP-addresses. Often configuration file formats allow an administrator to add comments for documentation purposes, which can help during problem analysis. Automatically generated configuration files often miss such comments.

For the comparison of the individual configuration manipulation methods, we will focus on the following aspects:

- syntax awareness: is the correct syntax of the configuration file automatically ensured, or is it in the responsibility of the user?
- defaults preservation: is the method able to change only certain parts of the configuration file, in order to keep default settings untouched?
- expressibility: is the method able to utilize all aspects of the corresponding configuration file format, especially handling of comments?

2.4 Manipulating Configuration Files with Puppet

The following sections will describe different methods for manipulating and defining the content of configuration files.

2.4.1 Content Defined by Strings

The built-in resource type "`file`" offers the Puppet developer two parameters to define the content of a file: "`content`" and "`source`". The first one takes a String argument and, as the name suggests, directly defines the content of file. An example was already shown in Listing 2.1 and Listing 2.2.

An alternative to the classic string definition are *here documents* or *Heredocs*, as also known from the *shell* or *Perl* scripting languages. A Heredoc defines a string, which is allowed to span over multiple lines without special treatment of quoting character such as '`"`'. Heredocs are enclosed by a special Heredoc start tag '`@("CONTENT")`' and the corresponding end tag '`| CONTENT`', whereas the keyword (here "`CONTENT`") is freely selectable. As for normal string definitions, Heredocs also allow us using variables in their content. This makes the definition of a complex string, such as multi-line configuration files, directly in the Puppet source code possible. However, this is not a good programming practice, as it mixes code and data in the same source file.

The advantage of this file-content approach is its simplicity. However, extensive uses, especially together with Heredocs, lead to rapidly growing source files and system

administrators quickly loose the overview of their Puppet source code. Syntax awareness does not exist. It is possible do define any content, therefore this method has a very high expressibility. However, managing only portions of configuration files is not possible either, as this is an all-or-nothing approach.

2.4.2 Pull from Other Sources

As described above, the resource type "`file`" offers a second parameter for defining the content of files: "`source`". If this parameter is set to a suitable value, Puppet will pull the content for this file from the defined location, either by downloading from a Web server, Puppets own built-in file transfer mechanism or from a locally available file. However, this file-source method does not support any dynamic content definition. The content is treated as-is. Therefore, it is not adequate for configuration file manipulation.

2.4.3 Content Generated by Templates

Beside resource types and classes, Puppets DSL also includes the concept of *functions*, as known from typical procedural languages. Functions have a unique name, can take an arbitrary number of arguments and one return value as result. In Puppet they are often used for type checking, type conversions and string manipulations.

Puppet has two built-in functions, which are quite useful for defining contents of files: "`template()`" and "`inline_template()`". Both functions enable rendering of strings based on ERB templates. ERB (Embedded RuBy) is a special feature of Ruby to embed Ruby code in text files. This embedded code is evaluated during template parsing, which defines the result in a dynamic way. The function "`template()`" takes a file name as string argument, defining the path to the used template file, whereas the function "`inline_template()`" directly takes the template definition itself as string argument. Both functions return the rendered template result, which can be directly used by the "`file`" resource type, for example. Listing 2.3 shows an example ERB template to define the content of the UNIX resolver configuration file.

```
<% if @dns_search != '' %>
search <%= @dns_search %>
<% end %>
<% @dns_servers.each do |server| %>
nameserver <%= server %>
<% end %>
```

Listing 2.3: Example ERB template

The corresponding use of such a template within a "`file`" resource declaration is shown in Listing 2.4.

As one can see in Listing 2.3, Ruby code fragments are enclosed by the special tags '`<%`' and '`%>`'. Everything else is treated as normal text. Similar concepts can be found for

```
class dns($dns_search, $dns_servers) {
  file { "/etc/resolv.conf":
    ensure  => "file",
    content => template("path/to/template")
  }
}
```

Listing 2.4: Example ERB template use

example in PHP. Puppet allows us to use its local class variables (here "`$dns_search`" and "`$dns_servers`") within the template as Ruby instance variables. Listing 2.3 demonstrates simple uses of conditions ("`if...`") and loops ("`each do...`").

Beside the Ruby-ERB template engine, starting with version 3.8, Puppet includes its own template engine, called *EPP*. It brings a better integration with the Puppet DSL. The corresponding function "`epp()`" and "`epp_inline()`" are equivalent to their ERB counterparts, whereas the EPP (embedded Puppet) template engine is used.

Defining content of configuration files by ERB/EPP templates is powerful. It enables variable substitution, conditions, loops and much more, as it is based on Ruby code execution for generating the resulting file content. Since it is possible to define any form of text content, the resulting output is not tied to any special format or syntax. Therefore, the expressibility is very high. However, syntax validation of special configuration formats is not in the scope of this method. Therefore, it is quite easy for Puppet developers to generate configuration files with an invalid syntax.

This form of content definition is primary driven by the template functions, which have to be used together with a Puppet resource type to 'transfer' the result to the target configuration file. As already said, the resource type "`file`" is often used for this purpose. However, as we have explained in Section 2.4.1, the "`file`" resource type is not able to manage portions of a file. Therefore its not possible to preserve default values using this method.

2.4.4 Line Based Manipulation

Until now, only methods for defining the complete contents of files were discussed. When it comes to partial modifications of configuration files, the resource type "`file_line`" from the Puppet module *puppetlabs-stdlib* is a suitable candidate. This resource type facilitates line based modifications of files based on regular expressions, similar to the UNIX tool *sed*.

Listing 2.5 shows an example resource declaration of the "`file_line`" resource type. The desired value defined by parameter "`line`" will replace the line matched by the regular expression of parameter "`match`". If a matching line is found, this line will be replaced by the defined value, otherwise a line with the desired value will be added. This

```
file_line { "/etc/resolv.conf_searchlist":
  ensure => 'present',
  path   => "/etc/resolv.conf",
  line   => "search $dns_search",
  match  => "^search .*",
}
```

Listing 2.5: "file_line" example

makes this resource type suitable for modifying configuration settings in configuration files, as it can be used to replace single configuration values, while ensuring settings are not added twice. Therefore, this method supports partial modifications of configuration files and enables its users to preserve default settings supplied by the OS distributions.

In terms of expressibility, the "file_line" resource type is very flexible, since it allows us to define any content. However, when it comes to modifying multiple lines at once, for example to add comments to certain settings, this method is not suitable, since the "match" property will only match a single line, as the resource type name already suggests. Therefore, we will consider expressibility of this method to be limited. The "file_line" resource type does not care about the managed content itself. Therefore, syntax validation of the given content is not performed at all.

2.4.5 Format Specific Modules

A more advanced way for modifying configuration files are methods, that take care of the special format and syntax of configuration files. A prominent example of such a resource type is the "ini_setting" type, defined by the module *puppetlabs-inifile*. This module allows us modifying single settings in configuration files respecting the INI-file format.

```
ini_setting { "puppet-server":
  ensure  => 'present',
  path    => "/etc/puppet/puppet.conf",
  section => "main",
  setting => "server",
  value   => "example.com"
}
```

Listing 2.6: "ini_setting" example

Listing 2.6 shows an example application of the "ini_setting" resource type. This example will modify the value of the setting "server" within the section "main". If this configuration setting does not exist in the defined file, the setting will be added. One important thing to mention here is, that this resource type treats each setting within an INI-based file as a single resource instance. Therefore, it facilitates partial modifications of configuration files in simple way, while ensuring the correct syntax of the file.

Beside the "ini_setting" resource type, Puppet has built-in support for modifying important configuration files of UNIX based systems. The resource types "host", "user", "group" and "mailalias" are examples here. "host" manages entries in "/etc/hosts" database, "user" will ensure certain user accounts exists and therefore manages "/etc/passwd", "group" does the same for user groups and "mailalias" manages mail alias files. Each of these resource types, respects the format of its corresponding configuration files and treats each setting as a single resource instance. However, some resource types are able to manage more than just the configuration. For example, the "user" type can be used to manage the home directory of the desired user, too.

Additionally, Puppet has built-in resource types to generate configuration files for the monitoring tool *Nagios*, which is a well-established monitoring solution. The "nagios_*" ("nagios_host", "nagios_service", "nagios_command" ... resources types can be used to manage single entries for *Nagios* configuration files. Again, these resource types are treating each setting (Nagios object definition) as single resource instance and therefore are able to manage the corresponding configuration files partially.

Many application specific modules found in [Lab16d] facilitate partial modifications of the corresponding application configuration files, however they are limited to a specific application and will not be considered here.

As we have seen, each of the described methods automatically ensures the compliance of the underlying syntax of the configuration file. Additionally, each method manages files partially, through modifying dedicated entries or regions. Therefore, it it possible to preserve default values. In terms of expressibility we will consider this method also as limited, as none of these modules is able to add comments to the corresponding settings. However, this is just limited by the underlying resource type implementation.

2.4.6 Frontends to Central Configuration Backends

The configuration definition strategies defined here are not modifying the underlying configuration files directly, instead they use tools to manage a central configuration database. Prominent resource types for doing this, are the "registry_key" and "registry_value" resource types found in the module *puppetlabs-registry*. These two resource types, enable manipulations of the Microsoft Windows registry configuration database.

Another example for such a strategy is the "gconf" resource type, defined by the *rohlfs-gconf* module, which will manage configuration settings for the *Gconf* configuration database, used by the *GNOME* platform.

Both methods facilitate partial modifications of configuration settings, which leaves other values untouched. Syntax validation is automatically ensured by the underlying configuration frontend tools used by these resource types. However, expressibility is again limited as none of the two tools supports adding comments to configuration settings, although this is mainly a limitation of the underlying configuration database and the used frontend tooling.

2.4.7 Augeas

The fact that many configuration file formats have emerged, which makes automatic modification of configuration files difficult, was the main motivation for the development of *Augeas* [Lut08]. Augeas is a library allowing us to retrieve and modify configuration values of different configuration files, which adhere to supported formats, by a standardized API.

Puppet has a built-in support of the Augeas API, and can be used with the "`augeas`" resource type. Listing 2.7 shows an example application of this resource type, modifying the file "`/etc/puppet/puppet.conf`". This example will set the configuration option "`server`" of section "`main`" to the value of the variable "`${server}`" and adds a comment line directly before the "`server`" configuration option.

```
augeas{ "puppet-server" :
  context => "/files/etc/puppet/puppet.conf/main",
  changes => [
    "set server ${server}",
    "set #comment[following-sibling::server]\
      [last()] 'central puppet server'"
  ]
}
```

Listing 2.7: Augeas example

This method is one of the most advanced strategies for automatic modification of configuration files. Augeas automatically ensures the correct syntax of configuration formats. Additionally, modifications are done on a per-configuration setting basis. Therefore, it allows the user to leave default values untouched. And further, through the *XQuery* like syntax for referencing sections around a specific configuration setting, it also facilitates modifications of surrounding regions, especially comment lines within a configuration file.

2.5 Libelektra

Elektra is an open-source initiative aimed to provide a general purpose configuration library, called *Libelektra*. The desired use case of this library is to provide applications a common configuration framework on a key-value basis in a platform independent way [Raa10].

2.5.1 Global Shared Key-Value Database

Libelektra operates on a global shared key-value database, also called key space, to store and retrieve key-value pairs. This means that all applications utilizing Libelektra access the same key-value database. This global key-value database enables configuration sharing across applications.

Libelektras' key-value database, in this thesis also called key space, is hierarchically organized. Keys are addressed by a unique name, which consists of multiple elements separated by slashes [Raa10]. This concept is comparable to file systems and absolute file names.

2.5.2 Namespaces and Cascading

Beside this hierarchical structure, the key space is split in several namespaces. Libelektra implements the following namespaces:

- "`system`": contains global system configuration.
- "`user`": contains configuration settings for a particular user account. Therefore, each user account has its own "`user`" namespace.
- "`dir`": allows us to store different configuration settings for different working directories.
- "`spec`": contains key specifications for other keys.
- "`proc`": for in-memory keys.

Each key name is prefixed with its namespace, therefore the following two keys in Listing 2.8 refere to different keys.

```
system/sw/myapp/config
user/sw/myapp/config
```

Listing 2.8: Example Libelektra key names

A powerful concept which is based on namespaces is *cascading*. Cascading allows us to retrieve key values from the most suitable namespace. This works by searching the requested key in each namespace in a defined order. If a value for the key is found the process is aborted and the requested key-value pair is returned. Cascading is initiated by requesting a key without namespace (key name starting with a "`/`"). Libelektra searches for a matching key within all namespaces in the following order: "`spec`"[1], "`proc`", "`dir`", "`user`", "`system`".

This namespace order has an important aspect: keys within the "`user`" namespace can override keys in the "`system`" namespace. A key within the "`dir`" namespace may override the same key within the "`user`" namespace and so on. This means, that an administrator or package maintainer can provide a default configuration within the "`system`" namespace, which is read-only by normal users. However, a user has the possibility to override this default configuration by adding the same key with a different value into its own "`user`" namespace.

[1] only for metadata, not for values

2.5.3 Modular Architecture and Mounting

Per default, Libelektra stores all keys for one key database namespace within one configuration file. However, sometimes users want to have more control over their configuration files. Therefore, Libelektra allows us to integrate different configuration files of different formats into the global key space. The process of integrating a configuration file is called *mounting* and can be compared to the concept of mounting file systems. This means that we are able to *mount* a specific configuration file on a given *mount point* within the Libelektra key database hierarchy. As a consequence of this, every key below this mount point will be stored within the mounted configuration file, instead of the default configuration file.

Mounting can also be used to integrate existing configuration files into Libelektra, which allows us to read and manipulate the configuration settings within this file with the Libelektra API and tools.

Libelektra's core is realized by a modular architecture and can be extended by different plugins. Plugins implement the actual work and are of different types. For example, *storage plugins* read configuration settings from disk and transform it to key-value pairs and vice versa. *Checker plugins* can verify values and structure of keys and issue an error message if a problem was found.

Multiple plugins are combined to a *backend*, whereas one particular backend is used for one particular mount point. Therefore, if we want to mount a configuration file we have to specify a list of plugins which should be used for mounting the desired configuration file.

2.5.4 Specifications

As already indicated in the previous section, Libelektra is able to perform value and structure validation. Therefore, Libelektra allows us to define *specifications* for each key [Raa16b]. Specifications describe certain aspects of a key such as used data type, allowed values or restrictions to the name of the key. These specifications are stored within a special key namespace, the "`spec`" namespace.

Each key in the "`spec`" namespace defines the specification for a corresponding key within other namespaces. The specification information is stored in form of metadata (key-value pairs for a particular key). The "`spec`" plugin is responsible to transfer this specification metadata to the corresponding keys of the other namespaces. Once this is done, all other plugins, such as checker or filter plugins in the backend can use this specification metadata. This process allows Libelektra to clearly separate key specifications and actual values.

2.5.5 Arrays

Libelektra does not support complex data types, such as arrays of values. However, many configurations contain such value lists. In order to implement such behaviour, Libelektra

uses a sub-key approach to store an ordered list of values. Therefore, each list element is stored within a sub-key of the actual target key. Listing 2.9 shows an example of this approach.

```
system/sw/myapp/config/#0 = zero
system/sw/myapp/config/#1 = one
system/sw/myapp/config/#2 = two
...
system/sw/myapp/config/#_10 = ten
```

Listing 2.9: Libelektra key holding an array of values

Listing 2.9 shows an Libelektra key with 11 elements, starting from element "`#0`" to "`#_10`". Libelektras' internal data structures always keep keys in an alphabetical order. Therefore, the array element with index greater than 9 use an underscore before their index number. This way it is possible to maintain a desired order for all array elements with the standard sorting method of Libelektras' data structure.

2.6 Related Work

2.6.1 CM-Tools

In scientific areas, there is a lot of research going on in the field of configuration management. Spinellis, for example, in [Spi12] gives a good introduction to the use of CM-tools and why it is worth the effort.

One of the pioneers on the theoretical work on CM-tools was Burgess with his works [BC06; Bur95; Bur03], which formed the basis for *CFEngine*. *Meta-Config* [NIB11] is a CM-tool chain specially designed for provisioning private cloud setups. It comes with its own configuration management engine, however the authors stated to replace it in flavor of *Puppet*, which did not happen until now. Vanbrabant introduces in [Van14] a configuration management framework designed for distributed systems. This framework also comes with its own configuration management engine. Święcicki presents in [Świ16] the CM-tool *Overlord*, which uses its own domain specific language to generate individual configuration programs for each configured system. The author states, that this approach is very flexible and extensible. Additionally, the generated program does not have any dependencies on the target system. A completely different approach of a CM-tool is described by Sherman et al. in [She+05]. The authors present in their paper *ACMS*, a highly fault-tolerant and scalable configuration management tool, which was designed for the infrastructure at Akamai Technologies and is used to operate more then 15,000 servers located in over 60 countries.

Since there are many different CM-tools, all with individual features, usage scenarios and different pros and cons, it is not an easy task for system administrators to decide which tool fits best to their own needs. A lot of CM-tool comparisons can be found on the Internet ([Dun15; Ven13; Tsa10]) and in scientific literature [DJV10; Pan12; Mey+13].

Delaet, Joosen, and Van Brabant present in [DJV10] a comparison framework for CM-tools, which should help to make decisions on a solid basis. For illustration purposes, this framework is used for evaluating 11 different CM-tools. [Pan12] extends this framework by a *community* parameter, which should quantify the public community behind each CM-tool. [Mey+13] describes an automatic quality assurance system for open-source CM-tools. It will automatically measure certain quality attributes on community provided configuration scripts and inform developers of the corresponding scripts, once problems with their extensions arise. This should help to increase the quality of community provided tool extensions.

Shambaugh, Weiss, and Guha present in [SWG16] *Rehearsal*, a tool for static Puppet manifest verification tool. It was build to check if Puppet source files are deterministic and idempotent, which are both fundamental properties of correct Puppet manifests.

2.6.2 Manipulating Configuration Files with other CM-Tools

Other CM-tools have quite similar approaches compared to Puppet. *CFEngine* also allows us defining content of (configuration) files with the following three major approaches: copying from source, template-based and line-based editing. These different methods are all integrated in the *CFEngine* construct "`files`" *Promise*. The line based editing approach works like search and replace and is based on regular expressions, similar to the "`file_line`" resource type of Puppet. *CFEngine* has no Augeas integration. However, *CFEngine* provides format specific configuration file editing operations for common file formats, such as XML- or INI-formats [CFE16b].

Chef, another popular CM-tool implemented in Ruby, allows us defining file content by its resources "`file`", "`remote_file`" and "`cookbook_file`", whereas the latter two copy content from remote locations. *Chef* also supports the ERB template engine and is implemented by the "`template`" resource. Line based editing is only possible through executing specific scripts, for example using the methods provided by the Ruby class "`Chef::Util::FileEdit`". Format specific configuration file manipulations are not possible with standard *Chef* concepts. However, Augeas integration can be added with a community provided extension [Che16a].

Salt enables file management through its "`file.managed`" state. The main use case is to pull the desired content from a *Salt* master node, either as-is, without modifications or by applying one of the supported template engines. Line based editing is possible with the "`file.replace`" state, which also supports regular expressions. JSON and YAML files can be defined by the "`file.serialize`" state, which could be quite useful for simple configuration files, whereas this is an all-or-nothing approach. Additionally, format specific manipulations can be done through the Augeas integration of *Salt* [Sal16].

Ansible supports similar file operations, such as content pulling from remote sources, rendering file templates or line based editing. Relevant *Ansible* modules are "`copy`", "`fetch`", "`template`", "`lineinfile`" and "`replace`". However, *Ansible* has a unique feature especially for configuration file editing, provided by the "`blockinfile`" module.

This allows us managing a complete block of text at once. *Ansible* adds special marker tags before and after the inserted text block, which enables *Ansible* to identify the concrete region within a file. These marker tags are added as comment lines, in order not to break format requirements of the configuration file. Additionally, *Ansible* has built-in support for manipulation of INI-format based files. Augeas support can be added through a community provided module [Red16].

CHAPTER 3

Approach

As we already stated in Section 2.4, Puppet offers already several methods for manipulating configuration files. Each of them was introduced to serve specific use cases. A general form of configuration file manipulation on a per setting basis was introduced by the `augeas` resource type, as described in Section 2.4.7. It comes as a built-in resource type, which suggests that partial configuration file manipulations is an important feature for Puppet users.

However, the `augeas` resource type has a conceptually different design than other format specific resource types. Instead of treating configuration settings as one unit, i.e. one resource declaration per setting, `augeas` operates on tasks of configuration **changes**. This means, a user, defines changes in configuration files in form of tasks. These tasks are executed if certain conditions are met, e.g. the desired configuration setting does not have the expected value. However, these conditions have to be manually defined by the Puppet programmer. This way of configuration manipulation clearly does not follow the typical Puppet way of configuration management: define a desired state and Puppet will trigger actions to transform the current state to the desired one.

This conceptual difference leads to the introduction of several higher level Augeas based resource types, known as *augeasproviders* resource types [Dom17]. The *augeasproviders* project maintains a collection of configuration file format specific resource types, which perform file manipulations using methods provided by the Augeas library. In contrast to the build-in resource type `augeas`, the *augeasproviders* resource types treat each configuration setting as a first class citizen and therefore allows us defining configuration setting states declaratively. However, a drawback of this method is, that for each configuration file format a specific *augeasproviders* Puppet module has to be written, tested and deployed, although each of these *custom* resource types are all using the same underlying library.

Before describing our proposed solution, we want to define some requirements for a configuration file manipulation strategy operating on a per setting basis.

- Settings are treated as first class citizens, i.e. one resource declaration per key value pair.
- Different configuration file formats can be handled by the same resource type.
- Each single setting is addressable by its own unique and stable coordinates, i.e. it can be uniquely identified at any time.

3.1 Is Augeas a Good Candidate?

Implementing a resource type, utilizing the *Augeas'* concepts, which fulfills all three points in a general way, can be problematic because of the way Augeas or many of its lenses were designed. Augeas usually maps structural entities of configuration files into a numbered list of nodes. For example, the `Hosts` lens creates for each entry in `/etc/hosts` one node with the sub-nodes `ipaddr`, `canonical` and `alias`. All *hosts* entries are transformed to this representation and collected in a numbered list, i.e. `/files/etc/hosts/1/...` refers to the first *hosts* entry, `/files/etc/hosts/2/...` to the second and so on. From an Augeas point of view, this is a safe way, since this enables addressing each structural entity uniquely, even if a structural entity with the same data appears several times. However, this way of addressing elements has the drawback, that the content of the structural element – in the *hosts* example, IP-address, hostname or aliases – is not reflected in addressing this element. This means, the absolute pathname for *Augeas'* elements cannot be used to uniquely identify one specific structural element over different modification phases. Listing 3.1 illustrates these problems.

```
192.168.1.3   bbnode1 node1
192.168.1.10  bbclient1 client1
```

Listing 3.1: Hosts file 1

Augeas transforms the *hosts* file shown in Listing 3.1 to an XML-like node tree shown in Listing 3.2.

```
/files/etc/hosts/1/ipaddr = 192.168.1.3
/files/etc/hosts/1/canonical = bbnode1
/files/etc/hosts/1/alias = node1
/files/etc/hosts/2/ipaddr = 192.168.1.10
/files/etc/hosts/2/canonical = bbclient1
/files/etc/hosts/2/alias = client1
```

Listing 3.2: Augeas representation of hosts file 1

Now, if someone adds a new entry between these two existing entries (Listing 3.3), Augeas will transform this file to an XML tree shown in Listing 3.4.

```
192.168.1.3   bbnode1 node1
192.168.1.4   bbnode2 node2
192.168.1.10  bbclient1 client1
```

Listing 3.3: Hosts file 2

```
/files/etc/hosts/1/ipaddr = 192.168.1.3
/files/etc/hosts/1/canonical = bbnode1
/files/etc/hosts/1/alias = node1
/files/etc/hosts/2/ipaddr = 192.168.1.4
/files/etc/hosts/2/canonical = bbnode2
/files/etc/hosts/2/alias = node2
/files/etc/hosts/3/ipaddr = 192.168.1.10
/files/etc/hosts/3/canonical = bbclient1
/files/etc/hosts/3/alias = client1
```

Listing 3.4: Augeas representation of hosts file 2

Let us compare the two Augeas representations shown in Listing 3.2 and Listing 3.4. We see that the added 'hosts' entry for host `bbnode2` is addressed by `/files/etc/hosts/2`, whereas the old unmodified entry for host `bbclient1` is now addressed by `/files/etc/hosts/3`. So the identification of the *hosts* entry for host `bbclient1` has changed. This is not well suited for performing modifications on this 'hosts' entry. In most cases we are asked to change the IP address for a certain hostname instead of changing the IP-address for the 3rd hosts file entry. To solve this problem, Augeas uses an XPath like query language. The 'hosts' entry for host `bbclient1` in both examples can be addressed by the query path `/files/etc/hosts/*[canonical = "bbclient1"]/`. However, this addressing schema is not unique, since the same hosts entry can be identified by `/files/etc/hosts/*[canonical="bbclient1"]`, `/files/etc/hosts/*[ipaddr = "192.168.1.10"]/` and similar ones.

This addressing schema is also used for some INI-style configuration files (e.g. Samba or MySql) and Apt-source configuration files just to name a few.

Puppet requires for all resource declarations a unique identification. Otherwise, it would be possible, that two resource declarations with different titles (IDs) modify the same underlying resource, which leads to in inconsistent state. Using the above Augeas addressing schema to implement a Puppet resource type that treats each setting as a first class citizen, could lead to such non-uniqueness situations.

One could argue, that the *augeasproviders* [Dom17] project already implements Puppet resource types with unique identification schemas using the Augeas library. For example the "`augeasproviders_base`" Puppet module implements an alternative to the built-in "`host`" resource type, working the same way as the "`host`" resource type. However, as already stated above, this project implements a new resource type for each configuration file format. Therefore, this cannot be considered as a general manipulation strategy.

3.2 Integrating Libelektra

The general purpose configuration library Libelektra, as already introduced in Section 2.5, is a suitable candidate for this task. The global shared key database allows us to identify each setting uniquely, using its absolute key name, similar to Augeas. However, in contrast to Augeas, Libelektra identifies each structural element by an ID derived from the data of that element. For example, hosts entries are identified by their hostname, which enables addressing a *Hosts* entry by its hostname.

Libelektras' modular design and plugin architecture allows us integrating many configuration files with different configuration file formats into the same key space. This approach makes it possible to manipulate different configuration files of different formats with one common interface.

This global shared key-value database makes Libelektra a good candidate as a general purpose configuration library, that enables key-value based configuration file manipulation with a common interface for different configuration file formats. Therefore, we want to introduce *puppet-libelektra*, a Puppet module designed to facilitate configuration file manipulations on a key-value basis utilizing the concepts of Libelektra. Thus, it joins the two worlds of Puppet and Libelektra and acts as an interface to Libelektra from within the Puppet-DSL.

To allow Puppet users to benefit from Libelektra's concepts, the Libelektra library has to be integrated within Puppet in a suitable way. In the first place, we have to provide a suitable interface for managing Libelektra keys, their values and optionally key metadata. Additionally, we have to consider a second important concept of Libelektra: *mounting*. If we want to manipulate a specific configuration file, we have to *mount* this file into Libelektra's key space. Once this configuration file was integrated in the key space, its configuration settings can be queried and manipulated using Libelektra's API.

Figure 3.1 shows an architectural overview of how Libelektra is integrated with the Puppet agent. The compiled catalog, as described in Section 2.2.4, is a list of resources with their attribute values, which have to be applied for a specific node/host. The Puppet agent takes this compiled catalog as input and if it contains resources that affect the Libelektra configuration, the Puppet module *puppet-libelektra* will instruct Libelektra to perform the desired actions to enforce the resource's desired state. Figure 3.1 also shows a Libelektra mount configuration for two different mount points: `system/sw/myapp` which uses the *ini* plugin to integrate the file `/etc/myapp/config.ini` and `system/network/hosts` that uses the *hosts* plugin to integrate the file `/etc/hosts` into Libelektra's shared key space.

Puppet offers several kinds of possibilities to extend its functionality [Kan12], but for integrating Libelektra, we have to extend Puppets resource abstraction layer (RAL). This is done by implementing a *custom resource type* together with one or more *custom providers* [Lab16a]. A custom resource type resembles, as the name already suggests, a new Puppet resource type. This custom resource type implementation defines how the

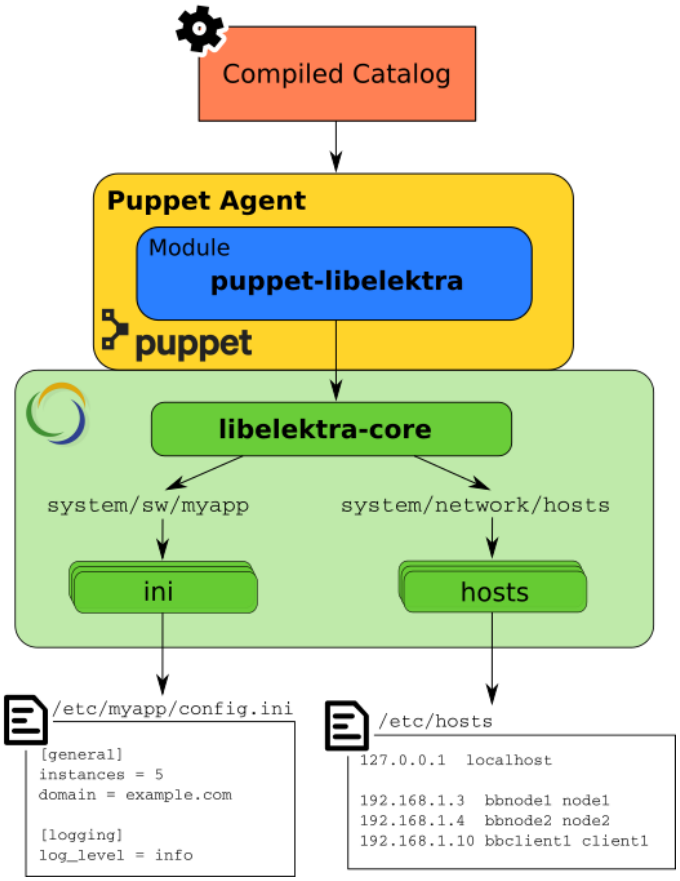

Figure 3.1: *puppet-libelektra* system overview

new resource type is called, which attributes it has and which values it accepts for its attributes. A provider is the main abstraction part within the RAL and sits between one specific resource type and the system managed by Puppet. So a provider determines the current state of a resource and, if necessary, performs actions to enforce this desired state. So basically providers are the working horses of Puppet. One provider is always associated with exactly one resource type, whereas one resource type can have one or more providers [Kan12]. This means, in the first place we have to implement a custom resource type the Puppet user is interfacing with. In the second place we have to provide a custom provider for this resource type, which interacts with Libelektra.

3.2.1 Design Decisions

To provide a good Libelektra integration, we have to consider two Libelektra concepts: *mounting* and *keys*. Libelektra's `KeySet` data structure is not required within Puppet. Puppet provides the data type *array* to create collections of resources [Lab16c]. The Libelektra concept of keys directly maps to a custom resource type, which enables

manipulating a single Libelektra key. This forms the heart of the *puppet-libelektra* module, as it allows us to define a desired state for a single Libelektra key. This further means, we can define a desired state for a particular setting within a specific configuration file. Thus, the setting becomes a first class citizen within the Puppet language.

Each Libelektra mount point is defined once and stored within Libelektras' own configuration. This circumstance drives the idea that the concept of mounting configuration files should be implemented by a separate resource type. This mounting resource type makes it possible to define a desired state for the Libelektra key space itself. Thus, we are able to define which configuration files are mounted in which places within the key space using which set of plugins.

Why two different resource types?

But do we really have to introduce two separate resource types? From a very high level point of view, it would be better to become even more abstract and hide the concept of mounting. A user typically just wants to express certain configuration settings with their values. The 'full' name of Libelektra keys could already be used to define, which configuration setting for which application should be addressed. In theory, this is true, since the hierarchical structure of Libelektras' key space supports this already. For applications already using Libelektra retrieving and storing its configuration settings, a separate mounting is not required, since the mount point was already defined during application installation or the application simply does not require a specific mount point and uses Libelektras' default storage for its configuration.

For applications not using Libelektra this would mean, that we would have to integrate each possible configuration file for all possible - or at least all installed - applications into the Libelektra key space. So we would end up defining a list of mount points for each configuration file beforehand. This is neither practical nor efficient. Additionally, we would lose the flexibility, as integrating custom configuration files becomes even harder because this would mean, that the list of mount points has to be extend once again.

A way without separate mount point definition could be realized by introducing a new Libelektra namespace. This new namespace could be used to automatically integrate configuration files by their absolute file names, thus having key names such as '`files/path/to/config_file.conf/section/setting`'. This concept is similar to the one used by Augeas. However, Libelektra currently does not support a `files` namespace or automatic mounting. Additionally, it opens new questions: How can Libelektra know how to parse the configuration file in question?

So, to allow manipulations on arbitrary configuration files, we have to provide a way to define the configuration file name. But what about merging the two resource types to one, which does an 'automatic' mounting if required? So we could define the configuration file, desired setting and value in one go, similar to the `ini_setting` resource type? Since we are using a general purpose configuration library, this would lead to a badly designed API for the following reason. For each setting we would have to define additionally the concrete

configuration file name, together with an appropriate Libelektra plugin used to parse and write the configuration file. So basically this means, that we have to add a file-plugin name pair to each setting in order to instruct Libelektra which configuration file should be manipulated. Such a design violates the DRY (don't repeat yourself) principal [HT00] and should be avoided. Much worse: how should *puppet-libelektra* behave if there are two different resource declaration for the same configuration file but with different plugins? The `ini_setting` resource type does not have this kind of problem, since it does not require any additional information how to parse the configuration file.

As long Libelektra is not able to automatically mount configuration files in a general and reliable way, we have to specify mount point definitions manually for each configuration file in question. Therefore, two separate resource types, one for a Libelektra key and one for mounting, seams to be the most practical solution. This separation has one additional major advantage: If a user manages configuration settings for an elektrified application (application using Libelektra [Raa10]) or Libelektra integrates such auto-mounting feature one day, we can simply stop using the mount resource type and manage all configuration settings using the key resource type.

3.2.2 Resource Type Attributes

Before we describe *puppet-libelektra*'s resource types in detail, we have to introduce resource type attributes in more detail. Puppet knows two types of resource type attributes: *properties* and *parameter*. A property specifies a specific and measurable characteristic of a resource. This means, if we can change this specific characteristic of a resource, we use a resource property for this characteristic. In contrast to that, a parameter defines how Puppet will manage the resource. For example, the `user` resource attribute `managehome` is a parameter, since it instructs Puppet to create the user's home directory if set to `true`. However, *managehome* is not a characteristic, which can be measured once the Puppet agent was executed [Lab16a; Kan12].

3.2.3 Resource Type *kdbkey*

As already stated above, to integrate Libelektra with Puppet we need a custom resource type to manage Libelektra keys. This resource type will be called *kdbkey*. We use the prefix *kdb* to allow the user an identification that this resource type is part of the Libelektra integration. Also, *kdb* is shorter than *libelektra* or *elektra*.

To facilitate the definition of all required aspects of a Libelektra key, our *kdbkey* resource type, requires the following attributes:

- `name`: defines the full key name and defaults to the resource's title.
- `value`: represents the value of a key.
- `ensure`: defines the existence of a key (i.e. present or absent).

These three attributes form the core of the *kdbkey* resource type and allow us to define the most basic aspects of a Libelektra key: the name of a key, its value and its existence. This simple concept enables us to fulfill our defined requirements for a general purpose configuration file manipulation strategy:

- Each setting is represented by a single resource definition in the Puppet code, therefore the setting is treated as a first class citizen.
- The resource type is configuration file format agnostic. This enables us to describe configuration settings in a general and abstract form.
- Each configuration setting is uniquely identified by its Libelektra key name, which is a stable and unique addressing schema.

Additional to these attributes, the *kdbkey* resource type has the following optional attributes, which enables Puppet users to use more advanced features of Libelektra:

- `prefix`: defines a prefix for the key name. If given, together with the `name` attribute, this defines the full key name.
- `check`: allows us to define specifications for this key.
- `comments`: defines comments for this key.
- `user`: if the key name is within the user namespace, defines which user account should be used.
- `metadata`: allows us to define arbitrary key metadata.
- `purge_meta_keys`: whether not defined metadata should be deleted.

The following sub sections explain each *kdbkey* attribute in detail.

Parameter `name`

This attribute is one of the most important ones, as it defines which Libelektra key should be managed. The value of this attribute defines the key name. This attribute is a *parameter*, since there is no way to declaratively describe that Puppet should update the key name. If we change the value for the `name` attribute for a specific *kdbkey* resource declaration, we do not refer to the same Libelektra key anymore. Instead, we describe the state for a **different** Libelektra key.

This attribute will be used in very seldom cases, because its default value will be linked to the resource's title. This behaviour is important, since it allows us to use the uniqueness of Libelektra keys for the uniqueness of *kdbkey* resource declaration. So if someone tries to define multiple *kdbkey* resource declaration for the same Libelektra key, Puppet will

abort the compilation phase, since the uniqueness for resource declaration was violated. However, a user has the opportunity to overrule this behaviour and choose different values for the `name` property and the resource's title. In this case it might be possible, that two different *kdbkey* resource declarations manage the same Libelektra key. This title-name split up is nothing new and was not designed exclusively for the *kdbkey* resource type, instead it is a general Puppet design decision and can be applied to each resource type.

The `prefix` attribute influences the behaviour of this `name` parameter.

Property `value`

The `value` attribute defines the value for a Libelektra key. Since the value is a measurable aspect, it is a resource type property. As we currently only support string values, all supplied values to the `value` property are implicitly converted to a string representation. However, there is one exception: an array of values. As already described in Section 2.5.5, Libelektra uses a set of sub-keys to realize an array of values. This special key handling is realized by the *kdbkey* resource type, too. Therefore, if an array of values is passed to the `value` property, this particular *kdbkey* resource declaration will be used to manage the set of Libelektra sub-keys representing all array elements. Nested arrays are currently not supported and are implicitly converted to a Ruby string representation.

Property `ensure`

As already described, the `ensure` attribute ensures the existence of a Libelektra key. This allows us to specify declaratively that a particular key should **not** exist. Since the existence of a key is a measurable aspect, this attribute is defined as a resource type property. The name of this property is very common within Puppet and almost all built-in resource types define such a property [Lab16c].

Parameter `prefix`

This optional attribute allows us to define a key name prefix and therefore is a resource type parameter for the same reasons as for the parameter `name`. So, if this parameter is specified, the full key name is defined by the concatenation of the values of `prefix` and `name`. This prefix-name split up has nothing to do with Libelektra key name concepts, such as namespaces. Instead, it has been introduced to facilitate the use of shorter key names and therefore is practical if used in combination with *resource defaults* (Section 2.5).

Property `check`

In Section 2.5.4 we already introduced the ability to add specifications for Libelektra keys. These specifications define restrictions on structure and value of keys. This `check` attribute allows us to add such specifications to a particular key. Since these specifications

are stored within the Libelektra key space and therefore are measurable, the `check` attribute is considered to be a resource type property.

Property `comments`

Some Libelektra storage plugins enable modification of comments within configuration files. This attribute adds the possibility to define such key comments and therefore is clearly a resource type property. If the defined comment string is really attached to the corresponding configuration file setting depends mainly on the underlying configuration file format and the implementation of the used Libelektra storage plugin.

Parameter `user`

The Libelektra namespace feature allows us defining configuration settings with the context of a specific user, i.e. key names starting with '`user/`'. Due to the fact, that the Puppet agent is executed with root-privileges in most cases, Libelektra keys within the user namespace always refer to configuration settings specific to the UNIX `root` user account. To support the definition of user specific keys for other user accounts, this attribute was introduced. The value of this attribute defines the user account to use for key names within the user namespace. Since it influences which specific Libelektra key will be managed, this attribute is a resource type parameter, similar to `name` and `prefix`.

Due to security restrictions, this parameter will only be considered if the Puppet agent is executed with root-privileges.

Property `metadata`

Libelektra supports adding arbitrary metadata to its keys in the form of key-value pairs. This attribute adds the possibility do define such metadata and therefore is a resource type property.

Parameter `purge_meta_keys`

The `metadata` property allows us defining, which metadata key-values pairs should exist for a specific Libelektra key. However, there is no way to express, that some metadata key should **not** exist. So this attribute influences how the `metadata` property will be applied and therefore the `purge_meta_keys` attribute is a parameter.

If given a `true` value, the Puppet agent will be instructed to ensure, that only specified metadata keys should exist and all unspecified but existing metadata key-value pairs have to be deleted. Libelektra internal metadata keys are not affected by this behavior. The default value for this parameter is `false`.

3.2.4 Resource type *kdbmount*

The second central concept within the *puppet-libelektra* module for managing configuration settings within a specific configuration file is **mounting**. To allow a user of the *puppet-libelektra* module to express, which configuration file should be mounted on which Libelektra mount point, we use an additional resource type, called *kdbmount*.

The *kdbmount* resource type requires the following attributes to express a valid Libelektra mount point:

- `name`: defines the Libelektra key name used for mounting.
- `file`: defines which configuration file is used for that mount point.
- `plugins`: list of Libelektra plugins together with plugin configuration settings.
- `ensure`: defines if the mount point should exist or not.

These four attributes form the core of the *kdbmount* resource type and enable the definitions of the most basic aspects of a Libelektra mount point. In addition to these attributes, the *kdbmount* resource type has the following optional attributes:

- `resolver`: defines the Libelektra resolver plugin to be used for mounting.
- `add_recommended_plugins`: add recommended Libelektra plugins to the list of plugins used for mounting.

The following sections explain each *kdbmount* attribute in detail.

Parameter `name`

The value of this attribute defines a Libelektra mount point to use for mounting. For the same reason as for the `name` attribute of the *kdbkey* resource type, this attribute is a resource type parameter.

Similar to *kdbkey's* `name` attribute, this parameter will be used rarely. The default value of the `name` attribute is linked to the resource's title. Therefore, the uniqueness of Libelektra mount points is used for the uniqueness of *kdbmount* resource declaration. So multiple *kdbmount* resource declaration for the same Libelektra mount point will result in the same resource title which Puppet disallows. This way we can avoid managing the same mount point multiple times elegantly. However, the user can choose to override this behaviour, if both the `name` parameter and resource title are defined with different values.

Property `file`

This property makes it possible to specify the configuration file for the Libelektra mount point. In contrast to the used Libelektra key name used for mounting, Puppet does not automatically check the uniqueness of values for this property. One might think, that *kdbmount* should avoid mounting the same configuration file multiple times. However, the value of this attribute does not reflect the absolute path to the underlying configuration file in all cases. The concrete configuration file is determined by a Libelektra resolver plugin. Therefore, two different *kdbmount* resource declaration using the same value for the `file` property, might use different configuration files. Therefore, it is in the hand of the user to ensure one specific configuration file is not mounted several times.

Property `plugins`

When mounting a configuration file, we have to instruct Libelektra which storage plugin should be used for this file. Otherwise, Libelektra has no information how to parse its content. Therefore, at least we have to be able to specify the used storage plugin. Thus, the `plugins` property adds this possibility. However, Libelektra supports the usage of multiple plugins for the, which implement additional functionality, such as specification checks, logging, notifications etc. Therefore, the `plugins` attribute is used to specify a list of plugins used for mounting. Additionally, for each plugin we are able define arbitrary plugin configuration settings in the form of key-value pairs. An example of this is shown in Listing 3.5.

```
kdbmount { 'system/sw/myapp':
  ensure      => 'present',
  file        => '/etc/myapp/config.ini',
  plugins     => {
    ini => { separator => ' '},
    enum => {}
  }
}
```

Listing 3.5: Example `kdbmount` resource declaration with plugin configuration

Property `ensure`

The `ensure` property describes the existence of the mount point declaratively, i.e. it defines whether the Libelektra mount point should be `present` or `absent`. Again, since it is a measurable aspect of a particular Libelektra mount point, this attribute is defined as resource type property.

Property `resolver`

As already explained during the `file` attribute description, Libelektra uses resolver plugins to derive the concrete configuration file name to use. This property can be used

to override the Libelektra default and specify a particular resolver plugin.

Parameter `add_recommended_plugins`

Each Libelektra plugin has the possibility to define a list of required or recommended plugins. Required plugins are automatically added to the list of used plugins for a mount point. Recommended plugins are optional, but can be useful in addition. If this parameter is set to `true`, which is the default case, Libelektra will use all plugins, found in the recommendation lists of already specified plugins, for this mount point. For example the storage plugin "`hosts`" defines the following plugins in its *recommends* list: "`glob`", "`error`" "`network`". If the "`hosts`" plugin is used for mounting and the `add_recommended_plugins` parameter is set to `true`, Libelektra will consider recommended plugins too, therefore the following plugins are used for this mount point: "`hosts`", "`glob`", "`error`" and "`network`". If `add_recommended_plugins` is passed a `false` value, only specified plugins and required plugins, here only "`hosts`", are used for mounting.

CHAPTER 4

Implementation

In Chapter 3 we have introduced our proposed solution for a general purpose configuration manipulation method for Puppet on a key-value basis. This chapter describes some important implementation details of the proposed *puppet-libelektra* module.

As already noted in Section 2.2, the CM-tool Puppet is implemented with the script language Ruby [Mat17]. This design decision of the Puppet developers was taken as they focused on a high developer productivity. Additionally, Ruby supports building non-hierarchical class relationships [Kan12].

The Libelektra core library is written in C with bindings for C++ and other languages [Ele17]. An alternative to using the Libelektra-API is to use the Libelektra command line tool `kdb`, which provides access to the Libelektra key space. Although the Puppet code base provides good utility function for executing external commands, an Puppet-Libelektra integration using the `kdb` command as integration point would have a huge performance penalty. For each key space access we have to execute an external tool, thus launching a new process and parsing the configuration file to manipulate.

Therefore, to provide a good Puppet-Libelektra integration, we have to close this language gap by building Ruby bindings for the Libelektra library in the first place.

4.1 Libelektra Ruby Bindings

The standard MRI Ruby interpreter makes it possible to integrate a new Ruby library using the Ruby C-API [Fit15; Mat14]. Since Ruby is an object-oriented programming language [Fit15], whereas Libelektra has a plain C-level API, building a one-to-one function mapping is not a very good idea, since we lose the advantages of the object-oriented paradigm. Therefore, we use the C++ bindings for Libelektra as an integration point for our Ruby bindings. This way, we directly map the C++ class structure to an

adequate Ruby class structure and add some extensions to provide a better integration with the Ruby standard library.

Instead of writing the whole C++ to Ruby mapping code by hand, we use a wrapping code generator. Such a code generator, takes the C++ class definition as input and generates wrapping code implementing the same C++ class structure for a specific target language. A very good candidate for this job is SWIG [SWI17a], which "[...] is a software tool that connects programs written in C and C++ with a variety of high-level programming languages, [...] [including Ruby]. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code" [SWI17a]. SWIG is already used within the Libelektra project to build Python and Lua bindings.

Generating wrapper code for a specific target programming language using SWIG is done in the following way:

1. In first place the SWIG user writes a SWIG interface file. This source file provides the main blueprint for the wrapping code and includes instructions which C/C++ header files are used and how the wrapping code is generated.
2. SWIG is then executed with a set of command line options, takes the interface file as input and generates a C source file containing the wrapping code for our target language.
3. Finally, the generated C source file is compiled by a standard C compiler such as *gcc* and linked to a shared library. This shared library is used as an extension to the target language [SWI17b].

The Libelektra project consists of two main libraries. A Libelektra core library, providing its basic data types `Key`, `KeySet` and `KDB` together with functions for querying and manipulating keys within the Libelektra key space. The second one is the Libelektra tools library, providing utility functions for manipulating the Libelektra key space itself. In contrast to the core library, the 'tools' library is written in C++ [Ele16]. Since we require the functionality of both libraries to implement the *puppet-libelektra* Puppet module, we have to build Ruby bindings for both libraries.

4.1.1 Wrapping Libelektra Core library

As already shown above, the Libelektra core library contains the following base data structures together with corresponding manipulation functions [Ele16]:

- `Key`: represents a Libelektra key
- `KeySet`: a collection for keys
- `KDB`: handle to a key database

The Libelektra C++ bindings consists of three classes within the `kdb` namespace. Each member function of these three classes wrap the corresponding C functions for manipulating the core Libelektra functions. The C++ bindings include additional functionality for a better C++ integration, such as exceptions, iterators and operators.

By convention, a native Ruby extension consists of one shared library implementing one Ruby module, whereas the shared library is named after the module it contains (Ruby modules start with a capital letter, however the library only contains lower case letters) [Mat14]. However, the Libelektra core API module, named *Kdb* breaks this convention to allow us building the Ruby binding with Ruby code and the SWIG generated wrapper code. Figure 4.1 shows a component overview of the Libelektra core API module *Kdb*.

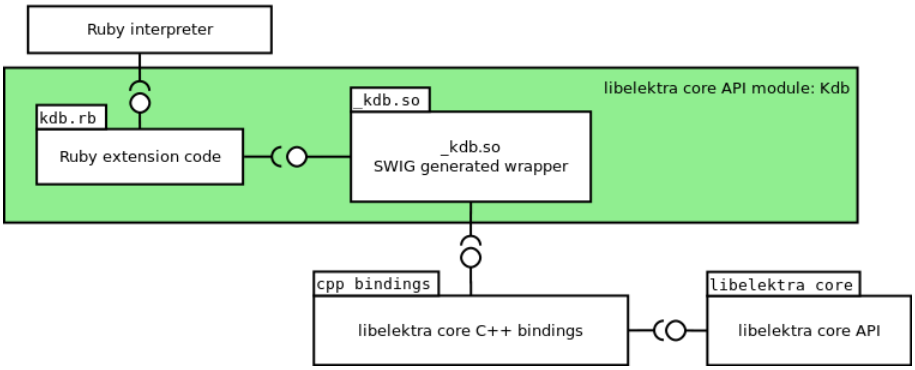

Figure 4.1: Module *kdb* component structure

The SWIG generated wrapper code is compiled and linked to a "`_kdb.so`" library. However, this library implements the module "`Kdb`" with all Ruby classes corresponding to the Libelektra C++ classes described above. This naming inconsistency is resolved by the Ruby source file "`kdb.rb`", which loads the "`_kdb.so`" library and adds certain helper functions to the already implemented Ruby classes. Such a feature extension is possible, since Ruby enables us to add functionality to already existing classes. This feature is typically called *reopening*, *augmenting* or *monkey patching* a class [Fit15]. Finally, the Ruby interpreter loads the *Kdb* module by calling "`require 'kdb'`", whereas all classes, from the "`_kdb.so`" library, monkey patched with extensions found in "`kdb.rb`", are available within the module "`Kdb`". We found this design in other binding modules such as the *libvirt* Ruby bindings [Lal16].

To provide a good integration with the Ruby standard library, the Ruby *Kdb* module contains several differences to the Libelektra C++ bindings. One of the most obvious one is the auto renaming feature of SWIG. This auto renaming feature renames all class, field, method and function names to follow the Ruby naming conventions [SWI17b]. Additional to this auto renaming, the Ruby bindings contain the following notably differences:

`Key` and `KeySet` iterators

Iteration over a collection of objects is typically done in Ruby using the "`each`" method. This method is available in all collections and takes a block of code, which implements the iteration body. To support this kind of iteration, the collection class `KeySet` is extended by an "`each`" method, whereas the C++ iterator function are excluded and not accessed by the wrapping code. However, the `KeySet` cursor function are still available. Additionally, to the new "`each`" method, `KeySet` includes the Ruby mixin "`Enumerable`". This mixin implements useful collection functions such as "`find`", "`sort`", "`map`" etc.

The "`Key`" class has a list of metadata key-value pairs. Internally, this is realized by a "`KeySet`" data structure. The Ruby class "`Key`" provides read-only access to this internal "`KeySet`" through a "`meta`" getter method. This way we can directly benefit from the "`KeySet`" extensions and provide Ruby style iteration also for metadata keys for the "`Key`" class.

Array semantics for `KeySet`

As described above, the "`KeySet`" class implements an "`each`" method and provides useful collection methods via "`Enumerable`". To provide a class that really behaves like a Ruby collection, such as "`Array`", the Ruby "`KeySet`" version implements two additional operator methods: "`[]`" and "`<<`". The first one, "`[]`" enables index-based access to keys, whereas "`<<`" is the append operator, to add new keys to the key set. These two methods, together with the "`each`" method enable the class "`KeySet`" to behave as the "`Array`" class.

`Key` construction with variable argument list

The Libelektra C-API provides "`Key`" allocation function with a variable argument list to allocate and initialize a new key in one go. However, Ruby does not support variable argument lists. Instead, the Ruby syntax facilitates the imitation of variable argument lists and offers a different and more flexible way. Ruby allows us to create a hash object with "`value: "hello", meta: "world"`". This creates a hash object with the keys "`value`" and "`meta`". Calling a function with such a hash initialization, looks like calling the method with keyword arguments. Instead, the method is called with only one argument, the initialized hash object. This way, we can imitate methods with a variable keyword argument list [SWI17b; Bro09]. This technique is realized in the constructor of "`Key`".

`KeySet` creation

The Libelektra C/C++-API allows us to pass an arbitrary number of "`Key`" instances to the "`KeySet`" allocation function, which is realized again by a variable argument list. This time the Ruby *Kdb* module uses a Ruby array for this form of "`KeySet`" initialization.

Exception handling

An important aspect of building a Ruby extension via a C++-API is exception handling. If one of the C++ calls throws an exception, the wrapping code has to catch the C++ exception and initiate the Ruby exception process. Since the Ruby VM is not designed to catch C++ style exceptions, the thrown C++ exception is not handled. Therefore, the running program will be aborted. To avoid such situations we have to carefully look for all C++ calls that might throw an exception and instruct the SWIG wrapper generator to add exception handling code around this C++ call. The exception handling code simply converts the C++ exception instance to a corresponding Ruby exception instance and throws a Ruby exception.

4.1.2 Wrapping Libelektra tools library

Some functionality, such as mounting is implemented within the separate Libelektra tools library. As the Puppet module *puppet-libelektra* requires this functionality, we also implemented Ruby bindings for this library.

The Libelektra core Ruby binding may also be used by other applications, so we have tried to keep it small and unbolted. Therefore, we decided not to wrap both Libelektra libraries within one Ruby module. Instead, both libraries are wrapped by separate Ruby modules, which may be distributed and used on its own. The Libelektra tools library is wrapped by the Ruby module called *Kdbtools*. However, since the 'tools' library also makes use of the data structures found in the Libelektra core library, we have to consider this dependency for our Ruby modules, too. Therefore, the *Kdbtools* module requires the *Kdb* module to work correctly.

In contrast to the Libelektra core library, the 'tools' library is developed with C++. This C++-API is directly used for generating the wrapping code with SWIG. Similar to the *Kdb* Ruby module, the *Kdbtools* module uses the two folded architecture, too. The SWIG generated wrapping code is compiled and linked to a "`_kdbtools.so`" shared library, which is loaded by the module file "`kdbtools.rb`". In contrast to "`kdb.rb`", the "`kdbtools.rb`" does not implement any additional functionality at the moment, however since this architecture has proved to be useful for the module *Kdb*, it it used for *Kdbtools* too, to facilitate possible extensions in the future.

Using the wrapped data structures found in module *Kdb* within the SWIG generated wrapping code for the module *Kdbtools*, proved to be a real challenge. The SWIG documentation [SWI17b] includes a section about creating multi-module projects but without going into much detail. It simply states to include the interface file for the required module (*Kdb*) and to use the same value for the option "`SWIG_TYPE_TABLE`". However, this does not work for Ruby generated wrapping code, as the SWIG type table for the types "`Key`" and "`KeySet`" includes wrong or uninitialized values. Ruby wrapping objects for these data structures created within the *Kdbtools* wrapping code do not match the type information found in the module *Kdb*. Therefore, "`Key`" and "`KeySet`" objects created by the 'tools' library are not usable within Ruby code. This seems to be a

bug in the current SWIG version[1]. To solve this problem, the *Kdbtools* module patches SWIG's type table records during module initialization to correct the wrong entries for the Ruby types "`Kdb::Key`" and "`Kdb::KeySet`". This "ugly" workaround is required until newer SWIG versions have a fix for this problem.

The most part of the SWIG interface file for *Kdbtools* contains instructions for exception handling. The Libelektra tools library uses many custom exception classes, which makes the SWIG interface file complex. Due to the lack of proper tool support, searching for functions probably throwing exceptions was done manually.

4.2 Puppet Module *puppet-libelektra*

As already described in Section 3.2, Libelektra is integrated with Puppet by implementing a new Puppet module extending Puppets resource abstraction layer (RAL). Therefore, the *puppet-libelektra* module implements two new resource types: *kdbkey* and *kdbmount*. Section 3.2 also mentions, that for creating a new custom resource type, we have to implement at least two different classes. One *type* class and one or more *provider* classes. Each resource type implements two different providers. One provider using the Ruby bindings, called *ruby*, and another provider using the Libelektra tool "`kdb`", called *kdb*. The default provider is *ruby*. However, if the Ruby bindings cannot be loaded, the alternative *kdb* provider will be used. This way we can ensure, the *puppet-libelektra* module is usable once Libelektra is installed on the managed machine.

The following sections describe the implementation of the custom resource types *kdbkey* and *kdbmount* in more detail.

4.2.1 Architecture of a Custom Resource Type

The architecture for both resource types, *kdbkey* and *kdbmount*, is basically the same. Each resource type implements a *type* class and two *provider* classes with one common parent class. This common parent class is mainly used for sharing utility functions. Figure 4.2 shows the class relationship for the *kdbkey* resource type.

A new resource type is integrated in Puppet by creating a new 'type' class, derived from "`Puppet::Type`" (Figure 4.2 class "`kdbkey`"). Providers are created by defining a new class derived from "`Puppet::Provider`". Figure 4.2 shows three classes derived from "`Puppet::Provider`".

- "`kdb`" implements a provider utilizing the Libelektra command line tool "`kdb`".
- "`ruby`" implements a provider utilizing the Libelektra-API directly.
- These two classes share a common parent class "`common`", which implements shared functionality. This parent class is derived from "`Puppet::Provider`", which makes the classes "`kdb`" and "`ruby`" to Puppet providers.

[1] Bug report: `https://github.com/swig/swig/issues/903`

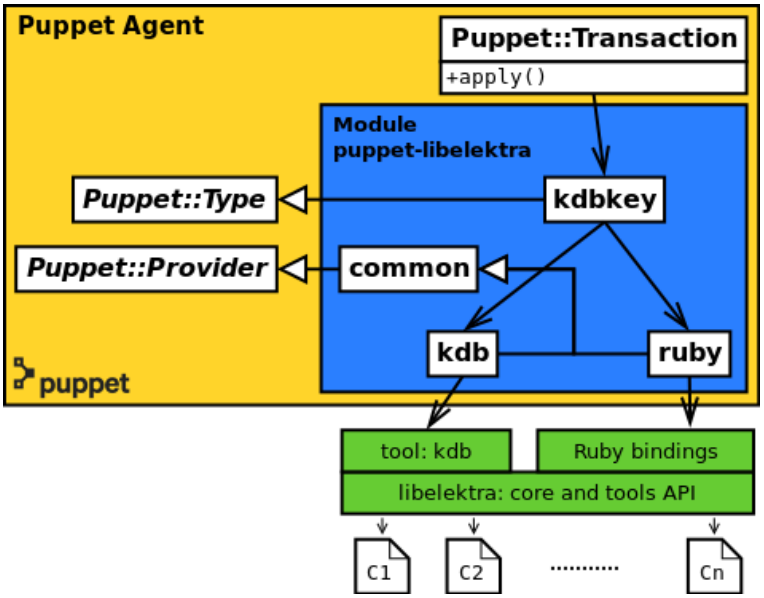

Figure 4.2: Class relationship for resource type *kdbkey*

Figure 4.2 also shows communication paths between the Puppet agent and Libelektra. The "`apply`" method for an instance of "`Puppet::Transaction`" traverses all resources found in the catalog to be applied. For each found *kdbkey* resource, a new "`kdbkey`" type instance and new provider instances, "`kdb`" and "`ruby`" are created. The resource update process is delegated to the new "`kdbkey`" resource instance, which further delegates the actual work to one of the newly created provider instances. Which of the two provider instances is actually used, is determined by the type instance. Puppets default resource type implementation, "`Puppet::Type`" offers several mechanisms for this decision.

1. Each provider itself checks if they can be used on the managed machine in question. Such checks usually test if required system commands or operating system features are available.

2. After all provider suitability checks were done, the resource type instances chooses the most suitable provider.

This provider will then be used for performing all updates. However, since an automatic provider decision is not possible in all cases, the user may overrule this by specifying a concrete provider by the meta-attribute "`provider`".

The interface between the resource type class and its provider classes is not fully specified and may be implemented freely by Puppet module designers [Lab16a]. However, Puppet suggests the following default interface, as the default resource type implementation makes use of this interface:

- For each *ensurable* resource type (having an "`ensure`" property), corresponding providers should implement an "`exists?`" method to query if the resource in question exists or not. However, if the provider implements a so called *prefetching* mechanism, the "`exists?`" method is not required.
- Providers for *ensurable* resource types should implement a "`create`" and "`destroy`" method, to create or delete the managed resource.
- For each resource type **parameter** (changeable resource attributes), providers should implement a Ruby-style getter and setter method. The getter methods are used to check the resource's current values, whereas the setter methods may update these values, if required.
- An optionally "`flush`" method is called to perform all updates in one go. The "`flush`" method will be called, if the resource received any updates by calling one of the parameter setter methods.

The concrete provider class implementation contains all required code parts for querying and updating the underlying resource on disk. All provider implementations found in the *puppet-libelektra* module follow this provider interface recommendation, as it is suitable for our tasks and helps to reduce lines of code.

Since providers implement the operating system interfacing parts, custom type classes, mostly implement the Puppet agent interfacing parts and therefore only define the resource type's attribute implementations. The two *puppet-libelektra* resource type implementations, "`kdbkey`" and "`kdbmount`", mostly consists of these attribute definitions and implement all attributes defined in Section 3.2.3 and Section 3.2.4.

There is one notable aspect of the "`kdbkey`" resource type implementation. Normally, each resource type class defines exactly one *namevar* attribute (an attribute which is associated with the resource title and uniquely identifies a resource instance). However, in the case of "`kdbkey`", we have two attributes, which uniquely identify a Libelektra key: "`name`" and "`prefix`". Therefore, these two attributes together form the *namevar*. However, if both attributes are omitted by a resource declaration, which one should receive the resource's title value? If the title is assigned to both attributes, *puppet-libelektra* will reference the wrong key, since the key name is defined by the concatenation of "`prefix`" and "`name`". Therefore, Puppet requires to implement a static "`title_patterns`" method, which resolves this conflict. In the case of *kdbkey*, if both attributes "`prefix`" and "`name`" are omitted, the "`name`" attribute will receive the title's value.

4.2.2 Automatic Relationships between *kdbmount* and *kdbkey* Resources

If not explicitly specified, resource declarations found in the catalog do not have any order relationship. Therefore, the specific order in which resources are applied is undefined and

may vary between different Puppet agent runs. This is no problem, as long resources do not have any dependences on each other. However, in many cases resources do have dependences to other ones. Therefore, Puppet offers several ways to specify resource relationships.

In addition to manual relationship specifications, Puppet also offers a way to automatically compute relationships between resource definitions [Lab16a]. For example: a particular file may only be created, if its parent directory already exists. Therefore, if the file and its parent directory are both managed by Puppet, the tool is able to automatically compute an order relation between these two resource instances, since it does not make any sense to create the file before its parent directory is created. So, the "`file`" resource type (responsible for both, files and directories) automatically adds a '*requires*' relationship from the file to its parent directory resource instance [Lab16c].

In the context of Libelektra we are able to compute such automatic relationships, too. For example, if there is a configuration file mount point and a key, which is located below this mount point, we can directly conclude an order relationship between the mount point and the key. It only makes sense to first mount the configuration file on the specified mount point and afterwards create the defined key. If this is done in the reverse order, the specified key will not modify the configuration file, since the file is mounted after the key was created.

Automatic relationship definitions are specified by the resource type implementation. To be more specific: a resource type class may register an "`autorequire`" routine, which returns all possible resource types and resource IDs a particular resource instance may depend on. If one of the returned resource IDs is actually managed by Puppet and therefore found in the catalog, Puppet automatically adds an '*requires*' relationship between the particular resource and the resource found in the 'autorequire' list.

The *kdbkey* resource type implements such an "`autorequire`" method for *kdbmount* resource type instances. Thus, this "`autorequire`" method returns a list of all possible *kdbmount* resource instances, the particular *kdbkey* resource instance may depend on. The following example illustrates this behavior. The "`autorequire`" method of class "`kdbkey`" for the resource instance with ID "`system/sw/samba/global/workgroup`" returns the following list of *kdbmount* resource instances:

```
Kdbmount[system/sw/samba/global/workgroup]
Kdbmount[system/sw/samba/global]
Kdbmount[system/sw/samba]
Kdbmount[system/sw]
Kdbmount[system]
```

If any of these *kdbmount* resource instances is found in the catalog, Puppet adds a '*requires*' relationship from "`Kdbkey[system/sw/samba/global/workgroup]`" to the found *kdbmount* resource instance. Thus, the *kdbmount* resource instance will be

applied **before** the *kdbkey* resource instance is applied. All other non existing resource instances in this list are ignored by Puppet. This *kdbmount* resource instance list is generated by simply removing each path component of the *kdbkey* resource name one by one.

The same concept is applied for other *kdbkey* resource type instances too, since such an automatic order relation between a key and its parent keys can also be computed automatically. So the "`autorequire`" method of class "`kdbkey`" for the Libelektra key from the example above, "`system/sw/samba/global/workgroup`", generates the following list of *kdbkey* resource instances:

```
Kdbkey[system/sw/samba/global]
Kdbkey[system/sw/samba]
Kdbkey[system/sw]
Kdbkey[system]
```

The only difference between this list and the *kdbmount* list is, that the resource instance for ID "`system/sw/samba/global/workgroup`" is not included. Otherwise, the specified key would have an '*requires*' relationship to itself and therefore leads to a dependency cycle.

The *kdbmount* resource type does not specify any automatic order relationships.

4.2.3 *kdbmount* Providers

Section 4.2.1 already mentioned the two *kdbmount* providers, "`kdb`" and "`ruby`". Each of them implement one way of managing the Libelektra key space.

Prefetching of Existing Mount Points

Both providers implement the previous mentioned *prefetching* mechanism. Prefetching works in the following way:

1. In the first step a static provider method ("`instances`") is used to query all existing resources on the target machine, for this specific type.
2. For each found resource, a new provider instance, initialized with the current resource values, is created.
3. A static "`prefetch`" method is then used to associate the newly created provider instances with corresponding resource type instances.
4. Provider instances for non-existing resources are created afterwards.

So, instead of querying each resource separately, the provider is used to query all resources of a particular type in one go. This mechanism is better suited, if all available resources for a specific type can be queried with just a few method calls [Lab16b].

For this reason, both providers implement a static "`instances`" method to query all available Libelektra mount points and to create provider instances for each of them. This avoids querying each Libelektra mount point separately.

`ruby` Provider

As already mentioned, the "`ruby`" provider uses the functionality of Libelektras' 'tools' library. However, this library currently implements functions for creating mount points only. Querying information of existing mount points has to be done by the provider implementation itself. However, this can be done by interpreting the information found in Libelektras' path `system/elektra/mountpoints`. New mount points and changes to existing ones are done in the same way. Based on the user defined *kdbmount* property values, the "`ruby`" provider uses Libelektra 'tools' to build a new *mount backend* configuration, which is simply added to `system/elektra/mountpoints`. Backend configurations for existing mount points are simply overwritten.

`kdb` Provider

The "`kdb`" provider works in a similar way. Existing mount points are prefetched using the tool "`kdb`" with a single "`mount`" command argument. Since this tool call provides information for all existing mount points, only one invocation is required. New mount points are created by executing "`kdb mount`" with a suitable set of arguments, which are required to define a mount point as requested by the user. Mount points are deleted by executing "`kdb umount`" with the mount point name as command argument. Changes are done by simply recreating the mount point.

4.2.4 *kdbkey* Providers

As *kdbmount*, also the *kdbkey* resource type implements two different providers, "`ruby`" and "`kdb`", to enable using the *puppet-libelektra* module if the Ruby bindings are not available. In general, the "`kdb`" provider uses the Libelektra tool "`kdb`" for managing keys, whereas the "`ruby`" provider directly utilizes Libelektras' core API.

The following sections give a deeper insight on specific implementation details for both providers.

No Prefetching

Kdbkey providers do not use the prefetching mechanism, since this may cause too much overhead. If Libelektra mounts a lot of different configuration files but Puppet is used to manage only a subset of them, prefetching requires reading and parsing all mounted configuration files, wasting memory and CPU cycles. Therefore, both *kdbkey* providers

implement an "exists?" method to query the existence of one particular Libelektra key.

Key Specifications

Section 2.5.4 describes Libelektra key specifications. The *kdbkey* property "check" allows us defining such key specifications. This specification definition is done by passing a hash object of specification metadata to the "check" property. Each key within this specification hash defines a metadata key for the desired Libelektra key. Listing 4.1 shows an example *kdbkey* resource declaration with two key specifications.

```
kdbkey { 'system/sw/myapp/priority':
  ...
  check => {
    'type'  => 'short',
    'range' => '0-9'
  }
}
```

Listing 4.1: *kdbkey* resouce declaration with a key specification

However, instead of adding this specification metadata to the managed Libelektra key itself, the *puppet-libelektra* module writes this metadata keys to the corresponding key within the spec namespace. This namespace is used by Libelektra for all key specifications. The Libelektra "spec" plugin takes care of transferring metadata from the spec key to real key. This way we can use key specifications also for storage plugins not supporting metadata persistence. So, for the example from Listing 4.1 this means, that the specification metadata is written to the key "spec/sw/myapp/priority".

Despite this clear separation of specifications and values, there is one exception where this approach is not working properly. If Puppet is used to add a new key with specifications, the new key is created together with its corresponding spec-key. However, Libelektra applies metadata from the spec-key during the read phase, not during its write phase. Therefore, validation plugins are not working correctly since they have no validation metadata available. To workaround this limitation, the *puppet-libelektra* module adds the specification metadata to both keys, the spec-key and the real key.

Since Libelektra uses the metadata key name prefix "check/" for all specification metadata keys, the "check" property simply adds this prefix to each defined specification key. This facilitates writing shorter specifications. So the key specification from Listing 4.1 will lead to the metadata key names "check/type" and "check/range".

Managing an Array of Values

Section 2.5.5 describes how Libelektra uses multiple keys for building an array data structure. To easily enable manipulating such an array, *kdbkey* provides the possibility

to pass an array of values to the "`values`" property. The two *kdbkey* providers are able to handle such situations. If an array value has to be written, both providers create all necessary array sub-keys. It is important to consider the reverse direction, too. If an existing key with multiple values is updated by a single value, all array sub-keys have to be removed again.

Keys for Different Users

The Libelektra `user` namespace is unique for each user account. If Puppet is used to manage keys within the `user` namespace, Libelektra considers configuration files for the user account the Puppet agent is executed with. However, sometimes it may be necessary to manage user-keys for a different user account. Therefore, the "`user`" parameter of *kdbkey* allows us to change the default behavior. But the Libelektra API does not offer any functionality to realize this for the moment. Instead, Libelektra uses the user account of the current process together with a set of environment variables to determine the correct location of configuration files for keys within the `user` namespace.

To realize this functionality, both providers use the same strategy. They simply change the user account and environment variables for the process using the Libelektra library, to make Libelektra believe it is executed within the context of the requested user. The *kdb* provider can do this in a very simple way. Since all Libelektra interaction is done by an external command ("`kdb`"), all command invocations are done with permissions of the requested user.

Implementing the same functionality for the *ruby* provider requires more effort. The *ruby* provider does not use external commands, so changing the user context for the process, which is using the Libelektra API, requires changing the user context of the Puppet agent itself. Although, this seems to be simple, UNIX operating systems only permit changing user privileges from more privileged to less privileged users. So if the Puppet agent is executed with `root` permissions and the *ruby* provider changes the current user of this Puppet process to an unprivileged one, this privilege change cannot be reverted, since the Puppet agent is now executed with permissions of an unprivileged user. However, UNIX offers different types of permission changes and changing the real user-ID is not required in this case. The *effective user-ID* is used by the operating system for file system operations. Thus, it is perfectly suited for our task, since Libelektra mainly reads and writes files. In contrast to the real user-ID, the effective user-ID may be changed back to original one, since the real user-ID is still a privileged one. So, the *ruby* provider changes the effective user-ID for the current Puppet process, which allows Libelektra to read and write files with permissions of the requested user account. However, there is a second issue. Libelektra determines the correct file locations by a set of environment variables. Therefore, the *ruby* provider "fakes" the environment variables "`USER`", "`HOME`" and "`XDG_CONFIG_HOME`" in addition. These permissions and environment modifications are done before each "`kdbOpen`", "`kdbGet`" and "`kdbSet`" call. Afterwards these modifications are reverted. This way Libelektra can be "tricked" to maintain user-keys for a different user account.

It may be worth noting, that all this permission switching operations are only working correctly, if the Puppet agent is executed with `root` permissions.

CHAPTER 5

Feature Comparison

Our proposed solution *puppet-libelektra* is not the only Puppet configuration file manipulation method. Instead Puppet offers a variety of resource types to manipulate configuration files, as already illustrated in Section 2.4. The aim of the next three chapters is to show how our solution – with its two resource types "`kdbkey`" and "`kdbmount`" – fits into the set of existing Puppet file manipulation methods and if a system administrator benefits from using our solution.

This chapter documents the first evaluation step, whereas Chapter 6 and Chapter 7 present two further evaluation steps.

5.1 Goals

The goal of this evaluation is to compare the proposed *puppet-libelektra* module with other Puppet file manipulation methods from a functional and technical point of view. First we want to show, if it is possible to create syntactically incorrect configuration files with one of Puppets configuration file manipulation methods. In other words, we want to answer the question, if it is save to blindly trust the output of Puppets configuration file manipulation methods, or if system administrators should be aware of weaknesses and limitations of the evaluated methods. Second we want to explore the runtime performance of each Puppet configuration file manipulation method to answer the question, if any of the evaluated methods has significant runtime performance deficits. This will be an important aspect when managing tens or hundreds of configuration settings.

5.2 Evaluated Configuration File Manipulation Methods

We picked three different configuration file formats for this evaluation step and used up to three Puppet configuration file manipulation methods (resource types) for each file

format. We have chosen to use the following file formats for this evaluation: Hosts files, INI-style configuration files and JSON configuration files.

For each configuration file format we used two general purpose methods, our proposed solution "`kdbkey`" and the Puppet build-in resource type "`augeas`", and one format specific method: for INI-style configuration files the resource types "`ini_setting`" and for Hosts files the resource type "`host`". Puppet does not offer a format specific manipulation resource type for JSON files.

The following list summarizes the used file formats and evaluated Puppet configuration file manipulation methods:

- Hosts files
  - resource type "`host`", a built-in Puppet resource type
  - resource type "`kdbkey`" with Libelektra plugin "`hosts`"
  - resource type "`auges`" with Augeas lens "`Hosts.lns`"
- INI-style configuration files
  - resource type "`ini_setting`", available via the Puppet module *puppetlabs-inifile*
  - resource type "`kdbkey`" with Libelektra plugin "`ini`"
  - resource type "`augeas`" with the Augeas lens "`Koji.lns`"
- JSON configuration files
  - resource type "`kdbkey`" with Libelektra plugin "`yajl`"
  - resource type "`augeas`" with the Augeas lens "`Json.lns`"

For each of these file formats we compared the mentioned Puppet resource types in terms of robustness of syntax validation (Section 5.3) and runtime performance (Section 5.4). In addition to the general purpose and format specific file manipulation methods we also used the Puppet resource types "`file`" and "`file_line`" for comparing the runtime performance.

5.3 Robustness of File Format Syntax Validation

When modifying configuration files, ensuring the correct syntax of the underlying format is important. Otherwise, programs using the configuration file are not able to correctly parse the file and therefore cannot retrieve information stored in it. Some Puppet file manipulation methods automatically ensure the correct syntax of configuration files, assisting a Puppet user to avoid creating syntactically wrong configuration files. This section compares the robustness of these automatic syntax checks.

5.3.1 Hosts Files

Many operating systems use Hosts files to resolve host names to IP-addresses. On UNIX systems "`/etc/hosts`" is a typical example for such files. The syntax for a Hosts file can be found in Linux Man page "`HOSTS(5)`" [16]. Listing 5.1 shows a sample Hosts file.

```
# localhost IP
127.0.0.1     localhost

# sample comment
192.168.13.5  saturn.mydomain.org saturn   # saturn host
```

Listing 5.1: Sample Hosts file

Each line consists of a single Hosts entry, containing an IP-address, a host name and a list of aliases. All these elements are separated by white space. Text from a "`#`" character until the end of the line is comment, and is ignored [16].

Beside this file format the following additional syntactical checks can be applied: validity of the specified IP-address, host name and aliases, as defined by RFC1123 [Bra89].

Resource Type `kdbkey`

Libelektra brings a storage plugin, called "`hosts`" for parsing and writing Hosts configuration files. Each line is translated to a set of keys, describing a single Hosts entry. The last Hosts entry in Listing 5.1 is translated to the following set of keys:

```
.../ipv4/saturn.mydomain.org = 192.168.13.5
.../ipv4/saturn.mydomain.org/saturn = 192.168.13.5
```

Libelektra additionally has an IP-address validation plugin to check the correct format of the given IP-address. In addition, also the IP-address number ranges are check, rejecting IP-addresses such as "`345.1.1.1`".

The "`hosts`" plugin does not perform any host name or alias validation. Since the "`hosts`" plugin uses the host name and each alias within key names, there is actually no way to check the validity of host or alias names for the moment. This missing host and alias name validation is a major problem, if a user passes a host or alias name containing white space. If creating a new Hosts entry by adding a host key, consisting of two words, thus separated by a space character, Libelektra writes this host name as-is directly into the file. During a subsequent read cycle, this Hosts entry is interpreted differently, since the first word will be considered to be the host name and the second word is interpreted as the first alias name. From a Puppet point of view this is now a different entry, thus a subsequent Puppet agent run will add the faulty Hosts entry again, leading to a duplicated entry. The same problem exists for alias names. An alias consisting of two, space separated words, is written directly into the file and interpreted

differently during the next parsing cycle. However, since the "`hosts`" plugin collapses duplicate entries during the parsing process, this duplication process is limited to two occurrences.

The problem of the missing host name validation becomes even worse, if a *new-line* character is used within a host or alias name. This breaks the syntax of the Hosts file, leading to a syntactically incorrect configuration file.

The implementation of *kdbkey's* "`comments`" property splits multi-line comments line by line. Thus, an incorrect white space character, such as the new-line character, cannot lead to a syntactically invalid configuration file. However, this protection is not applied when manually manipulating the corresponding "`comment`" metadata directly.

Resource Type `host`

The "`host`" resource type is a built-in resource type designed to manipulate single entries within a Hosts file. Since this resource type is specifically designed for this purpose, its implementation performs a validation of all passed values. IP-addresses are automatically checked for validity. Also, host and alias names are checked for not allowed characters. However, the maximum length of 255 characters for host or alias names is not checked.

Creating several Hosts entries for the same host name is not possible, since the host name for each resource declaration is used as unique resource identifier.

The "`host`" resource type features a "`comment`" property to add arbitrary documentation text to each Hosts entry. However, these comments are treated as inline-comments only, which does not allow us defining multi-line comments. Moreover, a check to prevent writing multi-line comments is missing. Therefore, if passing a string containing a new-line character to the "`comment`" property, the "`host`" resource type may create a syntactically incorrect configuration file.

Resource Type `augeas`

Augeas features a "`Hosts`" lens, which allows Augeas to parse and modify Hosts files. However, this lens is not designed to perform strict value validation for IP-addresses, host and alias names. Invalid characters or the maximum allowed name length are not checked, as well as invalid IP-addresses. It is up to the user to perform these checks manually. However, Augeas' "`Hosts`" lens does not permit white space in one of the supplied values. Therefore, similar problems, as Libelektras' "`hosts`" plugin suffers from, are avoided. Also, comment lines are protected from invalid new-line characters.

As Augeas accepts invalid IP-addresses for Hosts entries, the "`augeas`" resource type can also be tricked to write a syntactically incorrect Hosts file.

5.3.2 INI-style Configuration Files

The INI file format is an informal configuration file format consisting of *sections*, *properties* and *values*. Since there is no official standard for INI files, there exists a lot of different implementations with varying features. However, most implementations follow these basic principles: Each line consists either of a section start tag, a property value pair or a comment. Lines starting with a "`#`" or "`;`" character are treated as a comment and are ignored. Properties and values are separated by a "=" [Wik17]. Listing 5.2 shows a sample INI configuration file.

```
globalsetting = value

# comment line
[section 1]
property = value

[section 2]
property = value
```

Listing 5.2: Sample INI file

Since the definition of an INI file is not clear enough, we cannot be very concrete on how to break its syntax. For this evaluation, we will consider the following points as syntax violations:

- property value separator "=" in properties
- new-line characters in section or property names
- unmarked multi-line values. That is, if a multi-line value is not properly tagged as continuation of the value and may lead to interpreting parts of the value as new property.
- initially defined comment which is not prefixed by a "`#`" or "`;`" character

Resource Type `kdbkey`

Libelektra comes with several storage-plugins parsing INI files. The most mature implementation provides the "`ini`" plugin, which is used during this evaluation.

The "`ini`" storage plugin offers a *multiline* option, which facilitates reading and writing multi-line values. Continued lines of a multi-line value are indented by several white space characters, marking them as continuation the value for the current property. If this option is disabled, the plugin refuses to write values containing a new-line character.

However, section and property names are not protected by a possible line break. Therefore, if a section or property name contains a new-line character, the "`ini`" plugin writes a syntactically wrong configuration file.

When passing a value containing the separator character "=", the behavior of the "ini" storage plugin is unaccountable. The complete value enclosed in double quotes is written to the file. However, an additional property, consisting of the separator character enclosed in double quotes, with an empty value is added, too. Also, adding the separator character to a property name leads to an unexpected behavior. So, the "ini" storage plugin developers implemented extra features to support separator characters within properties and values, however this feature seems to be broken at the moment. A separator character within a section name is handled correctly.

As the "kdbkey" implementation itself takes care of possible new-line characters in comment values, comments are always properly written.

Resource Type ini_setting

The "ini_setting" resource type, implementing an INI parser written in Ruby, enables manipulating INI-style configuration files on a per setting basis.

Similar to the "ini" Libelektra plugin, the "ini_setting" implementation is not aware of possible new-line characters within section or property names. Therefore, this resource type writes syntactically incorrect configuration file if a section of property name containing a new-line character is used. In contrast to the Libelektra "ini" plugin, the "ini_setting" implementation also is not aware of possible new-line characters within property values. Similar to section and property names, multi-line values are simply written to the underlying configuration file, leading again to an incorrect syntax. Moreover, subsequent Puppet agent runs append these line continuations again, leading to an ever growing configuration file.

Adding a separator character to a section name or a value string is no problem for the "ini_setting" resource type. Both cases are handled correctly. However, adding the separator character to a property name, leads to an unexpected behavior. The property string is written directly into the configuration file. Therefore, subsequent Puppet agent runs do not detect this property string correctly, so the same faulty property string is added a second time, again, leading to an ever growing file.

Since the "ini_setting" resource type does not offer handling comment lines. Its implementation takes care of comment lines and correctly writes them back after manipulating the configuration file. It simply lacks proper resource type attributes to enable comment modifications.

Resource Type augeas

Augeas comes with several lenses for parsing different INI-style configuration files. However, there is no generic INI lens. Each lens is designed to parse one specific configuration file. So, for our evaluation, we have decided to use the lens "Koji.lns". The documentation for this lens states, that it supports multi-line values. As this lens is the only one supporting this feature, we have decided to use this one.

A separator character within the value string or a section name is handled correctly by the "augeas" resource type. However, if adding such a separator into the property name, the "augeas" resource type simply ignores this file modification and prints a warning message to the console.

We could not trick the "augeas" resource type and the "Koji.lns" to accept new-line characters in section or property names, values or comment lines. Therefore, it was not possible to insert any unintended line breaks. Either the lens did not accept the passed value or the implementation of the "augeas" resource type issued an error. So we were not able to create an "augeas" resource declaration writing a syntactically incorrect configuration file.

5.3.3 JSON Configuration Files

JSON[1] is lightweight, text-based, language-independent data interchange format with a fully defined syntax [Int13]. Some applications, such as Docker [Doc17], use it also for storing configuration values. Therefore, we will consider this data format in our evaluation, too. Listing 5.3 show a sample JSON file.

```
{
  "object": {
    "setting": "value",
    "array": ["one", "two", "three"],
    "number": 1,
  }
}
```

Listing 5.3: Sample JSON file

The JSON format specification [Int13] does not permit special characters, such as the double quotation ("") or control sequence character inside a string literal. This can be used to test syntactical issues of the evaluated Puppet methods. So if it is possible to place such characters inside a string literal, the configuration file breaks the syntactical rules of the JSON format.

Since Puppet does not provide any native JSON format manipulation resource types, we can only evaluate two methods here: "kdbkey" and "augeas".

Resource Type `kdbkey`

Libelektra provides a JSON parsing storage plugin called "yajl". This storage plugin seems to fully follow the JSON syntax specifications. Adding a double-quotation or control sequence character to either a key name or key value, does not lead to a syntactically incorrect configuration file. Each of these characters are correctly escaped, as defined

[1] JavaScript Object Notation

in the JSON specification. Therefore, the Libelektra "`yajl`" storage plugin cannot be tricked to create a syntactically incorrect configuration file.

Resource Type `augeas`

Augeas comes with a generic "`Json.lns`" lens, parsing and writing JSON configuration files. However, this lens is not very robust. If supplying a new line character within a value string, this new-line character is added directly into the resulting configuration file. Since line breaks inside string literals are not allowed by the JSON specification [Int13], such configuration files are syntactically incorrect.

Further, adding a double-quote character to a value or key name string is not supported, too. However, instead of writing such character directly into the resulting configuration file, the Augeas lens refuses such strings and issues an error message.

5.3.4 Discussion of File Syntax Validation Evaluation

To give a better overview of the above file syntax validation evaluation, we summarizes the evaluation results here.

Table 5.1 shows the results for Hosts file manipulation methods.

manipulation method (resource type)	`kdbkey`	`host`	`augeas`
IP-address validation	yes	yes	no
host name validation	no	partially †	no
alias name validation	no	partially †	no
comment validation	yes	no	yes
invalid syntax possible	**yes**	**yes**	**yes**

† Maximum host name length is not checked.

Table 5.1: Hosts file syntax robustness evaluation

Writing syntactically incorrect Hosts files is possible with all three evaluated resource types. Although, the "`host`" resource type provides good value validation, users have to be aware of using its "`comment`" property correctly. The other two resource types do not provide value validation for all elements.

Table 5.2 summarizes the evaluation for the INI file format.

The "`augeas`" resource type provides the best protection against creating syntactically incorrect INI configuration files. The "`ini`" storage plugin of Libelektra provides some more advanced features, however, as evaluated before, has some implementation problems.

Table 5.3 shows the results for the JSON format evaluation.

manipulation method (resource type)	`kdbkey`	`ini_setting`	`augeas`
line break protection sections	no	no	yes
line break protection properties	no	no	yes
line break protection values	yes	no	yes
line break protection comments	yes	–	yes
separator handling in sections	yes	yes	yes
rejects properties containing separator	no	no	yes
separator handling in values	no †	yes	yes
invalid syntax possible	**yes**	**yes**	no

† supported but leads to unexpected behavior

Table 5.2: INI file format syntax robustness evaluation

manipulation method (resource type)	`kdbkey`	`augeas`
escapes double-quote within string	yes	no
escapes control sequences (e.g. new line)	yes	no
invalid syntax possible	no	**yes**

Table 5.3: JSON file format syntax robustness evaluation

The Libelektra "`yajl`" storage plugin has a robust implementation, which always ensures a correct JSON syntax. The Augeas "`Json.lns`" lens has some problems correctly escaping string values, which leads to syntactically incorrect configuration files.

This syntax validation evaluation clearly shows, that users should not blindly trust in the used resource types to automatically create syntactically correct configuration files. Passing untrusted values to these resources types might break the syntax of the underlying configuration file, leading to unexpected behavior of applications reading these faulty files.

5.4 Runtime Performance

This section evaluates the runtime performance of different file manipulation methods.

5.4.1 Hardware Setup

All benchmarks described below, where executed on a machine with Intel Core 2 Duo (E7400) CPU at 2.80 GHz with 7817 MB RAM and a SSD hard drive (SanDisk SDSS-DHP128G). We measured the read and write throughput of the hard drive with the UNIX tool "`dd`", using the read and write flag "`direct`". The measurement was done

within the benchmark container environment. The hard drive of the used system has a read data throughput of 258 MB/s and write throughput of 240 MB/s.

5.4.2 Benchmark Setup

The benchmark machine runs Ubuntu 16.04.2 LTS. All benchmarks were executed in a Docker [Doc17] container, as also used for the user study tasks 7.3. The setup of the container consists of the following software versions:

- base system: Ubuntu 16.04.1 LTS
- Puppet 3.8.5
- Augeas 1.4.0
- libelektra: current master branch
  (Git sha1: e5624dd2ddb1a9ee09e467a9c65f40f1cea76925)
- Puppet module "`puppetlabs-stdlib`" 4.15.0
- Puppet module "`puppetlabs-inifile`" 1.6.0
- Puppet module "`puppet-libelektra`": current master branch
  (Git sha1: 14ba3022dd7fdac55d0e8bc482879d1385a178a5)

The nature of Docker containers [Doc17] slightly decreases I/O performance, due to the layering of container file system. However, this does not influence the benchmark results, since all benchmarks were executed in the same container environment.

5.4.3 Benchmarks

This runtime analysis consists of four different benchmarks. Each of these benchmarks is run with different Puppet file manipulation methods.

- INI file benchmark
  - resource type `kdbmount` and `kdbkey` with Libelektras' "`ini`" storage plugin
  - resource type `augeas` manipulating a standard configuration file
  - resource type `augeas` manipulating a custom configuration file
  - resource type `ini_setting`
  - resource type `file_line`
  - resource type `file` together with an ERB template

- JSON file benchmark
  - resource type `kdbmount` and `kdbkey` with Libelektras' "`yajl`" plugin
  - resource type `augeas` manipulating a standard configuration file
  - resource type `augeas` manipulating a custom configuration file
  - resource type `file` together with an ERB template
- Hosts file benchmark
  - resource type `kdbmount` and `kdbkey` with Libelektras' "`hosts`" plugin
  - resource type `augeas` manipulating a standard configuration file
  - resource type `augeas` manipulating a custom configuration file
  - resource type `host`
  - resource type `file_line`
  - resource type `file` together with an ERB template

Each benchmark for each variant is executed in the benchmark environment, using the command "`puppet apply <benchmark source file>`". This Puppet command performs the "*compile-catalog*" and "*apply-catalog*" phases in one go. However, we only measure the runtime for the "*apply-catalog*" phase, as this is the time it takes to actually modify the underlying configuration file. The runtime for the "*compile-catalog*" phase is mostly influenced by the size of the Puppet source files. The Puppet agent itself measures the runtime for its "*apply-catalog*" phase and adds the result in precision of hundreds of a second to the console output. We simply use this number as result of our benchmarks.

All benchmarks are performed in the following sequence:

- The target configuration file is deleted to ensure the file is generated from scratch.
- Execute the benchmark and measure the runtime for the "*apply-catalog*" phase. We consider this time as the *creation time*.
- The same benchmark is executed on the previously created configuration file again, whereas this runtime will be considered to be the *update time*.

Each "`kdbkey`" benchmark is executed using the "`ruby`" provider and a second time using the "`kdb`" provider. This will show if the execution of the external Libelektra command "`kdb`" influences the runtime.

The "`augeas`" benchmarks are also executed twice, with slightly different settings. The manipulation of a standard configuration file – files known by Augeas – involves parsing all (Augeas) known configuration files. To measure this parsing overhead, we will run each Augeas benchmark for a standard and a custom configuration file. The benchmark environment consists of 390 configuration files known by Augeas.

INI File Benchmark

Each variant of this benchmark is used to write a INI-style configuration file with 50 sections and 10 settings per section. To accomplish this task, we have to create suitable Puppet source files. Since such a source file may consist of up-to 550 resource declarations we use a simple code generation script to create the source files for our benchmarks.

Listing 5.4 shows the excerpt from the benchmark file for "`kdbmount`" and "`kdbkey`" resource types.

```
kdbmount { 'system/bench1/ini':
  file    => '/etc/bench_ini_kdb.ini',
  plugins => 'ini'
}
Kdbkey { prefix => 'system/bench1/ini' }

kdbkey { "section1/setting1": value => "value_1_1" }
...
kdbkey { "section1/setting10": value => "value_1_10" }
kdbkey { "section2/setting1": value => "value_2_1" }
...
```

Listing 5.4: `kdbmount` + `kdbkey` INI benchmark

Listing 5.5 shows the benchmark file for resource type "`augeas`" using the standard configuration file "`/etc/samba/smb.conf`". The benchmark file for "`augeas`" using a custom configuration file is the same as shown in Listing 5.5, with proper values for the attributes "`lens`" and "`incl`".

```
Augeas { context => '/files/etc/samba/smb.conf' }

augeas { "add section1":
  changes => "set target[0] 'section1'",
  onlyif  => "match *[. = 'section1'] size == 0"
}
augeas { "add section1 setting1":
  changes => "set *[. = 'section1']/setting1 'value_1_1'",
  require => Augeas['add section1']
}
...
```

Listing 5.5: `augeas` INI benchmark

Listing 5.6 shows the benchmark source file for the "`ini_setting`" resource type.

```
Ini_setting { path => '/etc/ini_bench.ini' }
ini_setting {"section1/setting1":
  section => "section1",
  setting => "setting1",
  value   => "value_1_1"
}
...
```

Listing 5.6: `ini_setting` INI benchmark

Listing 5.7 shows the benchmark source file for the "`file_line`" resource type.

```
File_line { path => '/etc/bench_fileline1.ini' }

file_line { "section1": line => "[section1]"}
file_line {"section1/setting1":
  line    => "setting1 = value_1_1",
  after   => "section1",
  require => File_line["section1"]
}
...
```

Listing 5.7: `file_line` INI benchmark

The benchmark for the resource type "`file`" uses a ERB template to generate the complete content of the INI configuration file. This template is evaluated during the "*compile-catalog*" phase and therefore has no influence on the benchmark results. Listing 5.8 shows the template code.

```
<%- Array(1..50).each do |i| -%>
[section<%= i %>]
<%-   Array(1..10).each do |j| -%>
setting<%= j %> = value_<%= i %>_<%= j %>
<%-   end -%>
<%- end -%>
```

Listing 5.8: `file` + ERB template INI benchmark

JSON File Benchmark

This benchmark is very similar to the INI file benchmark, whereas we now write a JSON file containing 50 objects each containing 10 string properties. However, there is no Puppet native resource type for partial modification of JSON files. Also, the "`file_line`" resource type is not used for this benchmark, since the complexity for manipulating a JSON file line based is too high.

The benchmark source file for resource type "kdbmount" and "kdbkey" is basically the same as shown in Listing 5.4. We only changed the parameter for the "kdbmount" declaration (line 2-3) to: "file=>"/etc/bench_json_kdb.json"", "plugins=>'yajl'".

The benchmark source file for both "augeas" variants are slightly more complex. The variant for a standard configuration file is show in Listing 5.9. The variant using a custom configuration file differs for Listing 5.9 by the parameters for the resource default (line 1): "context=>"/files/etc/bench_json_augeas.json"", "lens => "Json.lns"", "incl=>"/etc/bench_json_augeas.json"".

```
Augeas { context => '/files/etc/openshift/quickstarts.json' }
augeas { "add section1":
  changes => [
    "set dict/entry[last()+1] 'section1'",
    "defnode mydict dict/entry[last()]/dict ''" ],
  onlyif => "match dict/*[. = 'section1'] size == 0"
}
augeas { "add section1 setting1":
  changes => [
    "set dict/*[. = 'section1']/dict/entry[1] setting1",
    "set dict/*[. = 'section1']/dict/entry[1]/string 'value_1_1
       '" ],
  require => Augeas['add section1']
}
...
```

Listing 5.9: augeas JSON benchmark

The "file" resource type benchmark uses an ERB template again to create the full JSON configuration file beforehand. Listing 5.10 shows the ERB template code for this benchmark.

```
{
<%- Array(1..50).each do |i| -%>
  "section<%= i %>": {
<%-   Array(1..10).each do |j| -%>
    "setting<%= j %>": "value_<%= i %>_<%= j %>"<%= "," if j !=
        10 %>
<%-   end -%>
  }<%= "," if i != 50 %>
<%- end -%>
}
```

Listing 5.10: JSON benchmark ERB template

Hosts File Benchmark

The aim of this benchmark is to measure the runtime to add or update 250 host entries to a hosts file. Each entry consists of an IP-address, host name and one alias name.

Listing 5.11 shows the benchmark source file for "kdbmount" and "kdbkey" resource types.

```
kdbmount { 'system/network/hosts':
  file    => '/etc/bench_hosts_kdb',
  plugins => 'hosts'
}
Kdbkey { prefix => 'system/network/hosts/ipv4' }
kdbkey {
  "hostname1":            value => "192.168.2.1" ;
  "hostname1/hostalias1": value => "192.168.2.1" ;
}
...
```

Listing 5.11: kdbmount + kdbkey Hosts benchmark

Listing 5.12 shows the benchmark source file for resource type "augeas" using a standard configuration file. Again, the benchmark using "augeas" with a custom configuration file only differs by adding proper values for "lens" and "incl" attributes.

```
Augeas { context => '/files/etc/hosts' }
augeas { "add hostname1":
  changes => [
    "set 0/ipaddr '192.168.2.1'",
    "set 0/canonical 'hostname1'",
    "set 0/alias 'hostalias1'" ],
  onlyif => "match *[canonical = 'hostname1'] size == 0"
}
...
```

Listing 5.12: augeas Hosts benchmark

Listing 5.13 shows the benchmark source file using the "host" resource type and Listing 5.14 for resource type "file_line".

```
Host { target => '/etc/bench_hosts_host' }
host { "hostname1":
  ip           => "192.168.2.1",
  host_aliases => "hostalias1"
}
...
```

Listing 5.13: host benchmark

```
File_line { path => '/etc/bench_hosts_fileline' }
file_line { "host hostname1":
  line  => "192.168.2.1  hostname1 hostalias1",
  match => "  hostname1 "
}
...
```

Listing 5.14: `file_line` Hosts benchmark

The benchmark for the resource type "`file`" again uses an ERB template to generate the content of the required Hosts file. Listing 5.15 show this ERB template.

```
<%- Array(1..250).each do |i| -%>
192.168.1.<%= i %>  hostname<%= i %> hostalias<%= i %>
<%- end -%>
```

Listing 5.15: Hosts benchmark ERB template

5.4.4 Benchmark Results and Discussion

Table 5.4 presents the measured file creation and file update runtime for each executed benchmark.

resource type	INI benchmark		JSON benchmark		Hosts benchmark	
	create	update	create	update	create	update
`kdbkey` ruby provider	46.76	50.28	6.43	3.85	10.92	5.22
`kdbkey` kdb provider	103.85	168.81	34.92	33.91	39.43	38.19
`augeas` standard file	393.07	362.02	412.78	369.69	167.73	154.84
`augeas` custom file	50.20	20.34	67.32	24.63	12.22	3.53
`ini_setting`	2.91	3.27				
`file_line`	3.49	2.04			1.23	0.79
`host`					1.08	0.79
`file`	0.97	0.95	1.09	1.11	0.47	0.47

file create and update times in seconds
empty cells indicate that there is no evaluation possible

Table 5.4: Runtime performance benchmark results

This benchmark shows, that the runtime performance of "`kdbkey`" is heavily influenced by the used Libelektra storage plugin. The "`yajl`" plugin performs much better than the "`ini`" plugin, whereas the "`hosts`" plugin is nearly as good as the "`yajl`" plugin.

The runtime performance of *kdbkey's* "`kdb`" provider, as already expected, is much worse than the runtime of the "`ruby`" provider. For each created key, the "`kdb`" provider executes two external commands: one for querying the current value and one for setting

the desired value. Since the execution of an external command is an expensive operation, the runtime performance for this provider is low.

Even more noticeable is the dramatic runtime difference between the two different Augeas runs. As already noticed, the "`augeas`" resource type, if used for standard configuration files, parses all known configuration files beforehand, which has a big impact on the overall runtime. So, if runtime performance matters, Augeas users should specify a specific file and lens.

An interesting aspect is the good runtime performance of the resource types "`ini_setting`" and "`host`". The provider classes for these two resource types utilize functionality from the same parent class, called "`ParsedFile`". This class is a generic file parser, which parses each configuration file only once for all resource type declarations. Therefore, these two resource types do not have the overhead of parsing the same file for each resource type declaration separately, which results in this good runtime performance.

CHAPTER 6

Case Study

The proposed solution on its own does not bring any benefit to system administrators as long as our solution is not suitable to manage real computer systems. This chapter aims to show the possibility to manage any kind of configuration file on a real computer system with the methods of "`kdbmount`" and "`kdbkey`" only.

6.1 Goals

We first formulate a research question.

Research Question 1. *When using the proposed solution in a real-world scenario, is it possible to manage all required configuration files with the methods of the proposed solution? Are there any trade-offs necessary?*

The term *'real-world scenario'* here means that we want to manage a computer system, which is a common candidate for using configuration management solutions. Humble and Farley states in *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* [HF10] that managed environments are a key factor for continuous software deployment strategies. This applies to production environments but also to enabling systems such as the continuous integration systems itself. Therefore, we have chosen a similar system as our real-world scenario, involving a productive web-server and a continuous integration service. Section 6.2 describes this system in more detail.

By the term *'all required configuration files'* in research question 1 we mean, that we want to use the proposed solution for all configuration files, that have to be manipulated in order to realize our chosen real-world scenario system. Unmanaged (unmodified) configuration files are not in the scope of this case study.

In order to answer research question 1, we implemented appropriate Puppet modules to mange a computer environment with the use of or "`kdbmount`" and "`kdbkey`" resource types.

6.2 Real-World Scenario: Web-Server and Continuous Integration System

Our target system consists of the following services:

- A continuous integration service, which builds and tests software for a single project. This service consists of a continuous integration server and several build slaves systems, which are used to run the actual build jobs.
- The Elektra Snippet-sharing REST backend, which implements the storage backend for the Elektra Snippet-sharing service [1].
- A productive Web-server, hosting a project webpage and acting as a front end for other HTTP services, such as the continuous integration server and the Elektra Snippet-sharing service.

This system is realized for the Libelektra Project. Therefore, the continuous integration server will build and test the Libelektra sources in a continuous manner. The Web-server will be used to serve the Libelektra project webpage and all web related DNS subdomains for *libelektra.org*.

An overview of this system setup is shown in Figure 6.1.

The Web-server is realized by the *Apache Httpd* Web-server[2]. This Web-server is hosting the Libelektra project webpage, the Elektra Snippet-sharing frontend, API-documentation and different DNS subdomain redirects for *libelektra.org*. In addition, this Web-server is used as reverse proxy for the continuous integration server.

The software *Jenkins*[3] realizes the continuous integration service. This continuous integration service is split up in a build server and several build slaves. The build server is capable of managing all build jobs, whereas the build job is executed on one of the build slaves. This architecture enables us to run build jobs on different target platforms. A build slave is a separate computer system, either a real system or virtualised system. Our setup uses Linux-VServer[4] to run multiple build slaves on one real system. The build server, also called build master, is connected with its build slaves via and SSH

[1] `https://master.libelektra.org/doc/tutorials/snippet-sharing-rest-service.md`

[2] `https://httpd.apache.org/`

[3] `https://jenkins.io/`

[4] `http://linux-vserver.org/`

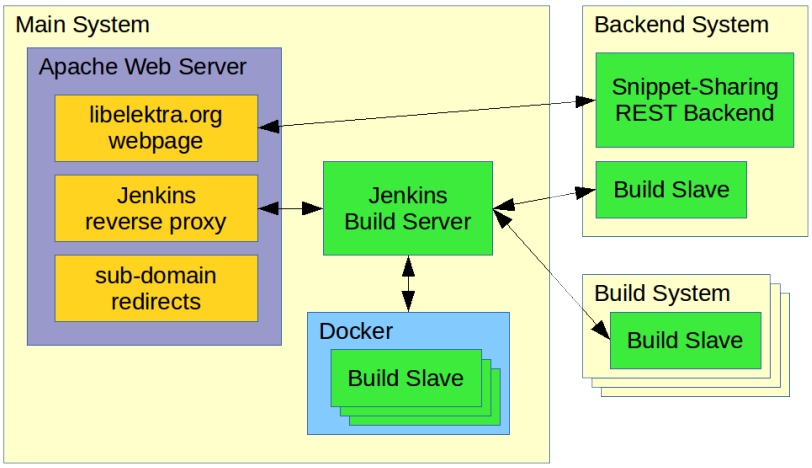

Figure 6.1: Case study system overview

connection. This connection is used to transfer the Jenkins build agent software to the build slave system and for communication with the build agent.

The Elektra Snippet-sharing REST backend is separated from the main system for security reasons. Therefore, it is running on the backend system. The Snippet-sharing frontend is served by the Apache Web-server. The frontend communicates with the backend using a REST API. Additionally, the backend system runs a Jenkins build slave to facilitate an automatic REST backend build once its source code was updated.

The main system runs a *Debian 8 Jessie* operating system. The build slaves are also based on Debian, whereas we currently integrate three different variants: Debian 7 Wheezy, Debian 8 Jessie and Debian 9 Stretch. Our backend system runs the Debian 9 operating system.

The following subsections describe the required configuration changes for each system in more detail.

6.2.1 Apache Web-Server

The setup of the Apache Web-server involves the following parts:

- Installation of the Apache software packages from the main Debian repositories.
- Manage the site-specific Apache configuration files.
- Enable/disable site-specific configuration files.
- Enable/disable Apache module-specific configuration files.
- Start/stop/refresh the Apache system service.

This involves the management of several Apache specific configuration files. Apache uses its own custom configuration file format [Fou17a], which is a mixture of XML-like configuration sections and configuration settings in the form of simple key-value pairs. Other configuration files and formats are not involved.

6.2.2 Jenkins Build Server

The Jenkins build server setup involves the following steps:

- Adding the Jenkins Debian repository URLs to the Debian repository list.
- Installation of the Jenkins Debian package.
- Manage the Jenkins system service configuration file.
- Manage several Jenkins configuration files.
- Create the Jenkins system account.
- Start/stop the Jenkins system service.

The list of Debian package repository URLs are defined by several configuration files with a custom configuration format.

The Jenkins system service configuration file is a basic shell script, containing shell variable assignments only. Therefore, we can consider the format of this configuration file as a simple key-value format with certain restrictions: no white space around the assignment operators allowed, values should be enclosed by double quotes.

The Jenkins configuration itself consists of a set of XML configuration files. It is important to note, that we do not plan to fully automate the whole Jenkins configuration, like creating Jenkins users and build jobs. For the moment we prefer to manage these settings with the Jenkins web interface. Only global Jenkins configuration settings, which usually only are manipulated during initial setup time, are in the scope of this case study.

6.2.3 Jenkins Build Slaves

The management of a Jenkins build slave system involves the following parts:

- Install packages required to run the Jenkins build agent. This includes an SSH-server and the JAVA runtime.
- Create the Jenkins system account.
- Setup an SSH "`authorized_keys`" configuration file to allow key based authentication of the Jenkins build server.

- Configure basic Git settings, such as "`user.name`" and "`user.email`". This is required to allow the build agent to perform Git checkouts and create Git tags.
- Install packages and software required to correctly build the Libelektra software.

This Puppet setup involves two different configuration files with different formats. The SSH "`authorized_keys`" file contains a list of public SSH keys and names, which are allowed to establish an SSH connection without entering a password. The second configuration file is the user global Git configuration file, which uses a INI-style format with sections. Other system configuration files are currently not in the scope of this case study.

6.2.4 Docker Container Runtime

The container technology Docker [Doc17] is used to facilitate more flexibility in handling different build platforms. Therefore, we have to install and configure the Docker software on the main system, which involves the following steps:

- Adding the Docker Debian repository URL to the Debian repository list.
- Install the Docker package.
- Management of the Docker daemon configuration file.
- Start/Stop of the Docker system service.

The Docker repository URL has to be added to the list of Debian repositories in the same way as for the Jenkins repository URL.

The Docker daemon uses a single JSON-based configuration file.

6.2.5 Elektra Snippet-Sharing REST Backend

The Elektra Snippet-sharing REST backend, being part of the Elektra initiative, is installed from its source code on the backend system. Therefore, in addition to configuring this application, we also have to make sure its compile and runtime dependencies are installed. So managing this application is done with the following steps:

- Install compile and runtime time dependencies.
- Configure the REST backend application.

Since the Elektra Snippet-sharing REST backend is part of the Elektra initiative, this application uses Libelektra for reading the configuration. Therefore, it does not make use of a special configuration file format.

6.3 Implementation Details

This section describes the implementation details of the Puppet code pieces realizing our real-world scenario system as described in the previous section.

6.3.1 Puppet Source Code Structure

Our Puppet source code for this case study consists of three main parts:

- Puppet modules implementing the functionality to realize our required systems.
- A Puppet site manifest containing a basic node definition and class assignments.
- A Hiera data store, containing all project related data, which should not be part of the module implementation.

This three-folded structure is typical for Puppet source code and further described in Section 2.2.3.

In addition to the self-written modules, we use of the modules "`puppet-libelektra`" (proposed solution) and "`puppetlabs-stdlib`" for adding additional functions, such as "`has_key`" or "`ensure_resources`".

The following Puppet modules were developed to realize the described functionality:

- "`apache`": generic Apache Web-server handling and configuration
- "`aptrepo`": management of Debian repository lists
- "`buildmaster`": base module for the Jenkins build master
- "`buildslave`": management of build slaves
- "`docker`": Docker installation and configuration
- "`homepage`": defines the structure of *libelektra.org* webpages, sub-domain redirects and reverse proxy configuration
- "`jenkins`": Jenkins build server management
- "`homepagebackend`": Elektra Snippet-sharing REST backend management

Implementation details for each of these modules is described in the subsequent sections.

The "`manifests`" directory contains a single "`site.pp`" manifest, which defines a single node definition for the main system. For build slaves no such manifest exists.

The "`hiera`" directory holds a single "`site.yml`" file. This file defines all class parameter values for our setup and contains the following information:

- Jenkins configuration settings
- Docker configuration settings
- package lists and custom commands to setup and configure a build slave
- SSH public key for build slaves

The following subsections describe each of the developed modules in more detail. We will focus on the configuration file handling parts.

6.3.2 Module: "apache"

The "apache" module is designed to install, configure and manage the Apache Web-server. The module consists of the following classes and defined types:

- class "apache": main module class including all other subclasses.
- class "apache::install": manage the installation of the Apache Web-server.
- class "apache::service": manage the Apache system service.
- defined type "apache::module": enable/disable Apache modules for Debian style Apache configurations.
- defined type "apache::site": manage Apache site configurations for Debian style Apache configurations.
- defined type "apache::reverseproxy": allows us to define an Apache "vhost" site, implementing a reverse proxy functionality.

The Apache HTTP server uses a custom configuration file syntax for its configuration files. The Debian style Apache configuration is split up in several pieces: a main configuration file and several modules and site configuration files. The module and site configuration files are included by the main configuration file via symbolic links. These symbolic links are manged by several scripts such as "a2enmod" or "a2ensite". This allows system administrators to selectively include or exclude module and site configurations without modifying any configuration files. The two defined types "apache::module" and "apache::site" allows us to manage this behaviour with Puppet methods and are just calling the appropriate "a2xx" scripts.

Apache configuration file manipulation is implemented by the defined type "apache::reverseproxy". Since Libelektra does not have a storage plugin for the Apache configuration file format, a very first version of this class used the Libelektra Augeas plugin together with the Augeas lens "Httpd.lns" to define and manipulate an Apache configuration file. However, we identified two main problems with this strategy.

The first problem arose, when we tried to define a new configuration file from scratch. Augeas was not built to parse non-existing or empty files and therefore always issued and error when we tried to define a new file. However, this problem could be solved by a simple workaround: create the configuration file with some initial content before Augeas tries to parse it. The file creation was realized by a simple "`exec`" resource type declaration, which was only executed if the target file was missing.

The second issue with the Augeas storage plugin strategy comes from Augeas internal Apache configuration representation. Augeas' "`Httpd.lns`" lens transforms each Apache configuration directive in two nodes: a node called "`directive`" with the directive name as value and a sub-node named "`arg`" with the directive value as node value. Thus, to define a not existing directive, we have to create two nodes during one Augeas parsing cycle. Otherwise, Augeas is not accepting the incomplete directive setting and refuses to write the configuration file. However, the current implementation of the "`kdbkey`" resource type is not able to define more than one Libelektra key with one "`kdbkey`" resource declaration, leading to exactly one managed Augeas node per "`kdbkey`" resource instance. Since each "`kdbkey`" resource declaration involves a separate Libelektra parsing/writing cycle, we are not able to create a new Apache directive definition with the resource type "`kdbkey`" with this Libelektra Augeas strategy.

Since there is currently no workaround for the second Libelektra Augeas issue, we are not able to use this strategy.

Currently the only way for defining a new Apache configuration file with "`kdbkey`" resource type declaration and therefore Libelektra methods is by using the "`line`" storage plugin. This Libelektra plugin facilitates line-wise configuration file manipulation. The whole file is transformed line by line into an Libelektra array. With the help of the array functionality of the "`kdbkey`" resource type, a whole configuration file can be managed with only one "`kdbkey`" resource type declaration. An example of such configuration file definition is show in Listing 6.1.

```
kdbkey { 'system/sw/apache/sites/jenkins_proxy':
  value => [
    "<VirtualHost *:80>",
    "  ServerName build.libelektra.org",
    ...
    "</VirtualHost>"
  ]
}
```

Listing 6.1: Manage a configuration file with the "`line`" storage plugin

Although, this is a working and simple solution, it should be avoided, since it does not give you syntactical abstraction. Instead, we manage the complete content of a configuration file.

6.3.3 Module "aptrepo"

The purpose of the "aptrepo" module is to manage the Debian package repositories list and to update the Debian package cache. The module is used by the "jenkins" and "docker" modules to manage the Jenkins and Docker Debian repository entries. This module consists of the following classes and defined types:

- class "aptrepo": main module class, currently empty and only exists by convention.
- class "aptrepo::update": executes the command "apt-get update" if required.
- defined type "aptrepo::repo": manages a single Debian package repository list entry. Additionally, this defined type installs the repository's GPG key, by fetching it from the internet.

The configuration related parts of this module are located in the defined type "aptrepo::repo". The Debian repository list is defined by the content of the configuration file "/etc/apt/sources.list" and the concatenation of all files under "/etc/apt/sources.list.d/". Thus, this repository list can be split up in different files. These configuration files use a custom configuration format.

Libelektra does not have a storage plugin to parse and write such repository list files, but there is an Augeas lens for such configuration files, called "Aptsources.lns". Therefore, we can mount these configuration files with the Libelektra Augeas plugin utilizing the "Aptsources.lns" lens.

However, this strategy has the same problems as for Apache configuration files. Non-existing entries cannot be created, as this requires defining three different Augeas nodes within one Augeas parsing cycle. In addition to that, a new configuration file is not created by Augeas, which leads to errors when defining the first entry for a new configuration file.

But, since each repository entry consists of a single line only and we can split up the definition of repository list entries in several files we can use a workaround for both problems. Before mounting the configuration file with the Augeas plugin, we check if the file already exists. If not, the file is created with an initial repository entry. This initial entry is updated by several "kdbkey" resource type declaration. This way, we can manage a single repository entry with "kdbkey" resource type declaration. Listing 6.2 shows the sources for this strategy.

Although, this strategy is working, it is considered to by an 'ugly hack' and should be avoided.

```
exec { "echo 'deb http://invalid.tdl dist' > $sources_file":
  path    => $::path,
  creates => $sources_file,
} ->
kdbmount { $mountpoint:
  file    => $sources_file,
  plugins => { 'augeas' => { 'lens' => 'Aptsources.lns' } },
}

kdbkey {
  "$mountpoint/1/type":         value => "deb" ;
  "$mountpoint/1/uri":          value => $url ;
  "$mountpoint/1/distribution": value => $distribution ;
}
```

Listing 6.2: Manage a single repository entry with "kdbkey" and the Augeas plugin

6.3.4 Module "buildslave"

The "buildslave" module is used to manage a Jenkins build slave system. It sets up a preliminary environment to allow the Jenkins build master to connect to this system and to start building Libelektra on it. The module consists of the following classes:

- class "buildslave": main module class. It provides configuration parameters for this module and includes all other sub classes.
- class "buildslave::customcommands": This class is used to enable the execution of arbitrary commands once all required packages are installed. It is used to facilitate software installation not found in Debian repositories.
- class "buildslave::packages": This class is used to install preliminary packages required to run the Jenkins build agent. Additionally, it installs all packages, defined by the parameter "packages" of the class "buildslave".
- class "buildslave::user": This class manages the system account for the Jenkins build agent. In addition, it is used to set up the SSH authorized keys and Git configuration for this user account.

The only configuration-related parts in this module are located in the class "buildslave::user". This class manages two different configuration files of different configuration formats.

The configuration file ".ssh/authorized_keys" contains a list of public SSH keys, for which key-based authentication is allowed and uses a custom configuration file format.

To enable the management of such configuration file formats, we have written a Libelektra storage plugin, called "ssh_authorized_keys", which is implemented in Ruby. This plugin transforms each key line in ".ssh/authorized_keys" file into several keys representing this key entry. The line in Listing 6.3 is transformed to the keys shown by Listing 6.4.

```
ssh-rsa AAAAB3NzaC1yc2EA....331B1TMiN jenkins@build
```

Listing 6.3: Example SSH authorized_keys line (shortend for readability)

```
.../jenkins@build = AAAAB3NzaC1yc2EA....331B1TMiN
.../jenkins@build/type = ssh-rsa
```

Listing 6.4: Libelektra key representation of example SSH authorized_keys line

Since this key line is represented by two different Libelektra keys, we might face the problem of requiring setting two keys at the same time to generate a valid SSH authorized_keys line. However, the "ssh_authorized_keys" plugin uses a default value for the key "<name>/type" set to "ssh-rsa".

Listing 6.5 shows an example of this method.

```
Kdbkey {
  user => $buildslave::user,
}

$mountpoint = 'user/ssh/authorized_keys'
kdbmount { $mountpoint:
  file    => '/.ssh/authorized_keys',
  plugins => { 'ruby' => { 'script' => $ssh_auth_plugin } },
}

kdbkey {
  "$mountpoint/$keyname":      value => $key ;
  "$mountpoint/$keyname/type": value => $keytype ;
}
```

Listing 6.5: Sample definition of an SSH authorized_keys entry

Listing 6.5 shows an important aspect of an SSH authorized_keys file. Such files are located in the context of a particular user account. Therefore, the file is mounted into the Libelektra user namespace. Additionally, to that, the "kdbkey" resources are defined with the attribute "user", set to the Jenkins build agent user account. Thus, the SSH key is added to the SSH authorized_keys file of the specified user.

Additional to the SSH authorized_keys configuration file, the class "buildslave::user" manages the Git configuration file for this user account. Since this configuration file is a standard INI-style configuration file format, we just make use of the Libelektra "ini" plugin to manage configuration values.

6.3.5 Module "docker"

The "docker" module is designed to install and configure the Docker runtime and consist of the following classes and defined types:

- class "docker": main module class. Provides basic module parameter and includes all other sub classes.
- class "docker::install": install the Docker package.
- class "docker::install::repo": define the Docker Debian repository list entry.
- class "docker::config": mounts the Docker daemon configuration file.
- defined type "docker::config::daemonsetting": defines a single configuration setting with the Docker daemon configuration file.

This module uses a data-driven configuration approach. Aim of this approach is to put all configuration settings into the Hiera data store. The Puppet module only provides functionality to write arbitrary configuration settings into the correct configuration file. This approach works by using a special Puppet function called "ensure_resources". This function enables us to define multiple resources of a specific type, by passing a hash containing parameters and their settings. Listing 6.6 shows the relevant sources.

```
# file: docker::daemonsetting
define docker::daemonsetting($value, $ensure = 'present') {
  kdbkey { $name:
    ensure   => $ensure,
    prefix   => 'system/sw/docker/daemon',
    value    => $value,
  }
}

# file: init.pp
class docker($daemon_config  = undef, ...) {
  if $daemon_config {
    ensure_resources('docker::daemonsetting', $daemon_config)
  }
  kdbmount { 'system/sw/docker/daemon':
    file     => '/etc/docker/daemon.json',
    plugins  => 'json'
  }
  ...
}
```

Listing 6.6: Data driven configuration definition

The class "docker::config" mounts the Docker configuration file with the "json" plugin. The defined type "docker::config::daemonsetting" simply defines a single "kdbkey" resource with the "prefix" parameter pointing to the used mount point. The name of the defined type instance is used as key name and the "value" parameter is simply passed to the "kdbkey" resource declaration. These two pieces are tied together in the main "docker" class. The class "docker::config" is included, thus the configuration file will be mounted once the "docker" class is used. If we pass a hash containing "docker::config::daemonsetting" resource values to the parameter "$daemon_config", the function "ensure_resources" will be called with this hash. This call will create "docker::config::daemonsetting" instances as defined by the passed hash "$deamon_config".

This approach enables us to define arbitrary Docker configuration settings without modifying the module code of "docker". All configuration settings can be defined by passing an appropriate hash to the parameter "docker_config". An example for such a hash is shown in Listing /reflst:cs:dd-hash.

```
{
  "insecure-registries" => {
    value => ["localhost", "127.0.0.1"]
  },
  "other-setting" => {
    ensure => "absent"
  }
}
```

Listing 6.7: Sample hash for parameter "docker_config" (Puppet syntax)

Values for class parameter are defined in the Hiera data store. An example is shown in Listing 6.8.

```
docker::daemon_config:
  insecure-registries:
    value:
      - 127.0.0.1
      - localhost
```

Listing 6.8: Hiera values for data driven configuration approach

This piece of Hiera data together with the data-driven configuration approach of the "docker" module defines one configuration setting ("insecure-registries") with entries for the configuration file "/etc/docker/daemon.json".

6.3.6 Module "homepage"

This module is used to define all Apache site configuration files required to server the *libelektra.org* website and consists of the following classes and defined types:

- class "homepage": main module class, containing other sub classes, defined type instances of "homepage::redirectvhost" for the webpage and an instance of "apache::reverseproxy" for Jenkins reverse proxy setup.
- class "homepage::mainpage": contains the Apache configuration for the *libelektra.org* webpage.
- defined type "homepage::redirectvhost": contains a Apache "vhost" configuration for a single sub-domain redirect.

The module only includes Apache site configuration settings for all *libelektra.org* webpages and sub-domain redirects. The defined type "homepage::redirectvhost" and the class "homepage::mainpage" both us the same "line" plugin-based strategy for defining an Apache site configuration file as already described in Subsection 6.3.2.

6.3.7 Module "jenkins"

The module "jenkins" is used to install, configure and manage the Jenkins build server. This module consists of the following defined types and classes:

- class "jenkins": main module class, providing module parameter and including all other subclasses.
- class "jenkins::install": installs the Jenkins package.
- class "jenkins::install::repo": adds the Jenkins Debian repository list entry.
- class "jenkins::service": manages the Jenkins system service.
- class "jenkins::user": manages the Jenkins system account.
- defined type "jenkins::configfile": mounts a single Jenkins configuration file.
- defined type "jenkins::config": manages a single Jenkins configuration setting.
- class "jenkins::serviceconfig": manages the Jenkins service configuration file.

As already described in Section 6.2.2, the Jenkins configuration consists of a set of XML configuration files. The "jenkins" module follows the same data driven configuration approach as the "docker" module (Section 6.3.5). Additionally, this concept is extended to the Jenkins configuration files itself. Therefore, the set of mounted configuration files are also defined within the Hiera data store. For mounting the XML configuration files, we use the Libelektra "xerces" plugin.

In addition to the Jenkins XML configuration, the Jenkins service script uses its own configuration file, which is a simple shell script containing only variable assignments. To manage this file, we have implemented an Libelektra plugin, called "`shellvars`" and is also written in Ruby. If simply transforms the variable assignments to key-value pairs, which then can be manipulated with the "`kdbkey`" resource type.

We decided to implement the "`shellvars`" plugin, since the "`simpleini`" plugin does not preserve the order of the key entries and the "`ini`" plugin is not able to use the assignment operator without spaces. We also tried the Augeas plugin with an appropriate lens. While reading worked fine, Augeas issued errors once we tried to manipulate certain settings.

6.3.8 Module "`homepagebackend`"

The module "`homepagebackend`" is designed to manage the Elektra Snippet-sharing REST backend application. It consists of one Puppet class only ("`homepagebackend`"), which is used to perform both required tasks at once:

- Installation of all required packages to compile and run the backend application.
- Configuration of the backend application, such that it integrates with our other services, notably the Elektra Snippet-sharing frontend.

We have already described in Section 6.2.5 that the REST backend application uses Libelektra to read its configuration. Therefore, we do not have to take care of the used configuration file format and the used Libelektra mount point. Instead, we simply manipulate the Libelektra keys read by the application, everything below "`/sw/elektra/restbackend/#0/current/backend`". The REST backend application comes with a configuration specification, which provides Libelektra key specifications for all its configuration settings, i.e. allowed values and a description for each possible configuration key. So in our Puppet class file we simply define "`kdbkey`" resources to configure all required configuration settings for the REST backend application. No "`kdbmount`" resource definitions are required here.

6.3.9 Bootstrapping

In order to manage our computer systems with the Puppet modules of this case study, we have to preconfigure this computer system beforehand. At the beginning we have to install the Puppet agent itself on the managed computer system. On Debian 8 and 9 this is done by installing the "`puppet`" package from the Debian repositories. On Debian 7 we have to install the Puppet agent from the *puppetlabs.com* package repositories, since the version in the Debian 7 repositories is too old.

In addition to the Puppet agent, we also have to install Libelektra. Normally, this can be achieved by installing the appropriate Debian package, too. However, since our Puppet

module code makes use of the "`ruby`" plugin, we have to compile and install the latest version. This requires to install all Libelektra build dependencies beforehand.

6.3.10 Execution

Once Puppet and Libelektra are installed, we can start applying our Puppet code. In this case study we do not use a Puppet master to manage our systems. Instead, we simply use the "`puppet apply`" command to compile and apply our Puppet code on the managed system. Therefore, before we can start, we have to bring the written Puppet code onto the managed system. This is done by cloning the appropriate Git repository and installing all required files under "`/etc/puppet`".

As already mentioned, the main system, running the Apache Web-server and the Jenkins build server, is based on Debian 8 and uses Puppet version 3.7.2. The same applies to the Debian 8 based build slave system. The Debian 9 based build slave system and the backend system already comes with Puppet version 4.8.2. On the Debian 7 based build slave system we have installed the Puppet version 3.8.7.

On the main system we started the Puppet agent with site manifest file. For the build slave systems we used the command "`puppet apply -t 'include buildslave'`" to run the Puppet agent.

On all four systems, one main system and three build slave systems, the Puppet agent completed successfully and configured the systems as desired. The runtime of each initial Puppet invocation was between one and five minutes and mainly depends on the package download speed.

6.4 Discussion

Based on this case study implementation we are able to answer research question 1: The proposed solution can be used in a real-world like scenario, however there are trade-offs necessary.

We designed this case study in such a way, that it involves the configuration manipulation of different sorts of applications, which can be categorized in the following way:

1. Applications using Libelektra as configuration framework.
2. Applications using a configuration file format for which Libelektra has a storage plugin.
3. Applications using a configuration file format for which Libelektra does not have a storage plugin.

Our proposed solution works best for applications of category one, since the same underlying library is used as configuration framework. We did not have to take care

about the actual configuration file format and did not have to specify any Libelektra mount point in order to configure such an application. We simply were able to specify all required configuration settings as "`kdbkey`" resources. If the configured application provides a Libelektra configuration specification in addition, a system administrator benefits from configuration validation automatically, without additional effort. In this case study the Elektra Snippet-sharing REST backend was the only application utilizing Libelektra.

Applications of category two, i.e. applications using a configuration file format supported by Libelektra, are also manageable with the proposed solution without much effort. We only had to specify the Libelektra mount point beforehand, using the "`kdbmount`" resource type. Once this was done, all configuration settings were manageable using the "`kdbkey`" resource type. In contrast to applications of category one, it is unlikely that application of this category provide a Libelektra configuration specification. Therefore, if one wants to benefit of Libelektras' validation functionality, suitable specifications for the used keys have to be provided. In this case study, we configured the following applications, which use a Libelektra supported configuration file format:

- Jenkins global configuration: XML-based; Libelektra plugin: "`xerces`"
- Git user configuration: INI-style; Libelektra plugin: "`ini`"
- Docker daemon configuration: JSON; Libelektra plugin: "`yajl`"

For applications of category three, i.e. applications using a configuration format not supported by Libelektra, we used different strategies which allowed us to manage configuration files of this applications.

In two situations we facilitated from Libelektras' extensible architecture and implemented a storage plugin for our use cases. This was done for SSH authorized keys files and for the Jenkins service configuration files. Technically we could have used Libelektras' "`simpleini`" storage plugin to handle the Jenkins service configuration file. However, this has some limitations, such as non order preserving and no support for comment lines. So we decided to implement a storage plugin using the "`ruby`" plugin. This shows, that we are able to integrate any other not supported configuration file format, as long it can be mapped onto Libelektras' configuration space.

Technically it should be possible to manage any configuration file format Augeas is able to handle with the Puppet resource types of our proposed solution, since Libelektra features a "`augeas`" plugin. However, this case study showed, that this approach is not feasible for real-world scenarios. While modifying single settings using the proposed solution with the "`augeas`" plugin is no problem, our solution showed weaknesses when it comes to defining new structural elements. In most cases this has to be done by defining multiple new Libelektra keys simultaneously, i.e. in one write cycle. But our solution can only define one Libelektra key at a time.

For this reason we could not use Libelektras' "augeas" plugin to manage the Apache configuration files, although Augeas comes with lenses for this configuration file format. As fallback solutions we used Libelektras' "line" storage plugin and the array feature of our solution to manage the Apache configuration files line-wise. However, this method is quite similar to the complete-content approach (see Sections 2.4.1-2.4.3) and performs no syntax and value validation. The approach of the "aptrepo" module implements a simple workaround for this Augeas problem, however this is not applicable for all cases.

In conclusion, the main problem is a result of the absence of appropriate Libelektra storage plugins. This was shown by implementing two storage plugins for the SSH authorized keys file format and the shell variable assignment files. Once Libelektra is able to correctly parse and write the configuration file format, our solution can be used to manage the underlying configuration file on a key-value basis.

CHAPTER 7

User Study

In order to evaluate the advantages of the different methods of configuration file handling from a users point of view, a user study was conducted.

A group of users was asked to solve some predefined system configuration tasks, in the form of Puppet programming exercises, by applying different Puppet concepts for handling configuration files.

7.1 Goals

This user study was designed to find an answer to research question 2.

Research Question 2. *Which approach of configuration file manipulation provides the best results if compared in terms of usability and maintainability? Does the proposed solution have an advantage over existing methods?*

Kitchenham et al. states in their work [Kit+02], that an empirical research requires a good research question, from which the method and evaluation can be derived. Therefore, we formulate more concrete sub research questions, which will help us to answer the more general one. The derived research questions will be based on the following thesis: An enhanced usability will reduce development and maintenance time. From this we formulate our sub resource questions as follows:

Sub Research Question 2.1. *When used in scenarios, which require a full definition of a configuration file, can the proposed solution help to reduce development time when compared with existing methods?*

Sub Research Question 2.2. *When used in scenarios, which require partial modification of configuration files, can the proposed solution help to reduce development time when compared with existing methods?*

Sub Research Question 2.3. *Can key-value-based configuration manipulation methods, such as the proposed solution, help to reduce the time spent for maintenance tasks?*

Based on these sub research questions we can now design our user study experiments for the following scenarios:

- define the content of a configuration file from scratch
- partial configuration file manipulations
- maintenance of Puppet code for manipulating configuration files

7.2 Experiment Design

To find an answer to our research questions, we defined four different Puppet programming tasks.

- Task 1: define a JSON based configuration file from scratch for a fictional program
- Task 2.1: manipulation of a Hosts file
- Task 2.2: manipulation of the Samba configuration file (INI based)
- Task 3: extend an existing Puppet module for a fictional application

In order to measure the developer productivity impact of the different Puppet resource types, each task has to be solved using a predefined Puppet file manipulation method. This method is considered to be the experiment factor, i.e. the changed independent variable [Woh+00].

This experiment factor has up to three treatment values for each task:

- Factor treatments for Task 1 and 3:
  - Method ERB: resource type `file` together with an ERB template
  - Method KDB: resource types `kdbkey` and `kdbmount`
- Factor treatments for Task 2.1:
  - Method HOST: resource type `host`
  - Method AUG: resource type `augeas`
  - Method KDB: resource types `kdbkey` and `kdbmount`
- Factor treatments for Task 2.2:

- Method INI: resource type `ini_setting`
- Method AUG: resource type `augeas`
- Method KDB: resource types `kdbkey` and `kdbmount`

The experiment is designed to be an intra-subject experiment, i.e. each subject solves the same task several times, using different methods. The main advantage of this approach is, that this increases the statistical power of the analysis with the same number of participants and makes it easier to measure the impact of the experiment factor. However, this approach requires to have short programming tasks. Otherwise, the total duration of the experiment is too long [Pre00].

We have to take special care about the order in which the different task methods are solved by the experiment subjects, since the order can have a significant impact [Pre00]. Therefore, subjects are assigned to one of four groups, whereas each group defines a specific order in which the different methods for one task are solved. The order of the tasks was not altered, so all groups start with Task 1.

The dependent variable in our experiment is the development time, i.e. the time required to solve the task using a defined method. So by altering the experiment factor, we measure the productivity impact of this Puppet file manipulation method. To verify if the productivity differences between two methods is significant, we use two hypothesis tests, the paired T-test [McD15a] and the Wilcoxon signed rank test [McD15b] with a significance level $\alpha = 0.05$.

For each task we have defined a set of automated test cases, the subject does not have access to. A task is considered to be completed if all defined test cases for this particular task have passed.

Before the experiment was started, the subjects had to fill up a questionnaire capturing general information about the subject and experiences with Puppet, software development and other related tools. After each solved task variant, the subject had to fill up another questionnaire to capture a subjective usability rating of the used Puppet method. The questionnaire used a predefined usability rating scale from 1 (very good) to 5 (very bad).

7.3 Experiment Environment and Measurement

When designing a software experiment, it is important to consider all influencing variables of this experiment and try to control most of them. This allows us to measure the effect of one explicitly changed variable (the experiment factor) [Woh+00]. Therefore, the experiment environment was designed to control as many influencing variables as possible:

- Same source editor for all participants: We used the Atom[1] text editor in version 1.15.0 with Puppet syntax highlighting and automatic syntax checking.

[1] https://atom.io/

- A new system environment for each task method: Puppet agent executions of earlier programming tasks do not have any influence on later Puppet agent executions.
- Puppet language and resource type guide: A 14-page-long Puppet introduction guide was written, including important language concepts as well as a reference and examples of all resource types used during the experiment. The content of this guide was presented to all participants before the experiment was started. In addition, it was allowed to use this guide as a reference during the experiment. Therefore, each subject had this guide in printed format available.
- Same hardware for all experiments: The whole experiment was conducted on equally equipped machines with the same screen size and resolution. We used machines with an Intel Core i5-6600 CPU at 3,30GHz with 16GB RAM connected to monitor with a resolution of 1920 x 1080 and a diagonal viewing size of 60,45 centimeters.
- Automatic time measurements and information recording: The experiment environment was designed to record as much information as possible. For each programming task, the environment recorded the task duration time, as well as additional information such as amount of script executions, source code changes or test results.
- Upper time limit for each programming task: The experiment environment tracked the time for each experiment. Once the upper time limit of 4500 seconds was reached, the experiment supervisor was notified about this fact. In addition, the current execution time displayed by the experiment environment.

To realize these goals, the experiment environment consists of three main parts: the experiment controller, a text editor and a task execution environments. An overview of this experiment setup is shown in Figure 7.1.

The experiment participant is working with two windows: a text editor and a terminal window. The text editor is used to write and modify the task's source code and to answer the questionnaire. The terminal window is used to interact with the experiment controller and with the Puppet agent.

For each programming task a new Docker [Doc17] container is started. This container forms the task execution environment and is used to run the Puppet agent and the corresponding task test cases. This way we can ensure a clean system environment for each programming task. The Atom text editor is also running within a Docker container. In contrast to the task containers, this editor container is the same for the whole experiment. The manipulated source files are shared through a Docker volume with the task containers. This way, the editor container influences the behaviour of the Puppet agent, executed with a task container, but the source files are not lost once a task container is terminated.

The whole experiment is guided by an *experiment controller*, which was written especially for this software experiment. The experiment controller starts and stops the desired

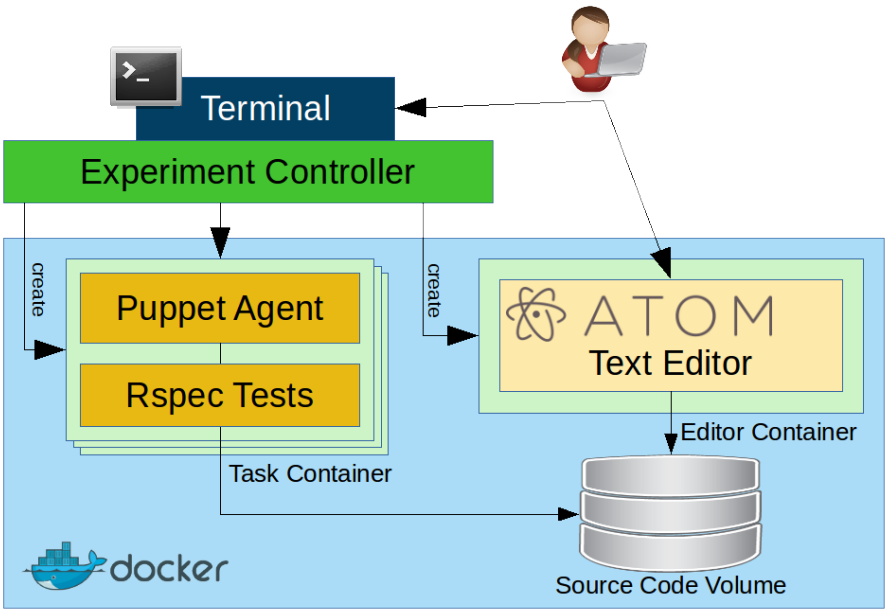

Figure 7.1: Experiment environment overview

Docker containers, opens the right source files in the text editor and collects information about the running experiment.

Once the experiment controller was started in the terminal window, users are asked to enter their name and the experiment group name they were assigned to. As soon as this information was provided, the text editor container is started. First the participant has to complete the preliminary questionnaire, which is appearing in the text editor. After finishing the questionnaire, the participant follows the instructions of the experiment controller to start with the first programming task. Depending on the assigned experiment group, the experiment controller will start the right variant of the first task, by starting a new task container and adding the right source folder to the editor window.

The experiment controller hands over the terminal window control (standard in and out) to the task container shell, in order to enable the user to run the Puppet agent and the task test cases. Each task container is equipped with two scripts: "`run_puppet`" and "`run_test`". The first one is used to execute the Puppet agent with the currently edited source files. The "`run_test`" script runs the Puppet agent together with test cases for the current task. Once all defined test cases have passed successfully, the user terminates the running task container by typing "`exit`". Now the terminal window control is handed back to the experiment controller, which starts the next task container.

Each task container writes a detailed log file, containing information about start and stop time, script executions and used shell commands. The measured duration times are derived by the container start and stop times.

Within the text editor container, a periodic Git commit job is running every 30 seconds. This periodic commit job records all changes of files within the source folder. This enables us to reconstruct the participants way to its solution. Additionally, all task container log files are tracked by this periodic commit job, too.

7.4 Programming Tasks

As already mentioned in Section 7.2, the whole experiment consists of four programming tasks in two or three variants. This section describes the programming tasks in more detail.

7.4.1 Task 1: JSON Configuration for Application `calculator`

The goal of this task was to finish a Puppet module that configures the fictional application "`calculator`". This application expects a configuration file in JSON format, with four settings in total. The Puppet module for this application was already predefined, whereas, only the configuration file management was missing, which had to be added by the participants. The main module class "`init.pp`" already defined four parameters, one for each configuration setting. This task had to be solved in two different variants:

Method ERB: Resource Type "`file`" and ERB Template

The participants were asked to solve this task by using the resource type "`file`" only. The content of the configuration file should be defined by an ERB template. The task description contained an example of JSON-style configuration file.

Method KDB: Resource Types "`kdbmount`" and "`kdbkey`"

Now the same task had to be solved using the resource types "`kdbmount`" and "`kdbkey`" only. The "`config.pp`" already suggested a Libelektra mount point for the configuration file.

7.4.2 Task 2.1: Hosts File Manipulation

The goal of this task was, to write Puppet code to manipulate a Hosts file. Participants were asked to add two new entries and update two existing entries. Each entry consists of an IP-address, a hostname and one alias name. Due to technical restrictions of a docker container, participants had to manipulate the custom hosts file "`/etc/hosts_bs`" instead of the standard system hosts file "`/etc/hosts`". This task had to be solved in three different variants:

Method HOST: Resource Type "`host`"

This variant had to be solved with the Puppet built-in resource type "`host`", which is specially designed to manipulate Hosts files.

Method AUG: Resource Type "augeas"

The same task had to be solved with the resource type "augeas" only. The task description contained an example of how the Augeas Hosts lens transforms a Hosts entry to its internal representation.

Method KDB: Resource Types "kdbmount" and "kdbkey"

This variant for Task 2.1 had to be solved with the resource types "kdbmount" and "kdbkey" only. As for method AUG, the task description contained an example of how the Libelektra hosts plugin transforms a Hosts entry to Libelektras' internal representation.

7.4.3 Task 2.2: Samba Configuration File Manipulation

The aim of this task was to manipulate the existing Samba configuration file "/etc/samba/smb.conf", an INI-style configuration file with multiple sections. The participants were asked to modify three settings in the main section "global" and to add two new share sections, each with three settings. The Puppet module code was already predefined and the participants had to extend the empty file "config.pp" with Puppet code to manipulate the configuration file in question. Again, this task had to be solved in three variants.

Method INI: Resource Type "ini_setting"

The participants were asked to solve this task with the resource type "ini_setting" only.

Method AUG: Resource Type "augeas"

The same task had to be solved with the resource type "augeas" and its built-in lens for the Samba configuration file. A sample representation of Samba lens was added to the task description.

Method KDB: Resource Types "kdbmount" and "kdbkey"

This method required to solve this task using the resource types "kdbmount" and "kdbkey". The task description did not say anything about the Libelektra storage plugin that can be used or how the internal representation may look like. It was up to the participant to find this information.

7.4.4 Task 3: Maintenance of Puppet Module "rubyhttp"

The goal of this task was to extend the existing Puppet module "rubyhttp" by two new configuration settings, "cache" and "memcached_connection", for a fictional Web-server, called "rubyhttp". This fictional program uses a JSON-style configuration file. The existing Puppet module consists of seven classes and source files, whereas the

configuration part is located in a single file "`config.pp`". Participants had to extend the main module class "`rubyhttp`" by two new parameters and add corresponding Puppet code to the configuration class "`rubyhttp::config`", to bring the value of the two parameter into the desired "`rubyhttp`" configuration file. In addition, the configuration setting "`memcached_connection`" should be only added to the configuration file, if the value of parameter "`cache`" is set to "`memcached`". This task had to be solved in two different variants.

Method ERB: Resource Type "`file`" and ERB Template

The Puppet module for this variant used the resource type "`file`" with an ERB template to write the desired configuration file. Therefore, participants had to extend the ERB template and add the requested parameter to the Puppet module. The task description contained hints, which source files may have to be extended.

Method KDB: Resource Types "`kdbmount`" and "`kdbkey`"

The Puppet module for this variant differed from the module from method ERB only by the configuration manipulation part, i.e. the source file "`config.pp`". Here, the configuration file was written by the resource types "`kdbmount`" and "`kdbkey`" only.

7.5 Subjects and Experiment Execution

Our experiment was conducted with 14 subjects, all of them being computer science students. One subject signed up as a result of advertising this user study in different university courses. The other 13 students are all tutors for the course "Programmkonstruktion".

As already mentioned earlier, before the first programming task was started, all subjects had to complete a questionnaire, capturing general information about the subject. The aim of this questionnaire was, to get an overall impression about the programming and tool experiences of each participant.

The first question captured system administration experience: three subjects had no experience, ten with basic experience and one subject had good system administration experience. None of the was a professional system administrator.

Four subjects state, that they already knew Puppet and did minor experiments with it. The majority did not know Puppet before this user study. One subject had also experience with another CM-tool (Slack).

The majority of our participants (10 subjects) stated, that they have a good software development experience, 3 subjects were professional software developers and one subject just had basic programming experience.

However, 8 subjects did not ever use Ruby, 4 of them tried is once and 2 use Ruby on a regular basis. The ERB template system was never used before by 13 subjects. However, 4 subjects used other template systems before.

Libelektra was not known by 7 subjects, 3 subjects already experimented with it and one is using it on a regular bases. The experiment included 3 participants who are part of the Libelektra developer community.

Augeas was unknown by 12 subjects and only two subjects already experimented with this tool. However, XPath, a central concept of Augeas, has been already used by 9 subjects, whereas 5 subjects did not know XPath. In contrast to this, all subject state, that they already used regular expressions once or on a regular basis.

All subjects already modified a configuration file by hand. 11 did that for JSON, 9 for INI-style files, 7 for Hosts and 4 for YAML.

All participants reported, that they are between 19 and 32 years old.

Before the actual experiment was executed, we performed an initial test run. The aim of this pre-study was to test the experiment setup and to check if the defined programming tasks are solvable within reasonable time. This pre-study was conducted with 4 subjects. Based on the results of this first pre-study we refined our experiment and conducted a second test run, again with 4 subjects. The second test run already worked as expected, therefore we did not have to alter the experiment after this second test session. Since the second pre-study was performed in the same way as the main study sessions, we decided to use the results of this second pre-study session.

The main experiment was performed in two sessions. Two subjects participated in the first session, 8 subjects in the second session. Together with the 4 subjects of the second pre-study session we have 14 experiment participants in total.

Based on the results from the first pre-study session, we decided to limit the maximum execution time for one task method on 4500 seconds. Once a subject hit this limit we gave some hints and helped to solve this task. Therefore, all measured duration times are limited on 4500 seconds.

Before the experiment was started, all participants were introduced into the Puppet language based on the introduction guide. As already stated earlier, participants were allowed to use this guide as a reference during the whole experiment. In addition to this guide, a task description paper was handed out to all subjects.

After the subjects were introduced to Puppet, the experimenter assigned each subject to a group and started the experiment environment for this subject. The group assignment was not done randomly, instead this was chosen by the experimenter. This manual assignment should help to keep the number of subjects per group balanced [Pre00]. This was done in a round-robin fashion. The experimenter picked the next group with the lowest number of participants and assigned this group to the next subject.

Since the duration time for each programming task was measured automatically, all subjects were advised not to take breaks during a programming task. For rare conditions, that happened nevertheless, we introduced a "`pause`" command, which recorded the break duration. This timeout was considered by our experiment analysis scripts.

7.6 Results and Analysis

This section presents the measured results and some analysis. Section 7.6.1 shows all measured results and a basic descriptive statistic. Section 7.6.2 presents our analysis steps to answer the question, if there are significant differences between the used methods. Section 7.6.3 shows a subjective usability rating based on the subjective impression of the subjects.

An anonymized form of the measured data together will all used analysis scripts for the statistics tool R [Fou17b] can be online at `http://puppet-userstudy-results.libelektra.org`.

7.6.1 Descriptive Statistics

Table 7.1 shows the collected task duration times in seconds. Each row contains all measured times for one subject. For example subject 1 required 604 seconds to Task 2.1 with method HOST and 505 seconds with method KDB.

subject	Task 1		Task 2.1			Task 2.2			Task 3	
	ERB	KDB	HOST	AUG	KDB	INI	AUG	KDB	ERB	KDB
1	1283	557	604	3962	505	710	3276	406	1407	840
2	3397	948	1456	4500	4500	1557	4500	1750	2563	1744
3	2269	4382	728	4500	2838	953	4215	1703	1364	2463
4	3507	1561	828	3173	1881	1361	3786	1522	380	2296
5	513	609	191	2114	662	476	1547	384	527	1422
6	1202	494	298	2458	607	733	1555	410	1057	479
7	994	844	275	3870	601	600	1483	385	307	527
8	1783	350	366	4500	537	394	4500	331	3654	2041
9	658	937	389	2610	800	505	4500	649	682	1772
10	1789	1130	347	4500	1428	348	2312	1397	443	1047
11	2371	1188	1334	1978	691	685	1915	1151	996	841
12	1599	1247	643	4500	2750	838	4500	1429	599	2036
13	717	385	287	4281	1253	440	1157	309	508	289
14	1235	426	245	4500	2226	385	895	897	361	726

Duration values in seconds

Table 7.1: Experiment results

Based on this data we performed some descriptive statistics, shown in Table 7.2. The first row contains the sum of all subjects for each task and method. So all subjects together required 23317 seconds for Task 1 with method ERB. The maximum, minimum, arithmetic mean, median and standard deviation for each task and method is shown in the next five rows. Based on this, we also calculated the difference between the total task duration time between two methods. For Task 1 and Task 3 this is done by calculating the difference between method ERB and method KDB. For Task 2.1 and Task 2.2 we did the same for each method combination. The same calculation was done for the arithmetic mean, which is shown in row number eight. The last row contains the number of occurrences, where the difference between two methods of one subject is less than zero. For example, for Task 1 we can see, that 3 subjects required less time to solve the task with method ERB than with method KDB.

function	Task 1		Task 2.1			Task 2.2			Task 3	
	ERB	KDB	HOST	AUG	KDB	INI	AUG	KDB	ERB	KDB
Sum	23317	15058	7991	51446	21279	9985	40141	12723	14848	18523
max	3507	4382	1456	4500	4500	1557	4500	1750	3654	2463
min	513	350	191	1978	505	348	895	309	307	289
arith. mean	1665.5	1075.57	570.79	3674.71	1519.93	713.21	2867.21	908.79	1060.57	1323.07
median	1441	890.5	377.5	4121.5	1026.5	642.5	2794	773	640.5	1234.5
std. dev.	941.56	1021.02	400.11	990.54	1191.49	365.42	1438.54	561.22	962.68	732.34
Δ Sum	8259		HOST - AUG: -43455 HOST - KDB: -13288 AUG - KDB: 30167			INI - AUG: -30156 INI - KDB: -2738 AUG - KDB: 27418			-3675	
mean diff.	590		HOST - AUG: -3104 HOST - KDB: -949 AUG - KDB: 2155			INI - AUG: -2154 INI - KDB: -196 AUG - KDB: 1958			-262	
#(diff. < 0)	3		HOST - AUG: 14 HOST - KDB: 12 AUG - KDB: 0			INI - AUG: 14 INI - KDB: 8 AUG - KDB: 1			8	

Table 7.2: Descriptive Statistics

A first glance on these calculations already gives us a good overview of our experiment results. The majority of subjects (79%) solved Task 1 faster, when they were using the proposed solution. A similar situation can be seen for Task 2.1 and Task 2.2 between method AUG - KDB and HOST/INI - AUG. Here the situation is even more clear: all subject (100%) solved Task 2.1 and 2.2, when they were using method HOST or INI then with method AUG. All subjects required less time for Task 2.1, when using method KDB compared to AUG. Only one subject was faster with method AUG compared to KDB for solving task 2.2. However, if we look in Table 7.1, we can see, that subject 14 required only 2 seconds more to solve Task 2.2 with method KDB compared to method AUG.

A different situation can be found in Task 3. 8 subjects (57%) solved Task 3 faster, when using method ERB. Also, the differences between the total duration times and arithmetic mean for method ERB and KDB show the trend, that subjects usually required less time to solve Task 3, when they were using method ERB.

To visualize these first tendencies of duration times between each method, we have created

box plots for each task. Figure 7.2 shows the box plot for Task 1.

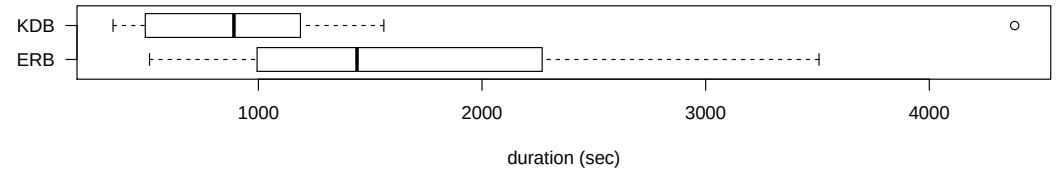

Figure 7.2: Box plot for Task 1

The task duration time is shown on the X-axis, whereas on the Y-axis we see the different treatment levels (task methods). The box represents the 25%, the 50% (also know as median) and 75%-quantile. This means, half of our duration times is within the box and the other half is outside the box. The marks next to the box, called whiskers, represent the 10% and 90%-quantile [Woh+00]. Small circles outside the whiskers show outliers.

The box plot for Task 1 (Figure 7.2) strengthens the assumption, that subjects usually were faster, when they used method KDB.

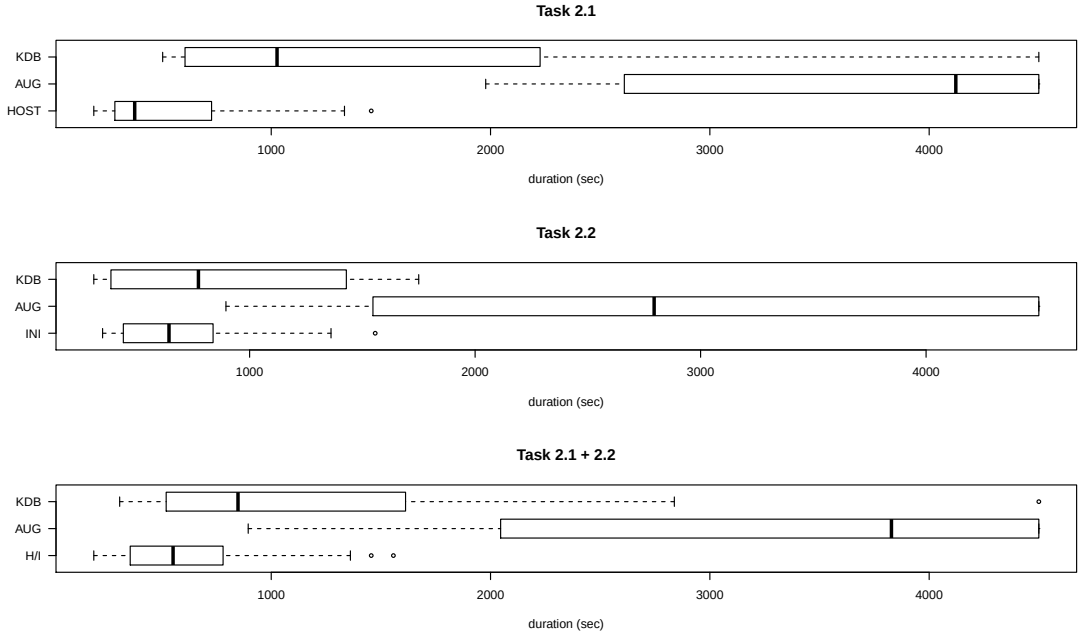

Figure 7.3: Box plot for Task 2

The box plot for Task 2.1 and Task 2.2 (Figure 7.3) shows even greater differences between the used methods. Very interesting are the boxes for method AUG. The boxes

for both tasks, and the overall Task 2 box do not even overlap with the boxes of the other two methods. It even seems, that the maximum time limit of 4500 seconds has cut-off the AUG boxes, as the 75% and the 90%-quantile are both 4500 seconds. This is a result of the fact, that 6 subjects (43%) hit the time limit for Task 2.1 with method AUG and 4 subjects (29%) for Task 2.2. 3 subjects hit the time limit for both tasks, 2.1 and 2.2.

This great difference is not seen between method KDB and HOST/INI, whereas the for Task 2.1 we can assume, that our subjects usually were faster with method HOST. The situation for Task 2.2 seems to be slightly different. The majority was usually faster when using method INI, however some subjects required less time with method KDB, since the 10% and 25%-quantile of method KDB are both smaller than those of method INI.

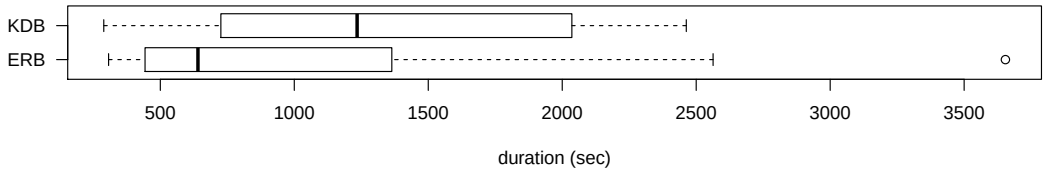

Figure 7.4: Box plot for Task 3

The box plot of Task 3 (Figure 7.4) shows a different picture, compared to Task 1. We can assume, the majority of our subjects required less time to solve Task 3, when the were using method ERB. However, the boxes of both methods have a quite great overlapping area. Additionally, the median for method ERB is smaller than the 25%-quantile for method KDB, which is very interesting.

7.6.2 Testing for Significant Differences

A classic tests method for intra-subject experiments is the paired T-test, which tests if the means of two treatment levels of one subject are substantially the same [Woh+00]. However, this statistical test has the precondition, that the differences between the treatment levels for each subject should be normally distributed [McD15a]. Testing a distribution for normality can be performed by the Shapiro-Wilk test [SW65; GZ12]. An alternative to the paired T-test is the Wilcoxon signed rank test, which also can be used, if the differences of our samples are not normally distributed [McD15a; Woh+00; McD15b]. However, Prechelt, Lutz mentions in the book [Pre00, p. 189] that testing for a normal distribution is often a dangerous thing. A statistical test is most meaningful, if the null hypothesis is rejected, i.e. the tested sample is not normally distributed. Furthermore, such tests are rather insensitive, if the sample is small [Pre00].

As a result of this, we have decided to perform the Shapiro-Wilk test to check for a normally distributed sample. Regardless of its result, we performed both hypothesis

tests, the paired T-test and the Wilcoxon test, in order to see if the results of both tests largely differ.

Hypothesis Theorem 1. *Null hypothesis H_0 of the paired T-test: the mean difference of the paired samples is zero. Alternative hypothesis: the difference is not zero. [McD15a; Woh+00]*

Hypothesis Theorem 2. *Null hypothesis H_0 of the Wilcoxon signed rank test: the median difference of the paired samples is zero. Alternative hypothesis: the difference is not zero. [McD15b]*

Hypothesis Theorem 3. *Null hypothesis H_0 of the Shapior-Wilk normal distribution test: the tested sample is normal distributed. Alternative hypothesis: it is not. [GZ12]*

The null hypothesis for the paired T-test and the Wilcoxon signed rank test can be expressed as: the required effort to solve the task with method X is the same as for method Y.

Based on the null hypothesis or alternative hypothesis from our statistical tests, we can make decisions on a certain significant level α. As already stated in Section 7.2 we are using a significance level of $\alpha = 0.05$ for this experiment analysis. This means, the probability of making the wrong decision is $\alpha = 0.05$.

Task	Methods	Shapiro-Wilk	paired T-test	Wilcoxon test	H_0 rejected
Task 1	ERB - KDB	0.4	0.06	0.02	**yes**[†]
Task 2.1	HOST - AUG	0.08	8×10^{-8}	6×10^{-6}	**yes**
Task 2.1	HOST - KDB	0.5	0.004	0.005	**yes**
Task 2.1	AUG - KDB	0.6	5×10^{-6}	4×10^{-4}	**yes**
Task 2.2	INI - AUG	0.1	3×10^{-5}	3×10^{-5}	**yes**
Task 2.2	INI - KDB	0.4	0.1	0.7	no
Task 2.2	AUG - KDB	0.3	7×10^{-5}	3×10^{-4}	**yes**
Task 3	ERB - KDB	1	0.3	0.2	no

[†] Result of Wilcoxon test used, since test for normality was rejected. However, the result of the paired T-test is very close to our chosen significance level

Table 7.3: Hypothesis test results

Table 7.3 shows the calculated p-values for each test and each method grouping. The null hypothesis is rejected if the calculated p-value is less than the significance level $\alpha = 0.05$. If this is the case, the difference of the compared methods is statistically significant, i.e. the probability of making the wrong decision is less than 5% (based on our chosen value for α). If we cannot reject the null hypothesis, the probability of making wrong decision is grater than 5%. The decisions to reject or accept H_0 is based on the Wilcoxon signed-rank test, as we had to reject the null hypothesis for the normality test in all cases.

As a result from these tests, we are able to conclude, that there are, with exception of two cases, always significant differences between the Puppet file manipulation methods used during this user study.

From the box plot for Task 1 (Figure 7.2) we can assume, that our subjects usually were faster when they used method KDB to solve this task. The hypothesis test underpins this assumption, as we also can reject the null hypothesis based on the Wilcoxon signed-rank test.

The box plots for Task 2.1 and Task 2.2 show, that there are huge differences between the times spent to solve Task 2.1 or 2.2 when subjects used method AUG or the specialized methods HOST or INI. The statistical hypothesis tests clearly underpinned this surmise. A similar situation, however not that dramatically, is seen between the methods AUG and KDB. Also, for this case we can conclude, that the difference between these two methods is significant. Therefore, our subjects had to spend much more time to solve Task 2.1 and 2.2 when they were using method AUG in contrast to the other two methods.

The remaining method combination for Task 2.1 and 2.2 is two-folded. Based on the Wilcoxon test we can conclude, that for Task 2.1 the difference between method HOST and KDB is significant. So our subjects required more time to solve Task 2.1 when they were using method KDB. The same surmise is not true for Task 2.2. Here we can see, that our statistical hypothesis test does not reject the null hypothesis. This means, that there is no significant difference between these two methods. However, directly concluding that the opposite is true and both methods are equal is wrong. A statistical hypothesis test gives us a clear statement, if the null hypothesis is rejected [Pre00]. From the box plot in Figure 7.3 we already can see this behaviour. Usually our subjects required more time to solve Task 2.2 with method KDB in contrast to method INI. However, there are cases where subjects were faster with method KDB. Also, the fastest time to solve Task 2.2 was done with method KDB. It seems that there are more factors involved in this situation, which we do not know.

The null hypothesis for the Wilcoxon test for Task 3 cannot be rejected. Therefore, we cannot conclude, that the difference between method KDB and ERB for this task is statistically significant. Based on our observations during the experiment supervision however, we guess that the order in which the methods have been used has some influence on the task duration times. Therefore, we have calculated a special box plot for Task 3 and grouped the duration times by the subjects group assignment and the used method. This box plot is shown in Figure 7.5.

This box blot show for each method-group assignment combination from Task 3 a separate box plot. Group A solved Task 3 with method ERB first and then with method KDB. Group B started with method KDB and finished with method ERB. As we have to split up the whole data into two groups, each box plot in Figure 7.5 is based on 7 samples only.

Despite this low number of samples, we can see a special situation in this graph. The situation for group B seems to be quite clear. Subjects of Group B, required much more

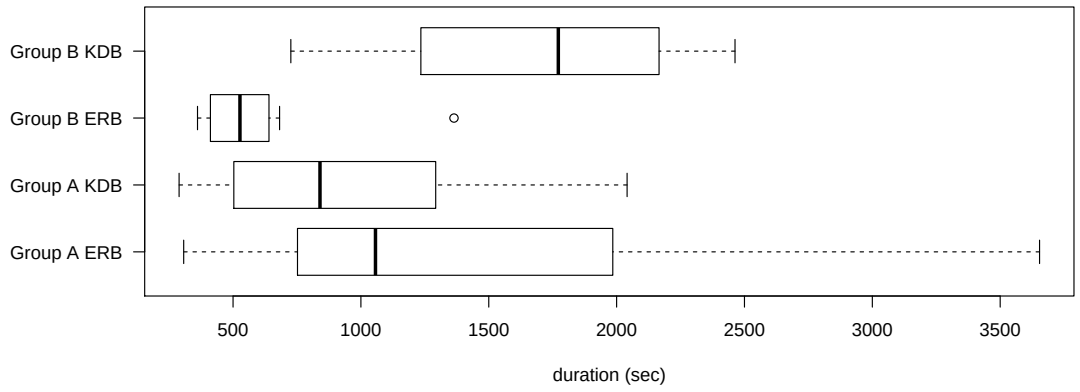

Figure 7.5: Box plot for Task 3 grouped by group assignment and method

time to solve Task 3 with the first method (KDB). The second round for participants of group B was done much faster. A Wilcoxon signed-rank test for the two method of group B delivered a p-value of 0.004. Thus, the difference between the two methods in group B is significant. The opposite situation can be seen for group A. There seems to be an evidence, that subjects of group A usually required more time to solve Task 3 the first time (here ERB). The second time (method KDB) they usually required less time. A Wilcoxon test for this group revealed a p-value of 0.3 and therefore, the difference is not significant.

As a result from this, we can conclude that both factors, the used method and the order in which the methods were used, might have an influence on the time required to solve the task. Having said that, none of the two factors showed a significant difference.

7.6.3 Subjective Impressions

Once a subject had solved a task using a specific method, they had to rate the usability of each method for one particular task.

Figure 7.6 shows the overall subjective usability ratings for each used method and is based on the data recorded by the questionnaires filled up by the subjects after each programming task.

This box plot shows, that the subjects had a good impression of the usability of the proposed solution (method KDB). The majority of subjects usually rated this method between 1 and 2. Only one method was rated with a higher usability: method HOST.

Subjects have the impression, that method INI has a comparable usability as method KDB. Some subjects (43%) rated KDB even better than INI. However, it is important to note, that the data for the box plot of INI is based on 14 samples, whereas the data of KDB is based on 56 samples, because method KDB was used within every task.

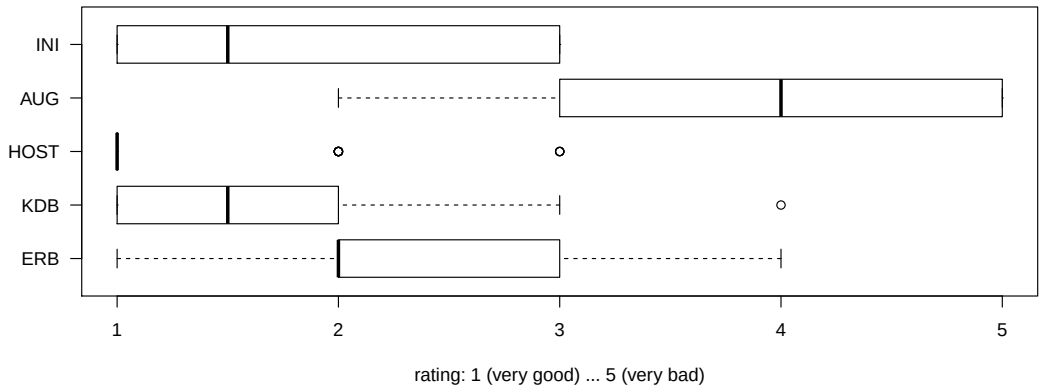

Figure 7.6: Box plot for usability ratings per method

Subjects have rated the usability of method AUG to be very bad. This observation conforms to the measured duration times of the last two sections.

The usability of method ERB was rated by the majority of subjects between 2 and 3 and therefore was rated better than method AUG but worse than method KDB or INI.

7.7 Discussion

The results of this user study showed, that there are significant productivity differences between the compared Puppet configuration file manipulation methods. Also, subjective usability impressions by the experiment subjects outline these differences.

The experiment results for Task 1 indicated, that the proposed solution has a significant impact on development productivity and therefore, helps to reduce the required development time. At least this result is valid for more complex configuration file formats, such as the used JSON format. Subjects often reported, that they have struggled with syntactical issues while they were implementing the ERB template for Task 1 using method ERB.

Task 2.1 and 2.2 was designed to compare four different partial file manipulation methods. Two very specific methods, HOST and INI and two general methods, KDB and AUG. We expected, that the specific methods would require a low time to complete the tasks and have a very good usability rating, as both resource type APIs are adopted to their underlying configuration file format. The resource type "`host`" of method HOST has three main elements that reflect the elements of a Hosts entry: "`name`", "`ip`" and "`host_aliases`". Therefore, it is very clear to a user, how to write the resource declaration to manage one particular Hosts entry with given values. The more general methods have a disadvantage here, as users have to figure out which keys have to be manipulated. The situation for method INI is similar. The resource type "`ini_setting`"

has four main attributes: "`path`" to specify the target configuration file, "`section`", "`setting`" and "`value`". This makes it also very clear to a Puppet user how to write a resource declaration to manipulate a specific setting within a defined INI configuration file.

Our expectations were confirmed by the results of the user study. Subjects usually required less time to implement Task 2.1 and 2.2 when they used a format specific method. The measured time differences between method HOST and the methods KDB and AUG are significant. Therefore, we can conclude that Puppet users will be most productive when they use the resource type "`host`" to manipulate Hosts entries.

The measured difference between method INI and KDB shows a similar trend. However, this difference is not significant. In fact, some subjects (43%) required less time using method KDB compared to method INI.

The user study also showed, that method AUG is difficult to use. On average subjects required more than 6 times longer to complete Task 2.1 with method AUG compared to method HOST and more the 2 times longer compared to method KDB. This is a result of the complexity of the resource type "`augeas`". Users have to be aware of its XPath like notion of specifying key names, the specific transformations to its internal representation of the configuration values and specific behaviour of the used lens. Especially new Puppet users require a lot of time to use this method effectively.

Task 3 was designed to show productivity differences when maintaining existing Puppet code for configuration file manipulation. Although, there is a tendency, that method ERB seems to be more efficient, our user study did not show a clear winner, as the measured time differences are not significant. This might be a consequence of a badly designed task, since the order in which both methods were carried out had also an influence on the development time. The task also required to add two new parameters to the main module class. However, this is very specific to the Puppet language itself and has no impact on the file manipulation. Further more, this had to be done for both methods in the same way. So, once this was done for the first method, subjects required less time to implement this part for the second method. Unfortunately, this problem was not detected during both pre-study sessions. As a consequence of this, we cannot derive a clear statement from the results of Task 3.

Based on the results of the conducted user study, we can formulate answers for the defined research questions.

Sub Research Question 2.1. *When used in scenarios, which require a full definition of a configuration file, can the proposed solution help to reduce development time when compared with existing methods?*

Sub research question 2.1: Yes, the proposed solution helps to reduce the time required to define a new configuration file from scratch.

Sub Research Question 2.2. *When used in scenarios, which require partial modification of configuration files, can the proposed solution help to reduce development time when compared with existing methods?*

Sub research question 2.2: As a general purpose configuration manipulation strategy, the proposed solution can help to reduce the time required to partially manipulate configuration files when compared with existing general purpose configuration manipulation strategies such as "`augeas`". In some cases the proposed solution is on par with specialized manipulation strategies, such as "`ini_setting`" and leads to similar development times.

Sub Research Question 2.3. *Can key-value-based configuration manipulation methods, such as the proposed solution, help to reduce the time spent for maintenance tasks?*

Sub research question 2.3: It is unclear if the proposed solution leads to a reduction in time spent for maintenance tasks.

Research Question 2. *Which approach of configuration file manipulation provides the best results if compared in terms of usability and maintainability? Does the proposed solution have an advantage over existing methods?*

Research question 2: As a general purpose configuration manipulation strategy, the proposed solution provides the best usability results when compared with other general purpose manipulation strategies. Specialized manipulation methods, such as "`host`" or "`ini_setting`" provide in some cases better results. In terms of maintainability, the proposed solution does not have a significant impact on developer productivity. Overall, the proposed solution has an advantage over existing Puppet file manipulation methods.

CHAPTER 8

Conclusion

8.1 Future Work

The conducted case study has shown that our solution is able to compete with challenges found in real-world scenarios, however, there is still place for improvements.

A major drawback of the current implementation is that each "`kdbkey`" resource declaration leads to its own Libelektra read/write cycle of the managed configuration file. Thus, the configuration file is parsed and written to disk for each manipulated configuration setting. The conducted feature comparison has shown that this approach leads to a low runtime performance. We decided on this procedure, since it integrates very well in the way Puppet works and it provides the most precise error messages. The main problem in this context is Libelektras' key validation. The Puppet agent performs resource updates for each resource declaration separately and instantly reports the result of this update to the user. Thus, if we choose to trigger a Libelektra write cycle for each configuration file only once, the "`kdbkey`" implementation has to delay this write cycle until all "`kdbkey`" resources for this configuration file have been updated. However, if one of these "`kdbkey`" resource updates leads to a validation error, this problem is detected at a very late stage, once the last "`kdbkey`" resource for the configuration file is updated. The main problem is that the last "`kdbkey`" resource might not be the faulty one, in fact any of the previously updated keys could be responsible for this error. However, from Puppets point of view, the last updated "`kdbkey`" resource is responsible for the error, so the presented error message will be misleading. This problem can be avoided by performing a validation cycle for each updated "`kdbkey`" resource declaration without actually writing the underlying configuration file. However, this requires an extension of the Libelektra API, since this is currently not possible.

The simple strategy, using one read/write cycle for each "`kdbkey`" resource declaration, is not feasible when using the Libelektra "`augeas`" storage plugin. The case study has

shown (Section 6.3.2 and 6.3.3) that we have to write multiple Libelektra keys at once using the "`augeas`" storage plugin, in order to define new Augeas structures. However, this is currently not possible with the *puppet-libelektra* module. A possible solution to this problem is to use only one Libelektra ready/write cycle for each mount point. This way, a Puppet user can define multiple "`kdbkey`" resource declarations to define a valid Augeas structure, while our proposed solution combines them all to one Libelektra write cycle.

Another improvement aspect of our solution is to get rid of the required configuration file mounting. As we described in Chapter 3, we have chosen to implement a separate "`kdbmount`" resource type, to concentrate the mounting information in exactly one location. If it is possible to derive this mounting information by the proposed solution automatically, the user does not have to care about the mounting process. The *spec-mount* feature of Libelektra can help here as demonstrated in the case study (Section 6.3.8). However, this requires to add mount specifications for all kind of configuration files on the managed system to the "`spec`" namespace, in order to provide Libelektra the information which files have to be mounted with which plugins. Ideally, the managed software already comes with this information, however this is currently not the case.

Puppet versions below 4.x are using the Ruby runtime installed on the managed system. This makes it easy to install the required dependencies for our proposed solution, Libelektra and its Ruby bindings. Starting with version 4.x, Puppet developers decided to ship the software with its own Ruby ecosystem. Using our solution with these Puppet versions requires integrating Libelektras' Ruby bindings into the Ruby runtime shipped with Puppet. How this can be done in an efficient and reliable way needs some further investigation.

8.2 Conclusion

The aim of this thesis was to provide a general purpose configuration manipulation method for the CM-tool Puppet, which operates on a key-value basis. This approach has the main advantage that configuration settings are abstracted to key-value pairs and users do not have to take care of the underlying configuration file syntax. In addition, an integrated configuration validation system allows us to define configuration specifications which are automatically ensured once configuration values are updated. This approach helps to detect configuration problems at an early stage.

Our solution integrates key-value pairs as a first-class citizen into Puppets configuration language. This approach provides the same abstraction level for configuration settings as for software packages and user accounts. Users do not have to care about low-level details of configuration files, instead they are able to concentrate on the actual work: configuring software systems.

To show that our solution can be used to solve real-world configuration management problems and works better than other existing general purpose configuration manipulation

strategies, we have evaluated our proposed solution in three different steps.

To show that the *puppet-libelektra* methods are able to solve configuration tasks for a real-world scenario, we have conducted a case study. Therefore, we have used our solution to manage a continuous integration system, consisting of four different computer systems. This case study has show, that our solution can be used for all forms of configuration files. However, the full power of the implemented "`kdbkey`" and "`kdbmount`" resources types can be used if Libelektra offers a suitable storage plugin for the managed configuration file format. Otherwise, we have to fallback to a full-content-at-once approach, available through Libelektras' "`line`" and "`file`" plugins. Alternatively, users have the option to write their own Libelektra storage plugins, as shown for the "`authorized_keys`" and shell-variables configuration files (Section 6.3.4 and 6.3.7). The possibility to manipulate configuration files using Libelektras' "`augeas`" storage plugin requires some enhancements of our proposed solution.

Our solution is able to increase the usability of configuration file manipulation by CM-tools. This was illustrated by conducting a user study, comparing the usability and maintainability in a quantitative and qualitative way. As part of this user study, we asked 14 participants to solve four Puppet programming tasks aimed to manipulate different configuration files. The time required to solve each task with our solution and with other existing Puppet configuration manipulation methods was measured and compared. This user study showed, that our solution is able to increase the development productivity significantly compared to other general purpose configuration manipulation strategies, regardless if the underlying configuration file is managed partially or created from scratch. Our solution is even on par with very specific configuration manipulation methods, especially designed for one configuration file format. Nonetheless, an increased maintenance productivity of existing Puppet code could not be shown.

To captured the subjective impressions of our participants, we asked them to rate the usability of the used Puppet file manipulation methods after each programming task. The results of the reported usability ratings show a similar situation as the quantitative analysis already did. Our participants rated the usability of our solution better than existing general purpose configuration manipulation strategies.

Additional to applicability and usability we compared the proposed solution with existing methods in terms of runtime performance. The runtime performance of our solution is better than some existing methods, such as the Augeas approach but worse than other solutions like the "`ini_setting`". This shows that there is place for improvement.

Despite the great amount of existing methods for manipulating configuration files, it is possible to do it in a better and more productive way. Our thesis showed, that specifying configuration settings in a more abstract way helps system administrators to manage their infrastructure.

List of Figures

List of Tables

List of Listings

Bibliography

[16] *Linux Programmer's Manual - HOSTS*. 2016. URL: http://www.man7.org/linux/man-pages/man5/hosts.5.html (visited on 04/28/2017).

[BC06] Mark Burgess and Alva Couch. "Modeling Next Generation Configuration Management Tools". In: *Proceedings of LISA '06: 20th Large Installation System Administration Conference* (Dec. 2006), pp. 131–147. URL: http://usenix.org/event/lisa06/tech/full_papers/burgess/burgess.pdf.

[Bra89] Robert Braden. *Requirements for Internet Hosts – Application and Support*. RFC 1123. RFC Editor, Oct. 1989. URL: http://www.rfc-editor.org/rfc/rfc1123.txt (visited on 04/28/2017).

[Bro09] Gregory Brown. *Ruby Best Practices*. 1st ed. O'Reilly, June 2009. ISBN: 978-0-596-52300-8.

[Bur03] Mark Burgess. "On the theory of system administration". In: *Science of Computer Programming* 49.1–3 (2003), pp. 1–46. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2003.08.001.

[Bur95] Mark Burgess. "CFEngine: a site configuration engine". In: *The USENIX Association, Computing Systems*. Vol. 8. 3. 1995.

[CFE16a] CFEngine AS. *CFEngine - Configuration management tool*. 2016. URL: https://cfengine.com/ (visited on 03/06/2016).

[CFE16b] CFEngine AS. *CFEngine Reference - Standard Library - Files Bundles and Bodies*. 2016. URL: https://docs.cfengine.com/docs/master/reference-standard-library-files.html (visited on 05/05/2016).

[Che16a] Chef Software, Inc. *Augeas cookbook*. 2016. URL: https://github.com/nhuff/chef-augeas (visited on 05/05/2016).

[Che16b] Chef Software, Inc. *Chef - Configuration management tool*. 2016. URL: https://www.chef.io/chef/ (visited on 03/06/2016).

[DJV10] Thomas Delaet, Wouter Joosen, and Bart Van Brabant. "A Survey of System Configuration Tools." In: *Proceedings of the Large Installations Systems Administration (LISA) conference*. 2010. URL: https://www.usenix.org/event/lisa10/tech/full_papers/Delaet.pdf.

[Doc17] Docker Inc. *Docker - Build, Ship, and Run Any App, Anywhere*. 2017. URL: https://www.docker.com/ (visited on 04/28/2017).

[Dom17] Dominic Cleal and Raphaél Pinson. *augeasproviders*. 2017. URL: http://augeasproviders.com/ (visited on 04/11/2017).

[Dun15] E Dunham. *Configuration Management Comparison*. June 5, 2015. URL: http://edunham.net/2015/06/05/configuration_management_comparison.html (visited on 03/08/2016).

[Ele16] Elektra Development Community. *Elektra Documentation 0.8.19*. Nov. 22, 2016. URL: https://doc.libelektra.org/api/0.8.19/html/ (visited on 04/19/2017).

[Ele17] Elektra Development Community. *ElektraInitiative*. Apr. 8, 2017. URL: https://www.libelektra.org/.

[Fit15] Michael Fitzgerald. *Ruby Pocket Reference: Instant Help for Ruby Programmers*. 2nd ed. O'Reilly, Aug. 2015. ISBN: 978-1-491-92601-7.

[Fou17a] The Apache Software Foundation. *Configuration Files - Apache HTTP Server Version 2.4*. 2017. URL: https://httpd.apache.org/docs/2.4/configuring.html (visited on 06/16/2017).

[Fou17b] The R Foundation. *R: The R Project for Statistical Computation*. 2017. URL: https://www.r-project.org/ (visited on 06/17/2017).

[GZ12] Asghar Ghasemi and Saleh Zahediasl. "Normality Tests for Statistical Analysis: A Guide for Non-Statisticians." In: *International Journal of Endocrinology and Metabolism* 10.2 (2012), pp. 486–489. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3693611/ (visited on 06/05/2017).

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. English. Addison-Wesley, 2010, pp. 49–54, 277–324. ISBN: 978-0-321-60191-9.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000. ISBN: 0-201-61622-X.

[Int13] ECMA International. *The JSON Data Interchange Format*. 2013. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 04/28/2017).

[Kan12] Luke Kanies. "Puppet". In: *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Amy Brown and Greg Wilson. Vol. 2. Kristian Hermansen, May 2012. ISBN: 1105571815.

[Kit+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. "Preliminary guidelines for empirical research in software engineering". In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002), pp. 721–734. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1027796.

[Kru+13] Spencer Krum, William Van Hevelingen, Ben Kero, James Turnbull, and Jeffrey McCune. *Pro Puppet*. Second Edition. Apress, Dec. 2013. ISBN: 978-1-4302-6040-0.

[KUC08] L. Keller, P. Upadhyaya, and G. Candea. "ConfErr: A tool for assessing resilience to human configuration errors". In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. June 2008, pp. 157–166. DOI: 10.1109/DSN.2008.4630084.

[Lab16a] Puppet Labs. *Custom Types — Documentation — Puppet*. 2016. URL: https://docs.puppet.com/guides/custom_types.html (visited on 11/16/2016).

[Lab16b] Puppet Labs. *Provider Development - Documentation - Puppet*. 2016. URL: https://docs.puppet.com/puppet/guides/provider_development.html (visited on 11/16/2016).

[Lab16c] Puppet Labs. *Puppet 4.8 reference manual — Documentation — Puppet*. 2016. URL: https://docs.puppet.com/puppet/4.8/reference/index.html (visited on 11/16/2016).

[Lab16d] Puppet Labs. *Puppet Forge*. 2016. URL: https://forge.puppet.com/ (visited on 03/06/2016).

[Lab16e] Puppet Labs. *Puppet Labs: IT Automation Software for System Administrators*. 2016. URL: https://puppetlabs.com/ (visited on 03/06/2016).

[Lal16] Chris Lalancette. *libvirt Ruby bindings*. Sept. 22, 2016. URL: http://libvirt.org/git/?p=ruby-libvirt.git (visited on 04/19/2017).

[Lut08] David Lutterkort. "AUGEAS - a configuration API". In: *Proceedings of the Linux Symposium*. Vol. 2. July 2008, pp. 47–56. URL: http://www.landley.net/kdocs/ols/2008/ols2008v2-pages-47-56.pdf (visited on 03/04/2016).

[Mat14] Yukihiro Matsumoto. *Ruby Extension Documentation 2.1.10*. July 2, 2014. URL: https://github.com/ruby/ruby/blob/v2_1_10/README.EXT (visited on 04/19/2017).

[Mat17] Yukihiro Matsumoto. *Ruby Programming Language*. 2017. URL: https://www.ruby-lang.org/ (visited on 04/19/2017).

[McD15a] John McDonald. "Paired T-Test, Handbook of Biological Statistics". In: (July 2015). URL: http://www.biostathandbook.com/pairedttest.html (visited on 06/17/2017).

[McD15b] John McDonald. "Wilcoxon Signed-Rank Test, Handbook of Biological Statistics". In: (July 2015). URL: http://www.biostathandbook.com/wilcoxonsignedrank.html (visited on 06/17/2017).

[Mey+13] S. Meyer, P. Healy, T. Lynn, and J. Morrison. “Quality Assurance for Open Source Software Configuration Management”. In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*. Sept. 2013, pp. 454–461. DOI: 10.1109/SYNASC.2013.66.

[NIB11] Thomas Damgaard Nielsen, Christian Iversen, and Philippe Bonnet. “Private Cloud Configuration with MetaConfig”. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. July 2011, pp. 508–515. DOI: 10.1109/CLOUD.2011.63.

[OGP03] David Oppenheimer, Archana Ganapathi, and David A. Patterson. “Why Do Internet Services Fail, and What Can Be Done About It?” In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS’03. Seattle, WA: USENIX Association, 2003, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1251460.1251461.

[Pan12] Sudhir Pandey. “Investigating Community, Reliability and Usability of CFEngine, Chef and Puppet”. Master’s thesis. Network, System Administration Oslo, and Akershus University College, 2012. URL: http://urn.nb.no/URN:NBN:no-31901.

[Pre00] Prechelt, Lutz. *Kontrollierte Experimente in der Softwaretechnik: Potential und Methodik*. Springer, 2000. ISBN: 3-540-41257-3.

[Raa10] Markus Raab. “A modular approach to configuration storage”. Master’s thesis. Vienna University of Technology, 2010.

[Raa16a] Markus Raab. “Elektra: universal framework to access configuration parameters”. In: *The Journal of Open Source Software* 1.8 (Dec. 2016). DOI: 10.21105/joss.00044.

[Raa16b] Markus Raab. “Improving System Integration Using a Modular Configuration Specification Language”. In: *Companion Proceedings of the 15th International Conference on Modularity*. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 152–157. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892691.

[Red16] Red Hat, Inc. *Ansible is Simple IT Automation*. 2016. URL: https://www.ansible.com/ (visited on 03/06/2016).

[Sal16] SaltStack Inc. *SaltStack automation for CloudOps, ITOps and DevOps at scale*. 2016. URL: http://saltstack.com/community/ (visited on 03/06/2016).

[ŚB18] Błażej Święcicki and Leszek Borzemski. “How Is Server Software Configured? Examining the Structure of Configuration Files”. In: *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017: Part I*. Springer International Publishing, 2018, pp. 217–229. ISBN: 978-3-319-67220-5. DOI: 10.1007/978-3-319-67220-5_20.

[She+05] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. “ACMS: The Akamai Configuration Management System”. In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 245–258. URL: http://dl.acm.org/citation.cfm?id=1251203.1251221.

[Spi12] D. Spinellis. “Don’t Install Software by Hand”. In: *Software, IEEE* 29.4 (July 2012), pp. 86–87. ISSN: 0740-7459. DOI: 10.1109/MS.2012.85.

[SW65] S. S. Shapiro and M. B. Wilk. “An Analysis of Variance Test for Normality (Complete Samples)”. In: *Biometrika* 52.3/4 (1965), pp. 591–611. ISSN: 00063444.

[SWG16] Rian Shambaugh, Aaron Weiss, and Arjun Guha. “Rehearsal: A Configuration Verification Tool for Puppet”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA: ACM, 2016, pp. 416–430. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908083.

[Świ16] Błażej Święcicki. “A Novel Approach to Automating Operating System Configuration Management”. In: *Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology – ISAT 2015 – Part II*. Cham: Springer International Publishing, 2016, pp. 131–142. ISBN: 978-3-319-28561-0. DOI: 10.1007/978-3-319-28561-0_10.

[SWI17a] SWIG Maintainers. *Simplified Wrapper and Interface Generator*. 2017. URL: http://www.swig.org/ (visited on 04/19/2017).

[SWI17b] SWIG Maintainers. *SWIG-3.0 Documentation*. 2017. URL: http://www.swig.org/Doc3.0/SWIGDocumentation.html (visited on 04/19/2017).

[Tan+15] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. “Holistic Configuration Management at Facebook”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, 2015, pp. 328–343. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815401.

[Tsa10] Aleksey Tsalolikhin. *State of the Art of Automating System Administration with Open Source Configuration Management Tools*. July 9, 2010. URL: http://www.verticalsysadmin.com/config2010/ (visited on 04/12/2016).

[Tur07] James Turnbull. *Pulling Strings with Puppet. Configuration Management Made Easy*. Apress, 2007. ISBN: 978-1-59059-978-5. DOI: 10.1007/978-1-4302-0622-4.

[Van14] Bart Vanbrabant. “A Framework for Integrated Configuration Management of Distributed Systems”. Ph.D. dissertation. Faculty of Engineering Science, KU Leuven, June 2014. URL: https://lirias.kuleuven.be/handle/123456789/453199 (visited on 03/08/2016).

[Ven13] Paul Venezia. *Review: Puppet vs. Chef vs. Ansible vs. Salt*. Nov. 21, 2013. URL: http://www.infoworld.com/article/2609482/data-center/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html (visited on 03/08/2016).

[Wik17] Wikipedia. *INI file*. Feb. 2017. URL: https://en.wikipedia.org/wiki/INI_file (visited on 04/28/2017).

[Woh+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Sortware Engineering: An Introduction*. Kluwer Academic Publishers, 2000. ISBN: 0-7923-8682-5.

[Yin+11] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. “An Empirical Study on Configuration Errors in Commercial and Open Source Systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 159–172. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043572.