Bachelor Degree Project

UNIVERSITY OF SKÖVDE

Software Configuration Management

A comparison of Chef, CFEngine, and Puppet

Bachelor Degree Project in Computer Science
G2F, 15 ECTS
June 21, 2012

Fredrik Önnberg
a09freon@student.his.se
Network- & Systemadministration ’09

Spring term 2012

Supervisor: Helen Pehrsson
Examiner: Gunnar Mathiason

Abstract

Configuring services in an ad-hoc way is less than optimal as human error can result in services that fail. Even though scripting offers a solution it does not allow for a uniform deployment of configurations. A possible solution to this problem is the use of software configuration management systems (SCMS) that allow administrators to specify what should be done, and not necessarily how. An implementation is conducted that focuses on the first part of implementing an SCMS to see if a transition to a SCMS governed environment can be worthwhile. Three SCMS will be investigated; *Chef*, *CFEngine* and *Puppet*. The results show that administrators can receive good support from the documentation as documentation of the three SCMS are mostly accurate. The support from the community is affected by region and activity is declining in some cases. Overall it is easy to install an SCMS and special purpose languages are effective for specifying functionality of services.

Keywords: Puppet, Chef, CFEngine, Implementation phase

Contents

1 Introduction **1**

2 Background **2**
2.1 Reference literature . . . 2
2.2 Defining Software Configuration Management . . . 3
2.3 Scripting and SCMS . . . 3
2.4 System Configuration Management Software . . . 5
2.4.1 Puppet . . . 6
2.4.2 Chef . . . 7
2.4.3 CFEngine . . . 8

3 Problem **9**
3.1 Purpose . . . 9
3.2 Motivation . . . 9

4 Method **11**
4.1 Objectives . . . 11
4.2 Literature analysis . . . 11
4.3 Experiment or Implementation . . . 12
4.4 Alternative methods . . . 12
4.5 Case study . . . 13
4.5.1 Survey . . . 13
4.5.2 Limitations . . . 13

5 Implementation and Results **15**
5.1 Attributes . . . 15
5.1.1 Documentation . . . 15
5.1.2 Community strength . . . 15
5.1.3 Ease of installation and dependencies . . . 16
5.1.4 Languages . . . 16
5.1.5 Compliance to ISO/IEC 90003 . . . 16
5.1.6 Attributes not examined . . . 16
5.2 Installation and dependencies . . . 17
5.2.1 Puppet . . . 17
5.2.2 Chef . . . 18
5.2.3 CFEngine . . . 18
5.3 Community strength . . . 18
5.3.1 Puppet community . . . 19
5.3.2 Chef community . . . 19
5.3.3 CFEngine community . . . 20
5.4 SCMS languages . . . 20
5.5 Documentation . . . 21

6 Analysis **24**
6.1 Dependencies . . . 24
6.2 Community . . . 24
6.3 Language used . . . 25

6.4 Documentation 26
6.5 SCMS and ISO/IEC 90003 27
6.6 Related work 27

7 Conclusion **29**

8 Discussion **30**
8.1 Ethical and social aspects 31

9 Future work **32**

References **33**

1 Introduction

As todays computer networks proliferates into complex environments, the management of the networks also becomes increasingly challenging. A system administrators role is not only to ensure that systems do not falter but to try and maintain some sort of predictability (Burgess, 2003). There are many reasons as to why systems fail. Even the most robust computer systems can fall victim to human error and some environments are more prone to this than others (Couch & Daniels, 2001). One common approach of administering has been to let the systems fail and then reset them. This approach is troublesome as reliability and security are compromised (Burgess, 2003). It can take a long time for administrators to troubleshoot and reset the systems. Some times a cause of a crash might not be found at all, which results in a possibility for another crash that can not be prevented. While it is possible to create decision trees to help with troubleshooting, they are an ineffective method (Couch & Daniels, 2001).

It is almost impossible for an administrator faced with a problem to cover all bases when creating solutions that are dynamic enough to be able to put a system into a certain state without knowing the precedent state. Because of this, software configuration management systems (SCMS) are very coveted as they aim to take a service from one state to another without forcing the administrator to execute each step in between (Estublier et al., 2005). The SCMS try to automate configuration management by allowing administrators to specify how they want the server to perform. This, in contrast to manual configuration, is effective since the administrator often does not have to bother about the starting state of the server. It also means administrators can specify how the server is supposed to work without bothering about how to get there. SCMS are not only helpful during troubleshooting, but also while deploying new services. One important aspect of using SCMS is that it allows for a uniform configuration method. An SCMS will automatically configure the systems by using native commands for each operating system that is supported. This means that an administrator does not need to learn how to use each operating system that is used inside a network as the SCMS will translate the commands into the proper commands for the target environment.

Software configuration management is a broad spectrum of topics. This thesis will focus on the first part of using an SCMS, the implementation. While SCMS aims to streamline and automate configuration management, it is not the only way of doing it. For this reason, it will be briefly related to scripting as scripting is a common way of administering services. Small to medium-sized organisations might not deem it worthwhile to implement an SCMS since it might require extra time and educational costs for administrators.

Different softwares have different functionality thus putting administrators in a position where they have to choose one that fits their environment. The general lack of scientific research of software configuration management software creates an issue for administrators when attempting to choose the right software. This thesis examines how general and mature the current platform independent SCMS are. This information can be used by administrators to gain knowledge of how an adaptation of an SCMS may affect their work in the computer network.

2 Background

Computer systems in organisational networks are bound to run many different services. There are often variations in the networks complexity, both structural and in how many different services are run. As organisations grow, the demands of the network devices increases. Even though the number of services might not change, any growth of the network will require increased administration. Deploying and managing services can be a very time-consuming and error-prone activity. Some services require configuration files to be put in correct locations and it is also a requirement to check that different components are compatible. All the manual labour comes with a risk of introducing faulty setups due to human error (Dolstra, Bravenboer, & Visser, 2005). To have manual configuration of services is less than optimal and may cause unintentional errors. Therefore, it is important to have a good configuration management practice.

The SCMS tools are meant to assist administrators in their work of both deploying new servers and configurations as well as assist with restoring software to their previous states. There are several SCMS tools available and this makes it hard for administrators to single one of them out that will fit the administrators specific needs. Some of the SCMS are more commonly used than others. By looking at message boards relevant to the subject on the Internet, it was possible to deduce that the most commonly used SCMS are Puppet, Chef and CFEngine. While there are a lack of scientific comparisons between the softwares there are documentation available for each SCMS. However, it is hard to build an opinion about the softwares based solely on documentation.

2.1 Reference literature

The main source of information are scientific papers. The scientific papers that are of interest for the work in this thesis are peer-reviewed articles that have conducted studies in the field of configuration management. From such articles it is possible to gather information about important aspects in configuration management. However, since there are few scientific sources, such as (Burgess, 2003), about the administration part of software configuration management many sources of information are from regular administrators and users. This information is gathered on Internet message boards and articles that voice concerns or general opinions about the softwares. The problem with this is that it is not an optimal source of information. The Internet is not necessarily a bad channel of information but message board posts and articles may be subjective and may not accurately reflect on the topics. Whenever such a source is used in this thesis it is clearly stated. The scientific value of a message found on an Internet message board is up for discussion. At the same time as it is not a highly regarded source of scientific information, it is a good compliment to traditional sources. The community-based information is valuable in the way that it can reflect the opinion of the users and also shows what the users say should be improved or even changed. Such opinions are often what administrators use in their evaluation when researching software and are therefore important when trying to identify bad and good aspects of softwares. It is because of these community-based contributions that such sources are included in the thesis.

2.2 Defining Software Configuration Management

Configuration management is often discussed when talking about the development of software, but it is not only during development it is needed. Unfortunately, there is no complete definition of what a SCMS should involve. The ISO/IEC 90003 (International Standards Office, 2004) does however mention what the scope of configuration management should include:

- planning of the process including defining activities, responsibilities and the tools to be procured;
- identifying uniquely the name and versions of each configuration item and when they are to be brought under configuration control (configuration identification);
- identifying the build status of software products under development, delivered or installed, for single or multiple environments, as appropriate;
- controlling simultaneous updates of a given software item by two or more people working independently (configuration control);
- providing coordination for the updating of multiple products in one or more locations as required;
- identifying, tracking and reporting of the status of items, including all actions and changes resulting from a change request or problem, from initiation through to release (configuration status accounting);
- providing configuration evaluation (status of verification and validation activities);
- providing release management and delivery.

Each of these individual parts are specified in the ISO/IEC 90003 (International Standards Office, 2004). They point toward a clearly defined structure for managing both deployment and configuration of software. The scope mentions that it is required to uniquely identify both the name and version of each configuration item. Since a software can have many incremental updates throughout its lifetime it is of interest to be able to uniquely identify each version. One effective way of managing these different versions and states are with the help of revision control. Revision control makes it possible to track and even revert changes if something unwanted happens. Each update, both small and large, can have significant impact on the function of the software. While the ISO/IEC 90003 mostly apply to the development of software it should be noted it can be applied to administration of configurations. Having a uniform and standardized way for software configuration management is beneficial and the ISO/IEC 90003 are relevant for administration of configuration.

2.3 Scripting and SCMS

One way of trying to circumvent human errors are to manage configurations with the help of scripts. Scripts often uses conditional statements that allow scripts to generate an output that depend on input given the script by an administrator, or input generated by the script itself. Scripts can be powerful tools when configuring services and

performing small tasks in network environments. They allow for automating parts of the configuration thus reducing the risk of human mistakes (Nemeth, Hein, Snyder, & Whaley, 2010). Even though scripts can be used to minimize mistakes, it does not protect against semantical errors. If an administrator makes a mistake during the construction of a script, there is no way of preventing that mistake from being introduced to the system if the script is executed. A script that is thoroughly examined and tested will however help in keeping a consistent behavior when applying patches and configuration files. However, to use scripts the administrator must know both the current state and then the new intended state of the service. A state is a specific set of rules that together allow the service to carry out a specific function. This knowledge about states applies both to deploying new services as well as modifying an existing service. When deploying a new service, an administrator will have to consider what software is installed on the operating system. Knowledge about additional software and the state of the service must be taken into consideration when writing scripts so that all pre-requisites are met before trying to deploy a configuration. Such a pre-requisite can be that the software that the configuration is meant to be a part of, or a dependency for a software, needs to be installed.

Knowledge of the new state might not be an issue since it has most likely been researched in order to establish the need for it. Even though the structure of the new state might be known, the transition from the current state to the new state can be troublesome. There are also occasions where services have dependencies that might be unknown that will cause issues in the transition phase. A model of transition from one state to another can be seen in figure 1.

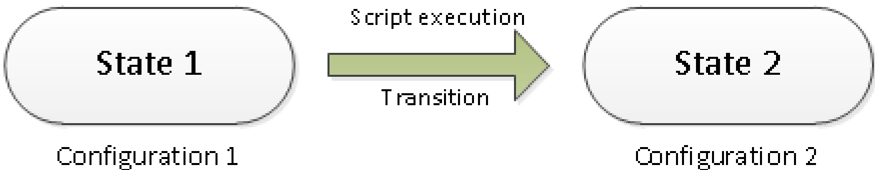

Figure 1: Transition phase

A common way of distributing updates and softwares are with the help of packaging software. These packages contain a large amount of files, each with its own configuration. If an administrator were to simply extract the contents of such a package it could completely overwrite any existing configuration in favour of the packaged content (Burgess, 2011). With the help of SCMS it is possible to govern such updates in order to make sure nothing is overwritten or lost. Still, when updating a single server it is possible to write a small script that updates the chosen part and then terminates. One can ask if it really is worth the investment in a SCMS instead of writing the occasional script as the learning curve have been reported to be steep (Rosin, 2011). This is especially relevant for small to medium sized organisations where the administrators time might be very valuable. In these organisations it would be a huge potential improvement if a SCMS proved to be the way to go even for smaller organisations. For big organisations the gain can be tremendous as the value of an SCMS will increase with the amount of services and servers. With a huge number of services to manage, it is simply too unpractical to manually manage each and every one of them. Even with scripts it can become tedious since the services might need different kinds of config-

uration. This is where SCMS, in theory, really would help by grouping services and servers and applying different configurations depending on which group it belongs to. However, it is still an area not well explored and therefore require some initial investigation of the most rudimentary functions of the SCMS.

Some other aspects of SCMS are that there should exist release management and configuration evaluation. As mentioned earlier, human error is common and as the SCMS are designed to recognise the operating system used it will only use commands and settings that can be found on that operating system. This means administrators does not have to keep track of which operating system that are on the servers. Furthermore, administrators only needs to learn the syntax used by the SCMS as it will take care of executing the proper commands. This will alleviate the issue of applying faulty configurations since the SCMS will warn if administrators uses improper commands. By having a proper delivery management it is also possible to streamline how configurations are delivered. A SCMS will allow for greater control by governing the configuration delivery.

2.4 System Configuration Management Software

In this section, different SCMS will be presented. It will first describe operating system support by several SCMS. Then it will give an introduction to the architecture and mechanics of the SCMS this thesis will focus on.

There are quite a few different tools in existence for automating configuration management. This thesis will focus on the SCMS that have support for all major server operating systems used in computer network environments. At the time of writing, this is *Puppet*, *Chef* and *CFEngine* as can be seen by table 1.

	AIX	**HP-UX**	**Linux**	**BSD**	**Mac OS X**	**Solaris**	**Windows**
Bcfg2	/		X	X	/	X	
cdist			X	X	X		
CFEngine	X	X	X	X	X	X	X
Chef	X	X	X	X	X	X	X
DACS			X	X		X	
etch			X	X		X	
LCFG			/		/	/	
Opsi							X
Pallet			X		X		
Puppet	X	X	X	X	X	X	X
Quattor			X		X		
Salt			X	X			/
Spacewalk			X			X	
X = Supported, / = Partial Support							

Table 1: List of SCMS and their supported platforms

By having only one SCMS in the environment, it is only necessary for administrators to learn one system. A problem that exist in many environments is that there are many different softwares that each have their own specific functionality. This mean that if several SCMS are used in an environment to control specific operating systems,

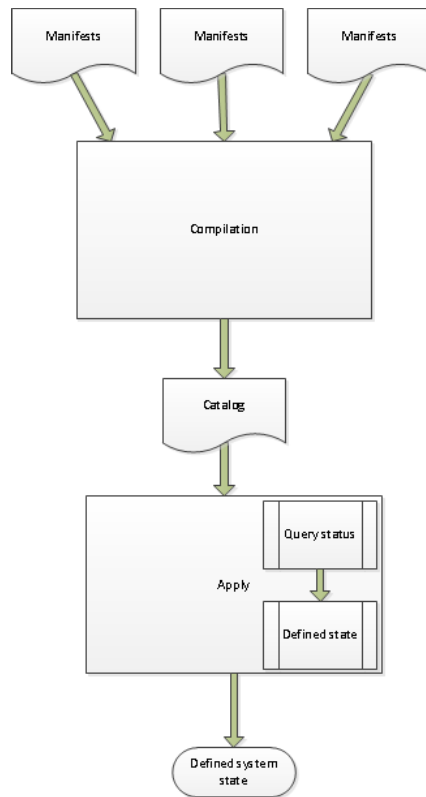

Figure 2: Puppet work flow

much of the advantage of using an SCMS is lost. While using several different SCMS to control different operating system enables for a uniform management within the bounds of the SCMS, it does not give a uniform management for the entire environment. By using one SCMS with support for all the major operating systems it results in a uniform way of operation throughout the entire environment. It also leaves administrators with only one SCMS to learn instead of a variety of SCMS, each with their own syntax and functionality.

2.4.1 Puppet

There are several ways of configuring *Puppet* such as server-client or serverless environments. In the server-client environment the server is called *Puppet master*, the client software is called *Puppet agent*, and the clients themselves are called *nodes* (Puppet Labs, 2011). There are different resources that *Puppet* uses in order to manage *nodes*. One of these resources are called *manifests* that contain specific information regarding configuration settings that will be deployed to its *agents*. The *manifests* are compiled into a *catalog* which is a directed acyclic graph that represents resources and how they need to be synced (Puppet Labs, 2012a).

The work flow process of *Puppet* can be seen in figure 2. When using a *Puppet master* it will take care of the compilation of resources and then apply it to the *Puppet agents*. If no *Puppet master* is used the *Puppet agent* can compile and apply the resources by itself (Turnbull, J. and McCune, J., 2011). In the same manner it is also possible to have a *Puppet master* control itself, which makes the *Puppet master* a part of the uniformly managed environment, and not some peripheral device.

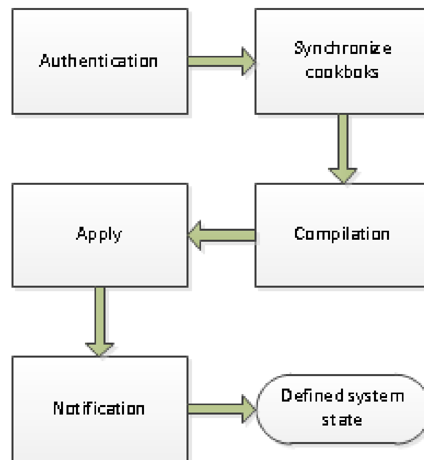

Figure 3: Chef work flow

2.4.2 Chef

Chef is, just as *Puppet*, able to function in both server-client and serverless environments. The server is simply called *Chef Server* and the client is called *Chef Client*. The serverless envronment is called *Chef Solo* (Steinglass & Thomas, 2011) and allows clients to compile and apply resources by itself. The resources used in *Chef* are called *Recipes* and *Cookbooks*. The *recipe* is a file containing configuration settings and the *cookbook* is the collective name of different recipes, together with other configuration files (Thomas & OMeara, 2012). The work flow in *Chef* resembles the one in *Puppet* and can be seen in figure 3.

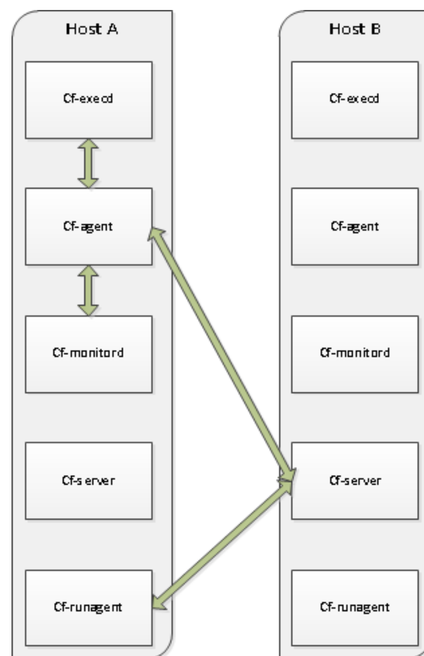

Figure 4: CFEngine communication

2.4.3 CFEngine

CFEngine clients can act in two modes, client or as a multinode structure (CFEngine, 2012a). When operating as a multinode structure, some clients will be configured as a *policy server* while others are normal clients. The clients contacts the *policy servers* that in turn are responsible for the distribution of *policies*. A *policy* is a collection of *promises*, which in turn are used to describe the desired state of a system (CFEngine, 2012b).

When a *policy server* has published a *promise* the *clients* can keep themselves updated by fetching new *promises* by using a pull-function. Each node in a *CFEngine* network are independent actors and works opportunistically which means that even if a *policy server* disappears, it will keep running by doing what it can with what is left. The communication of nodes can bee seen in figure 4. There are many different parts within a host, each with its own task. The components that can be seen in figure 5 are the responsible parts of applying changes to nodes. As an example, the `Cf-agent` is responsible of applying changes to nodes while `Cf-execd` is the scheduling daemon. The components are also able to work independently and communicate with each other as needed.

3 Problem

It is possible to use scripts, or manual labour, for configuration management. However, as discussed in the previous section, scripts can be unreliable and tedious to manage. Manual labour can become an issue due to its error prone nature. As mentioned by Dolstra et al. (2005), it is important to have good configuration management practice. With good configuration management, the authors talk about the difficulty of installing all dependencies and making sure each component is compatible with other software. The authors also mention that there are negative implications of manually handling configuration as some important aspects are lost, such as derivation management (Dolstra et al., 2005).

Due to the nature of ad-hoc configuration management, a better environment for deploying and managing software configurations is needed. Using an SCMS could solve this as they are meant to give a uniform environment for creating, managing, and deploying configurations.

3.1 Purpose

This thesis has tested the softwares, *Chef*, *CFEngine*, and *Puppet* as they support all the major operating system, as described in section 2.4.

The purpose of this thesis will be to compare and contrast three of the available SCMS. By doing this comparison, this thesis aims to identify if SCMS are both general and mature enough to be implemented by a wide variety of organisations. By being general, the SCMS must have support for many different operating systems as well as services. Furthermore, by implementing an SCMS, administrators should not have to sacrifice functionality of the servers and their services. A mature SCMS means that it has a good support structure such as good documentation and a supportive community.

3.2 Motivation

As mentioned in the previous chapter, an investment in one of the SCMS-tools can be costly. It is important that the tool can perform as desired in the network and it is also important that the tool can be utilized in an evolving network. The choice can be a burden on the administrator since the many tools have their own strength and weaknesses. This thesis gives an indication of generality and matureness that may help administrators when choosing an SCMS. There are some documentation available scattered on Internet forums and the developers own websites. Visiting online message boards in search for, sometimes very specific, questions about performance and functionality is very time consuming. Another problem is that developers tend to be biased towards their own products. Though a product might be of high quality and praised by its developers, it is still not necessarily the right one for everyone. The bias also makes it hard to consider information gathered from the developers website objectively. Even though developers may present a list of features on their website it may be embellished or exaggerated.

When searching on the Internet about the different strong and weak points of each of the SCMS-tools, there are only scattered pieces of information. Each of these softwares promise easier configuration management and unique qualities. This does not

make it easy on the administrator when choosing an SCMS that would fit into the environment and his specific needs. Choosing a SCMS is a big investment since some of them are said to have a steep learning curve (Rosin, 2011). They also need to be implemented into the environment which can be a time consuming task, especially if discovered that the chosen SCMS did not fit or perform as expected. Even though there are some places where information can be found about the different SCMS, it is hard to filter out the real information from all subjective forum posts. The investigation in this thesis examines the three available platform independent SCMS to try and get as generalizable results as possible.

As there are many different SCMS, the chosen ones to investigate are the ones that are well-discussed, have a wide supports of operating systems, and used by many; *Puppet*, *Chef*, and *CFEngine*.

4 Method

In this section, the objectives needed to answer the thesis question are explained. After the objectives are established, the methods used for each objective are explained together with a reason as to why the methods have been chosen. An iterative approach is taken to the method throughout the entire work. The last part of this section contains alternative methods with an explanation as to why they were not chosen.

4.1 Objectives

In order to be able to give an answer if the SCMS can be generally applicable, there are a few objectives along the way. The first step is to find the right attributes to test, which is important as this is the basis of finding an answer to the problem posed in section 3. The next steps are to conduct the implementation based on the attributes and then analyze its results. Lastly, a conclusion of the work as a whole is made. The following list shows the objectives for the work:

- Find attributes to test
- Conduct implementation based on attributes
- Analyze the results from the implementation
- Draw conclusions from the work

4.2 Literature analysis

In order to establish the attributes that are tested, a limited literature analysis is performed. The literature analysis is not exhaustive as the purpose is to find attributes that can be used to measure generality and maturity. Some risks that literature studies should take into consideration are completeness and validity (Berndtsson, Hansson, Olsson, & Lundell, 2007). A systematic approach is taken to the literature analysis in order to minimize the inherent risks of the method. Even though the literature analysis is limited and is not intended to be exhaustive, it is important to not disregard validity nor completeness as the literature analysis is the base of finding correct attributes. To try to counter the issues with validity and completeness, an iterative process is used during the work for this thesis that re-examines articles and finds new sources of information as the works progresses. By re-examining old articles after having made progress it is possible to identify if the articles actually are relevant and can continue to be used. It is also possible to identify if there are aspects that have not been previously identified and thus needs to be investigated further. However, the validity and completeness risks can never fully be mitigated and are constant issues with the method.

The sources chosen for the analysis are articles that discuss configuration management. There are research done about configuration management in software engineering that are relevant as it discuss the theory behind configuration management. Such research may have established core aspects of configuration management that should not be disregarded when examining configuration management software. The output from the literature analysis are attributes that are used to analyse the SCMS in regards to generality and maturity.

4.3 Experiment or Implementation

From the literature analysis there are attributes that are relevant to examine with the help of a practical test. By looking at the methods mentioned by Berndtsson et al. (2007), the relevant methods to choose from in this scenario are either implementation or experiment. According to Berndtsson et al. (2007), the goal of an implementation is to demonstrate that a solution has certain properties or behaves a specific way under certain conditions. While this accurately reflect the goal of the practical tests needed for this thesis, Berndtsson et al. (2007) also mention the implementation method in relation to developing new solutions for problems which raises the question if the method is relevant for the work in this thesis.

Berndtsson et al. (2007) say that an experiment typically is used to verify or falsify a previously formulated hypothesis. As there are no hypothesis that needs to be verified or falsified, this method is not completely applicable. In contrast to implementations, experiments are designed to prove or disprove an hypothesis while implementations test functionality under certain conditions. As the practical tests done in this thesis largely test functionality and properties reported by developers and users it is the implementation method that fits best.

The environment used in the implementation contain one server of each SCMS that is set to deploy configurations to different groups of clients, as illustrated by figure 5. Since the experiment is used to test functionality and not to simulate a real environment, the test environment is set up in a smaller scale. The installation of the SCMS is based on the available documentation and guidelines from the developers. The implementation is used to test relevant attributes derived from the literature analysis.

The data from the implementation phase are examined in order to find similarities and/or differences between the SCMS. The similarities and differences are used to draw conclusions if the different implementation methods of the SCMS can be seen as general functions or something software specific. The data gathered is compiled into data sheets containing tables, graphs and other forms of data representation.

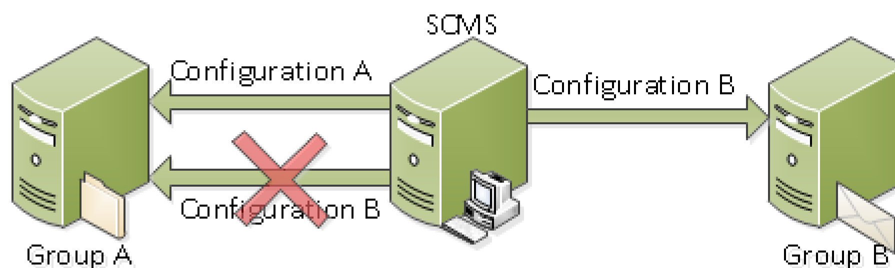

Figure 5: Deploying configuration to correct group

4.4 Alternative methods

In this section, alternative methods to the method chosen for this thesis are presented.

4.5 Case study

A case study is an in-depth examination of a limited number of cases (Berndtsson et al., 2007). A case study can use scenarios from organisations that have already implemented a solution with one of the platform independent SCMS. A case study can make use of the experience of administrators and gather information about aspects that they find important.

The reasons why a case study was dismissed was because of organisations particular needs. A case study can identify aspects that one organisation may find important, but other organisations may find the same aspects irrelevant. Another problem is to actually find organisations that have implemented the SCMS investigated in this thesis that are willing to participate and provide answers for questions that may arise. Some organisations might not be keen on releasing certain information. This means that the aspects identified by a case study may not be generalizable enough to answer the question posed in this thesis.

4.5.1 Survey

According to Berndtsson et al. (2007), a survey is a research method based on questionnaires and statistical analysis. By creating questionnaires that involve questions about functionality in a computer network it may be possible to identify the generality and matureness of the SCMS. The questionnaires can be sent out to different target groups in order to get a good overview of the functions needed from the SCMS in order to cover the administrators needs.

The problems that surveys have are the same as case studies. The results from the surveys might only reflect the individual needs of administrators. The amount of surveys and questions needed in order to be able to answer the question posed in this thesis are so many that it is an unrealistic method. To get good results from a survey, many organisations must participate and it is unlikely that enough organisations are willing to participate. Some questions in a survey may also require organisations to give information that the organisation can not share, which in turn results in an incomplete survey. The answering frequency needed is likely to be so high that a survey will not give results that can be used.

4.5.2 Limitations

There are limitations in the number of systems and softwares tested. The environment is limited and consists of one server and two clients. This environment contains enough hosts to use the functions of the SCMS properly. A larger environment would simulate real computer networks more accurately, but would not give any additional value to the implementation as two clients are enough to check the fundamental functionality of the SCMS.

As have been mentioned in section 2.4, the SCMS need to fulfill some requirements; the SCMS needs to have supports for all major server operating systems.

The implementation in this thesis is based on a set of attributes which are in relation to the first part of the implementation of a SCMS. This means that it is the most relevant to administrators that has yet to implement a SCMS environment. There are no investigation of the security implementations of the SCMS. To justly compare the security implementations, deeper research of security needs to be done. The actual

functionality within other operating systems than Debian are not tested as functions in the SCMS are the same regardless of platform.

5 Implementation and Results

In the following sections will be a description of the environment and installation of each SCMS.

As described in section 3.5, a server is installed with the SCMS. All machines use Debian Squeeze 6 and are virtualized on VMWare ESXi 5.0 with 1024 MB of dedicated RAM. The systems are installed using default settings and any non-default setting are reported wherever appropriate.

5.1 Attributes

This section describes the attributes that have been identified by the literature analysis.

5.1.1 Documentation

Part of the reason to choose SCMS over other solutions are based in satisfaction with the configuration management system. As ad-hoc configuration can result in errors and difficulties, it is likely that there are dissatisfaction among the administrators. It has been shown that documentation is an important satisfaction factor for information systems (Shaw, DeLone, & Niederman, 2002). As administrators often use documentation when researching implementations and solving problems it is important that the documentation both are correct and give the support that the administrators need. If the administrators are unhappy with an environment controlled by SCMS, it may overshadow the potential gain the SCMS would bring in configuration management. By consulting the documentation while performing the implementation it is possible to see if the documentation are correct and contain the necessary information for getting started. This shows if any parts of the documentation are missing or incorrect and also indicates how mature the support structure of the SCMS are. The information presented in the documentation can also indicate if it contains enough information to help administrators regardless of previous experience.

5.1.2 Community strength

While documentation is a valuable asset for administrators, it is likely it does not cover all areas that administrators need. One resource that are very utilized by administrators when configuring and managing services is the community. To get help when having issues, administrators may contact the relevant community and pose a question and hope to get some input that will help solve the issue. There have been evaluations where puzzles was solved faster in groups than alone (Heldal, Spante, & Connell, 2006). This indicates that the community can be a very helpful resource as the collective intelligence can help in solving problems faster than the lone administrator. Community contributions can be used by administrators to complement their own solutions when using SCMS as the resource structure is modular. By looking at how active the community are, it indicates if the collective intelligence can be used in an environment that utilizes SCMS. A big and active community indicates that the software is mature as it has gained support from the users.

5.1.3 Ease of installation and dependencies

By checking for issues in the installation process it is possible to identify if a change from scripting or manual labour to an environment controlled by SCMS can be troublesome. If administrators need to prepare their system it should be clearly stated either during the installation or in the official installation documentation. Another parameter that is investigated is how dependencies play a role during the implementation of the SCMS. A dependency is a software or library that need to be installed or configured as the main software relies on its functions. Conflicts between dependencies or even the SCMS may appear and cause issues for administrators and it is thus important to have a good overview of the dependencies installed. Dependencies can also be potential security risks as they possibly can be exploited to gain control of the system or infect systems with malicious software. During the installation, each package installed is counted. This, in order to find out how many dependencies the SCMS need in order to be operational. By checking the amount of dependencies it shows the amount of extra software and libraries that must be installed in order to run the SCMS. As there are two modes for the SCMS to operate, client or server, the dependencies are also separated into these modes. Dependencies required for the client but not for the server will thus not be listed as a requirement for the server. Investigating the installation process and the dependencies indicates if any special measures need to be taken before installing a SCMS.

5.1.4 Languages

An investigation of the different types of languages that are used in the different SCMS and which components they contain is conducted. The language is the core component of the SCMS that administrators use to create the resources that controls the environment and it is thus important that the language allow the administrator a good control. There have been been reports of steep learning curves (Rosin, 2011) which is a negative incentive of adopting the SCMS with a steep learning curve. Therefore, the languages are compared and analysed as to highlight the advantages and disadvantages between the types of languages used. The language analysis indicates diversity of the languages and how effective they are at creating resources.

5.1.5 Compliance to ISO/IEC 90003

Lastly, it is examined how close the SCMS follows the concepts of configuration management. An ISO/IEC standard exists that describes the theory behind configuration management. By checking if the theory behind configuration management have been implemented into the SCMS it shows that the SCMS works towards a general way of delivering configurations. During the implementation, functions of the SCMS are correlated to the standard to see if the SCMS follows the theory of configuration management. A comparison to the ISO/IEC 90003 indicates how much of the theory behind configuration management that have been directly implemented in the SCMS.

5.1.6 Attributes not examined

Some of the attributes identified have been left out of the comparison. One of these attributes is the support for different services. As the SCMS take care of configurations

for a number of different services it is of importance that the services have support in the software. However, the SCMS have been designed to allow administrators to configure any type of service as the control for this is on an abstract level where the SCMS extend into package managers or file structures. Due to this, the number of services supported by the SCMS are the same as those that exist in package managers or able to be deployed by other mechanisms. It is not the SCMS that limits the type of services and the number of services supported are therefore not used as an attribute in the tests.

Another attribute that has been left out are the security implementations of the SCMS. The literature analysis did not identify any aspects of the security implementations that limit generality or matureness of the SCMS. In order to justly compare the security implementations it would require a deeper study of security than the question posed in this thesis allow for.

The support for the operating systems are not checked beyond installation. It is tested that the SCMS can be installed on multiple platforms, but the actual functionality within the platform is not tested as it falls outside the limitations for this thesis.

5.2 Installation and dependencies

In this section, the results correlated of the attributes *Ease of installation* and *Dependencies required to run the SCMS* are presented.

The installation of SCMS was done iteratively where each SCMS was installed several times. The SCMS was installed after each other in cycles so that knowledge could be carried over to the next installation. With the help of the iterative process, each SCMS was examined multiple times in order to prevent any knowledge gained from one installation to skew data from the next installation.

During installation of software it is not uncommon that the user have to answer questions about which functionality the software should be installed with. An example of this is the installation process for `Postfix` in some Linux-environments where the user can select one of multiple options that then defines the default configuration of the software. If the user choose one of the options without considering the impact of the choice, it is possible that the software must be reconfigured in order to achieve the functionality that the user actually wanted. By displaying hints and explanations about such options it is possible to help the user avoid issues that requires the user to reconfigure the software or in some cases, require the user to recompile the software.

There are several ways of installing the SCMS. It is possible through the use of a package manager such as *Aptitude*, by using *Ruby Gems*, by using a tarball, or by precompiled binaries. The method chosen in this thesis is to use the package manager, if possible, as a package manager helps to improve the control of installed packages on an operating system. The `main`-repository contained old versions for both *Puppet* and *Chef* and the `sid`-repository must be used as it contain newer versions. The installation process was done with the help of the instructions on the official documentation.

5.2.1 Puppet

When installing the *Puppet Master* there are in total 183 packages that should to be installed. There are some packages that are not real dependencies, such as `apache2` and `vim-puppet`. However, as these packages offer extended functionality they will

be counted towards the total number of packages to be installed. The *Puppet Agent* need 11 packages to be installed. There are no extra configuration to be done during the installation and the package manager takes care of the installation completely.

The installation of *Puppet* did not ask for additional information. After telling the package manager to install, the installation completed successfully without errors. There are no hints after the installation as to which configuration file to turn to in order to begin setting up the *Pupper Master*. The main configuration file `puppet.conf` does have a *man-page* that lists arguments that can be used as parameters to *Puppet* executables. The *man-page* can thus be used as a helping tool when starting to configure *Puppet* for use, otherwise it is the offical documentation that has to be resorted to. In order to find the official documentation, a visit to the homepage of *Puppet Labs* (Puppet Labs, 2012b) will reveal a direct link that will take users to a web page containing information about installation, configuration and also tips to improve functionality. The official documentation contains relevant information about the post-installation preparation and will guide the user through the steps. Furthermore, it gives an introduction to the syntax of the special purpose language used by *Puppet* and helps administrators when writing the first manifest. The documentation is explained further in section 4.5.

5.2.2 Chef

As in the case of *Puppet*, a package manager was used to install *Chef*. Before installing *Chef* however, a GNU Privacy Guard-keyring need to be installed. With the help of the keyring, the Opscode repository is added to the operating system. There are no need to configure anything during the installation of *Chef*. The total amount of packages to be installed on the Chef server are 265 and on the agent it is 67.

The official documentation is a wiki page that can be accessed from the Opscode Chef home page. It consists of many different articles about different parts of Chef. The documentation does not only give information about how to install or information about the architecture. There are also links to guides about how to use *Chef* to install services. The guides give a more hands-on approach of how to use *Chef* and is a good complement to the other parts of the documentation which is more introductory in nature. The documentation is described further in section 4.5.

5.2.3 CFEngine

For installing *CFEngine* the official documentation from *CFEngine* provides a link to pre-compiled binaries. It is possible to use a repository and a package manager to install as well. Both ways of installing was tested and in the case of using the package manager it was not only an old version in the repository, but it did not follow the file structure as mentioned by the documentation. The pre-compiled binary followed the documentation well and installed without any problems.

There are few dependencies that need to be installed before *CFEngine*. In total there are only 3 dependencies that are required in order to run *CFEngine*.

5.3 Community strength

In this section, the results corresponding to the attribute *Community Strength* are presented.

The community investigation start with the SCMS official home pages as starting points. From there, links were followed that led to other parts of the community. Each link was followed and further investigated to establish relevance and content. The process is done iteratively in order to minimize the risk of missing important links.

The investigation of the community channels were performed during a one week time period between 23 april 2012 - 29 april 2012. During this period, chat channels were visited and posts on several community channels were monitored to measure the response times. As with many ofter softwares, an active community exists for all SCMS. It is not only end-users that are active participants. The SCMS have been out for a different period of time which means that the communities have had different amount of time to grow. *Puppet* was created in 2005, *Chef* was created in 2009, and *CFEngine* began in 1993. There are many developers that share their knowledge with the community. While it is possible to stumble upon sources while searching the Internet for information, the official home pages for the SCMS contain links to some of the sources. All three of the SCMS contain information about how to find the community sources. Looking at the home pages and checking what the recommended community channels are, they differ slightly among the SCMS.

5.3.1 Puppet community

Puppet have several options when contacting the community. There are newsletters, irc-channels, mailing lists, and other media such as video and presentations. At the time of writing, in the "Pupper Users"-mailing list there are 3998 subscribed members and 7169 topics. Since 2008 there have been an average of 678 submissions per month to the mailing list. It is possible to read the content of the list, but not submit new posts without being a subscribed member. It is likely that more people read the group than those that are subscribed members as the content are open to the public. There are more mailing lists available that target other user groups such as enterprise users or developers. There are 761 members subscribed to the mailing list for developers with 5616 topics. Just as the "Pupper Users"-mailing lists it is open to be read by anyone but only subscribed members can submit new messages. While "Puppet Users" and "Puppet Enterprise" are open to the public, the "Puppet Developer"-mailing list users must be approved before begin able to even read the content. Due to this, there are no members or topics accounted for as of the time of writing.

The IRC-channel have about 700 users during any time of the day. The IRC-channel are used to discuss both problems and ideas for upcoming versions. The developers are in the channel and participate in discussions.

5.3.2 Chef community

There is a part of the *Chef* homepage which is dedicated to the community. There are several mailing lists that users can subscribe to, one for users, one for developers and another for announcements by the developers. The user mailing list has 815 subscribers while the developer mailing list has 335. It is not possible to see the number of topics in the mailing list.

The other two ways of contacting the community through official channels are two other links on the homepage. One of the links go to their IRC-channel and the other link goes to a chat available directly on the homepage. The IRC-channel have around

300 users during any time of the day and there are developers in the channel participating in conversations.

Other than that, the community share *cookbooks* with each other by using the community home page as a sharing hub.

5.3.3 CFEngine community

In order to come into contact with the community through the official channel, there is a message board on the home page where users post questions and ideas. Everything posted on the message board are also delivered to a mailing list which can be subscribed to. While there is an IRC-channel that can be joined, it is not visibly linked on the home page. The IRC-channel contain around 100 users at any given time.

As of the time of writing this thesis, there are no other visible channels on the home page. By looking at the forum it was possible to find a mailing list on Google groups that has 1458 subscribed members with 3644 topics. The first post that is visible on the mailing list dates back to 1996. The current activity is low with only a few messages each month over the last 6 years. Between 2003 and 2006 there were however an average of 179 posts per month, so the the activity has declined.

5.4 SCMS languages

In this section, the results corresponding to the attribute *Languages used* are presented.

The investigation of languages were done both through a literature analysis of the SCMS documentation and through an implementation where languages were tested. The literature analysis examined the official documentations description of the language and how well it was documented. The test controls both correctness in the documentations language description and effectiveness of the language when it comes to resource creation.

Puppet uses a specially designed declarative language that share some similarity with vanilla `Ruby` but has been changed in order to serve the purpose of modelling resources better. As it is a custom language, not all parts of a true programming language exists. Functionality to the language have been added with almost every major content patch for *Puppet* as can be seen by table 2. A comparison to the other SCMS languages will be done in the analysis section.

To declare a service, a file containing information about the service is created. In the creation of a service it is possible to define certain characteristics such as name, if the service should be enabled, and if there should be some special requirements for the service.

```
1 service {"bind9":
2     name      => "bind9",
3     enable    => true,
4     ensure    => running,
5     hasstatus => true,
6     require   => Package["bind9"],
7   }
```

Figure 6: Puppet service definition

Feature	*0.24.x*	*0.25.x*	*2.6.x*	*2.7.x*	*Upcoming*
Plusignment operator (+>)	X	X	X	X	X
Multiple Resource relationships	X	X	X	X	X
Class Inheritance Overrides	X	X	X	X	X
Appending to Variables (+=)	X	X	X	X	X
Class names starting with 0-9	X	X	X	X	X
Multi-line C-style comments	X	X	X	X	X
Node regular expressions		X	X	X	X
Expressions in Variables		X	X	X	X
RegExes in conditionals		X	X	X	X
Elsif in conditionals			X	X	X
Chaining Resources			X	X	X
Hashes			X	X	X
Parameterised Class			X	X	X
Run Stages			X	X	X
The “in” syntax			X	X	X
The “unless” syntax					X

Table 2: Functionality in Puppet declarative language

```
1  service "bind9" do
2    action [ :enable, :restart ]
3    supports :status => true, :start => true, :stop => true, :restart =>
       true
4  end
```

Figure 7: Chef service definition

In figure 6, a service for a domain name system running Bind 9 is created. By using the declarative language of *Puppet* the service is configured to be running and that it requires the package `bind9`. The statement `require` means that the package called `bind9` is required to be installed.

The language used in *Chef* is Ruby. This means that all the power that are in the Ruby language is available to be used when writing recipes. The syntax when writing *Chef* resources follow the syntax for Ruby.

An example of a similar service declaration in *Chef* as the one for *Puppet* mentioned in the previous section can be seen in figure 7.

While *CFEngine* does not have an equivalent way of declaring a service, the code in figure 8 shows how to start a sequence for a service. This is the starting point in *CFEngine* for controlling services, just as the examples from *Puppet* and *Chef* above.

CFEngine uses a specialized declarative language. It is quite extensive and can do basically anything that modern scripting languages can do.

5.5 Documentation

This section presents the results from the implementation corresponding to the attribute *The official documentation*.

```
1 body common control
2
3 {
4 any::
5
6   bundlesequence   => {
7                        bind9
8                        };
9 }
```

Figure 8: CFEngine service starting point

The documentation was examined with the help of a literature analysis where documentation was thoroughly investigated to see if relevant information was presented. The information presented should contain enough information administrators to get help with both practical examples and theoretical explanations. The documentation should also be correct and in order to establish correctness of the documentation, the information was also tested with the help of the implementation.

Puppet have a set of documents targeting new users which contain basic information that new users might find helpful. The information describe the *Resource Abstraction Layer (RAL)* and how the RAL is used to compare the current state against the desired state of a service and then makes the necessary changes. This is one of the core principles of *Puppet* and by explaining it early to new users it will give a good entry point to understanding how the declarative language works. The way that the introductory documentation is structured follows a logical structure. There are clearly defined steps with thorough explanations of what is happening in each step. In this introduction there is not an extensive explanation of the language used in *Puppet*, but it does explain some fundamental parts of the language such as variables and conditional statements. To help give an overview of what the language supports, refer to table 2 where feature support of each version of *Puppet* is described.

The documentation for *Chef* have a section tailored for new users. This section is called *Chef Basics* and explains the parts that make up the SCMS with servers and agents and how they correlate. An important part of *Chef* is the command-line tool `knife`, which is mentioned in the *Chef Basics* section. However, while it is explained, there are no hands-on examples of how to use the tool under this section. In order to find out how the tool works, the reader must go to another section thus leaving the *Chef Basics* section. This is a frequent occurrence in the *Chef* documentation as articles often include links to other parts of the documentation which directs the reader away to a new article. In the *Chef* introductory documentation for new users there is one article about the language. As *Chef* uses `Ruby` it does not explain what the language can or can not do, as that is limited by what `Ruby` offers. To introduce how resources are written in *Chef*, there are hands-on examples where readers can learn the syntax.

The documentation of *CFEngine* is largely theoretical. There are many theoretical articles that describes the engine of the SCMS that will give the reader a good understanding of *CFEngine* works. Mostly, it is built upon the same concept as the other two SCMS with an introductory section where new users can get the know the SCMS. The articles in *CFEngine* often follow a single topic and then recommend what the users should read next. As *CFEngine* uses a custom language there is an extensive documentation of both syntax and usage. As for the hands-on experience, there are tutorials that

readers can follow and replicate in order to get a feel of how to write and use resources.

6 Analysis

This section will contain the analysis of both the results from the implementation and the gathered data from message boards, articles and other sources.

6.1 Dependencies

While it is possible to argue that dependencies play a small role regarding the choice of an SCMS it it is important to remember that each software installed on a system can potentially be exploited. While the security implications of dependencies are not investigated in depth in this thesis, it is still a factor that is notable. Furthermore, when using an agent in order to configure servers it means more preparation before the real configuration take place. As can be seen by table 3, there is one SCMS that require less dependencies than the others.

	Agent	Server
Chef	67	265
CFEngine	3	3
Puppet	11	183

Table 3: Amount of dependencies required of the SCMS

One of the reasons for the low dependency requirement for *CFEngine* is that it comes in a pre-compiled binary. The official documentation from *CFEngine* states that there are three packages that administrators should ensure that they exist on the system before installing the pre-compiled binary.

During the installation of the SCMS there were no complications. There are many dependencies for the server for some of the SCMS but as the server software often is on a dedicated machine there should rarely be a dependency conflict. In order to install the *Puppet* and *Chef* agents, Ruby must be installed. This is not necessarily an issue as it is common to have Ruby installed, but it is a fairly large framework that needs to be installed. Each dependency that is installed is a process that must be considered by administrators.

As the amount of dependencies increases, so does the necessity of researching possible software conflicts. This results in extra workload for administrators that has to ensure that an installation of an SCMS does not negatively affect the environment.

6.2 Community

By looking at the numbers, Puppet have the most active community. They have, by a clear majority, the most users on their IRC-channel. The amount of users during April 26 can be seen in table 4. One might argue that it is not the number of users that is important, but the activity. However, the more users that are in an IRC-channel the more likely it is that someone knows the answer to questions that arise. Furthermore, by being in the channel it is likely that they are willing to contribute since they have actively sought out the channel. Considering this, it is an advantage to have many users in the channel. The release date for the SCMS differs which means that the communities have grown over different periods of time. This means that it is not possible to say for

	CFEngine	Puppet	Chef
Users	71	728	335

Table 4: Number of users in IRC-channel on April 26, 2012

certain that one community has an advantage over another since it takes time to establish foundations. The activity within communities can however give an indication of the level of support that the community may be able to give.

The channel activity seem to follow timezones as well. From looking at the activity in the channels, *Chef* are more active during U.S daylight hours, while *Puppet* are more flexible. Unfortunately, during the period of monitoring the IRC-channels it has not been possible to deduce if timezones affect the *CFEngine* channel as the activity in general is low.

Chef is keeping log files from the IRC-channel which is accessible on the official home page, this is a nice touch as users can then access the backlog and read conversations that might have been missed, or are otherwise interesting. This is something that neither *Puppet* nor *CFEngine* offer which is something that should be considered.

6.3 Language used

To measure the effectiveness of the languages, the number of lines required to perform common actions in regards to specifying functionality of services are measured. No CPU or memory related efficiency have been measured. Furthermore, the syntax have been examined to see if common functionality such as iterations and selections are available to be used when creating resources in the different languages.

As mentioned in the previous section, *Puppet* uses its own declarative language in order to write its resources. In later version of *Puppet* this has been extended so that normal `Ruby` can be used. By doing this, not only can administrators use the specialized language designed to control the *Puppet*-environment, but they can harness the power of `Ruby` as well. The limitations of the specialized language can be overcome by complementing with `Ruby`.

With the use of the declarative languages in the SCMS the administrations can focus on what should be performed instead of how it should be performed. While certain conditional statements might need to be created, the SCMS take care of checking that requirements are met. In figure 7, a service is defined and controlled with the help of *Puppet*. This is a good example of how the declarative language is used. The language focuses on what to achieve and not how. With the help of certain keywords it is possible to set certain conditions for how the service should act. As can be seen by figure 7, the service is defined as running with the statement; `ensure => running`. The keyword `ensure` will, just as it sound, ensure that a certain condition is met. In the aforementioned example it will ensure that the service is running. For *Chef*, the equivalent of "`ensure => running`" is: "`action [:enable]`". How to use the *Chef* statement can be seen in figure 8. To ensure a running service in *CFEngine* the code in figure 9 can be used.

It might seem restrictive to create a specialized language for creating resources and managing the SCMS. However, considering that the goal is to manage services and as long as the language can do as much it will suffice. The power of Ruby can be

```
bundle agent restart_process
{
processes:

  "bind9"

      comment => "Make sure that the bind service is running",
      restart_class => "restart_bind9";

commands:

 restart_bind9::

     "/etc/init.d/bind9 restart";

}
```

Figure 9: Ensure running process in CFEngine

harnessed into creating advanced and sophisticated resources but it can also become overly complicated when performing smaller tasks. In the case of *Puppet* it is possible to extend the environment by using Ruby in conjuction of the *Puppet DSL*. *CFEngine* have, as mentioned in section 4.3, an extensive custom language that is a powerful tool in its environment.

6.4 Documentation

The documentation differs between the SCMS. *Puppet* have a good structure with both hands-on examples nested with theoretical explanations of each step performed. The reader can execute each step in his or her own environment and then continue to read about why the step was performed. The *Chef* documentation has another structure that is less linear than the documentation of *Puppet*. While reading the *Chef* documentation, the reader can easily get sidetracked due to the sheer number of links in the documentation. While the links give the reader more information about components of the *Chef* SCMS, *Puppet* often gives the same type of information without asking the reader to open a new webpage.

CFEngine is different from both *Puppet* and *Chef* as the *CFEngine* documentation mostly contain theoretical information. By having the theoretical information as the starting point for readers it enables the readers to get a good overview of why some of the functions in *CFEngine* are the way they are. However, it is not only theoretical information in the *CFEngine* documentation as the theoretical information are complemented with hands-on examples where needed.

Overall, the documentations of *Puppet* and *CFEngine* contains a nice structure with both theory and hands-on examples. *Chef* does contain the same relevant information as *Puppet* and *CFEngine* for its software, but it is redirect readers to other parts of the documentation in many of its articles which can be distracting.

6.5 SCMS and ISO/IEC 90003

As have been mentioned earlier, the scope of the ISO/IEC 90003 involves the management of configurations. It is important to remember that the standard is created for software engineering, and can not be directly applied to software administration. However, the theory behind configuration management is similar. It is important to have proper configuration management as configurations can have a huge impact on the computer network. As SCMS aim to be a solution for configuration management and automation, it is not unrealistic to expect some general appliance to an existing ISO/IEC standard wherever applicable. While some of the ISO/IEC standard is hard to implement in a software, such as process planning and responsibilities, other parts can be implemented and used in SCMS.

The ISO/IEC 90003 mention release management and delivery and all of the SCMS, by their nature, give a good support for release management and delivery. They all give a good control of the environment and enables administrators to easily control when and how to deploy their resources. *CFEngine* focuses on having clients pull new resources from the *policy servers* while both *Chef* and *Puppet* allows for both push and pull functions. Before deploying any configurations the SCMS also contain functions for checking the resources for errors. These errors are however syntax errors within the SCMS resources and not actually for the configuration files. This means that, while syntax errors are possible to detect within the SCMS, the SCMS are not able to check the final resource that are deployed to the servers on the networks. As the ISO/IEC 90003 states that configuration evaluation should be included in configuration management, the SCMS have not implemented any way in the software to evaluate the configurations. Even if this is a hard task to achieve considering the immense amount of possible configuration types, it could be possible to let administrators specify a number of allowed states and have the SCMS compare the state that resources result in against a number of allowed state for the service.

One of the more important aspects of the administrative part of configuration management is to keep version control. There are no inherent way in any of the SCMS to have a resource- or version history and to get this functionality, administrators will have to use existing platforms such as *github* or *SVN*. All three of the SCMS do recommend in their documentation that a repository that keep track of versions should be used, and refer to the previously mentioned *github* or *SVN*. By not including such a tool within the SCMS it will add another dependency of the SCMS as revision control is heavily recommended. The ISO/IEC 90003 also mentions *configuration identification*, which is a bit outside of the scope of *github* or *SVN*. However, the theory behind identifying versions and when they are brought under control by the SCMS is something that should be kept in mind. By keeping track of information of where configurations was brought under control by an SCMS it can be used as information about the effectiveness of using an SCMS.

While the SCMS discussed in this thesis do not fully follow the ISO/IEC 90003 they do follow the concept. There are existing functionalities that allow for simultaneous control, release management and revision control.

6.6 Related work

During the work for this thesis, no other works has been identified that discuss generality or maturity in SCMS. However, there are works about how system administration

should be performed. The work done in this thesis can best be set in relation to previously identified work that has been described in the background section.

In the work by Burgess (2003) a mathematical approach to system administration is taken. In his work, Burgess aims to mathematically show important aspects of system administration. Burgess say that trying to define system administration is a slippery business as there are many parts of system administration to take into consideration. Burgess define what system administration is about by removing subjectivity and mathematically show the true aspects of system administration. The work of Burgess (2003) is interesting as it is reflected in the SCMS examined in this thesis.

Estublier et al. (2005) describes the impact that research in software engineering have had on SCM. Estublier et al. (2005) mentions how the aerospace industry experienced issues caused by inadequately documented engineering changes. After several decades, software began having the same difficulties in term of managing change (Estublier et al., 2005). The same techniques used for solving the difficulties in the aerospace industry was used for solving issues software management had (Estublier et al., 2005). The research made for software engineering has thus been transferred into the realm of software configuration management. The theory about software engineering can be seen in the ISO/IEC 90003 that is used in the the work for this thesis to examine if the SCMS have implemented the theory of configuration management into the software.

7 Conclusion

As the SCMS examined in this thesis are similar to each other it is possible to say that it indicates a general and mature way of deploying configurations that can be used by many. The implementation of the SCMS shows that the documentation is mostly accurate for the three SCMS. All of the SCMS have been successful in aiding administrators during the implementation phase. The installation and implementation does not contain any particular difficult steps as the steps are covered by the documentation. There are some pre-requisites that administrators need to take into consideration when installing *Chef* and *Puppet* such as the dependency packages for Ruby. Some dependencies can deter administrators from installing if special requirements exists in the computer network or on the servers. As such requirements are unique for each environment it is impossible to say which dependencies that may cause an issue.

The declarative languages of *Puppet* and *CFEngine* are very powerful and much can be done with little effort. There are similarities in how *Chef* makes use of Ruby but tasks can become unnecessarily complicated when using *Chef* as it may require extra code.

There are good support structures that try to help administrators regardless of previous skill level. The experiment indicate that the SCMS do allow for good control of deploying configurations. It is possible to see that the theory behind configuration management have been considered when developing the softwares. The SCMS are similar when it comes to functionality and the major difference is the implementation methods, such as opportunistic and server-client environments. The biggest advantage of using SCMS is that it allows for an environment with a uniform deployment of configurations regardless of operating systems used in the computer network as the SCMS are platform independent. Even the smaller computer networks gain from having this uniform environment and should consider an implementation as good help can be found both through official and community channels. There are good support for each of the SCMS in regards of implementation that will aid administrators when taking the first steps to an environment where the services are controlled by an SCMS.

The work in this thesis can however give an indication how different developers have chosen to integrate functionality and support structure for the implementation phase. It is possible to see that there are similarities, such as support for version control, and that the overall support are good. All of the SCMS have good support structure from both official documentation and community alike.

8 Discussion

Finding the right attributes to examine is hard and there might be other attributes that would even more accurately reflect how general and mature the SCMS are. The attributes used in this thesis was chosen after reading studies about the theory behind configuration management such as (Estublier et al., 2005) together with reports of the difficulties of implementing SCMS such as (Rosin, 2011). Due to reading non-scientific material it may have skewed the view of how attributes are valued and there can therefore be other attributes that are more valuable in an implementation like the one in this thesis.

By performing an implementation in conjunction with studying different sources of information, such as the ISO/IEC 90003 standard and the official documentation, it is possible to evaluate how the concepts have been realized in the software and if information reported match actual situations. One issue that can arise when performing an implementation such as the one in this thesis is that knowledge is acquired during one step of the implementation that affect the following steps. In order to avoid such a bias, the entire implementation process have been iterative where steps and SCMS have been re-examined in order to revalue previously investigated attributes. The implementation take the study one step further and not only compare the theoretical findings but also gather own data, that can be used to compare with other sources to see if the data matches.

Aspects such as satisfaction factors as is mentioned by Shaw et al. (2002) can however be done without an implementation. The implementation adds some value to such an evaluation as it can be tested how accurate the documentation is. In the same manner, the implementation can try out solutions given by the community as a response to a question. However, it could be valuable to follow how support is given on mailing lists or message boards and see how many questions that actually are answered successfully. Such a follow-up on questions can be done without an implementation as support.

The difficult part of determining what characterizes a good documentation is that different people have different opinions. One persons opinion might be very different from another persons opinion. This can render parts of the work in this thesis moot as opinions may have affected the analysis. To try to counter this, the implementation was used to test as much of the information as possible. However, the actual content the developers have chosen to include in the documentation can still be discussed as opinions may differ about the parts included or left out.

Regarding the relation to the ISO/IEC 90003 standard it is up for discussion if the standard actually can be applied to this area of configuration management. The standard is created for another area and it might not be applicable to administration of software configurations. However, from my experience the standard does bring up very relevant points that are important when administering configurations, such as version control. Administrators need to have a good structure for the configurations used. It is simply not viable to handle many configurations in an ad-hoc way, and the ISO/IEC 90003 brings up relevant information to solve such problems. Still, the standard is created for another area and might not be applicable to administration even though the theory is sound.

8.1 Ethical and social aspects

When implementing SCMS in a computer network it is often to make administration more efficient. One consequence of this is that administrators may get less work, thus reducing the number of administrators needed to manage the computer network. This means that when administrators implements an SCMS it is possible that the administrators are working towards making themselves unemployed. Such a scenario may not be very likely as administrators have more tasks at hand than to just manage services, but it may be realistic in smaller organisations that barely have enough workload to justify in-house administrators. Crassly said, administrators are asked to make themselves useless when implementing automation software like the SCMS examined in this thesis. A comparison can be made to the industrialization in the 19th century where machines took over many of the jobs that previously was done by manual labour. It is not an especially accurate comparison as the industrialization is a much bigger step in automation, but it is a step in the same direction. The SCMS automates part of the work for administrators and will help ease the workload.

When automating parts of configuration management it is possible to reduce the workload of administrators and create a better working environment. The SCMS may help reduce the number or errors introduced with manual labour, and automate parts of configuration management that leaves administrators free to do other work. Administrators can with the help of SCMS focus on more important tasks than correcting mistakes introduced by human errors. The SCMS also provides support for higher level concepts such as release management, which also aids administrators in their work as streamlines configuration delivery. Overall, this may result in an increased effectiveness that is beneficial for organisations.

9 Future work

There are some aspects that are worth investigating further. This thesis does not contain the security aspects of the SCMS, which is an important aspect. The different SCMS have different security implementations that each have their own advantages and disadvantages. The SCMS is a vital point in the computer network as it contains information about configurations for other parts of the network and it is important that it is protected.

Another aspect that could be looked into is the actual performance of the SCMS. Several parameters can be looked at such as; Bandwidth usage, CPU usage and resource compilation speed.

There is a fundamental difference in how *Chef* together with *Puppet* works, compared to *CFEngine*. While *Chef* and *Puppet* often are deployed in the server-client mode, *CFEngine* is an opportunistic software that can adapt to network failures. As described in section 2.4.3, being opportunistic means that the software adapts to network failure and still allow *CFEngine* to function. The question is if this way of operating is more effective than having a redundant server in the case of *Puppet* and *Chef*?

In section 2.3 scripting is related to software configuration systems. A more in depth research of the differences between these two way of administering services should be done. While SCMS require additional software on the servers, scripting can often be used immediately. Which advantages and disadvantages do SCMS bring to computer networks?

References

Berndtsson, M., Hansson, J., Olsson, B., & Lundell, B. (2007). *Thesis projects: A guide for students in computer science and information systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Burgess, M. (2003). On the theory of system administration. *Science of Computer Programming*, *49*(1–3), 1 - 46.

Burgess, M. (2011, March). Testable system administration. *Commun. ACM*, *54*, 44–49.

CFEngine. (2012a). *Quick Start Guide.* Author. Retrieved from https://cfengine.com/manuals/cf3-quickstart#Hubs

CFEngine. (2012b). *Quick Start Guide.* Author. Retrieved from https://cfengine.com/manuals/cf3-quickstart#CFEngine-Components-_002d_002d-What-Are-they-and-How-Do-They-Work_003f

Couch, A. L., & Daniels, N. (2001). *The maelstrom: Network service debugging via "ineffective procedures".*

Dolstra, E., Bravenboer, M., & Visser, E. (2005). Service configuration management. In *Proceedings of the 12th international workshop on software configuration management* (pp. 83–98). New York, NY, USA: ACM.

Estublier, J., Leblang, D., Hoek, A. v. d., Conradi, R., Clemm, G., Tichy, W., & Wiborg-Weber, D. (2005, October). Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, *14*, 383–430.

Heldal, I., Spante, M., & Connell, M. (2006). Are two heads better than one?: object-focused work in physical and in virtual environments. In *Proceedings of the acm symposium on virtual reality software and technology* (pp. 287–296). New York, NY, USA: ACM.

International Standards Office. (2004). *ISO/IEC 90003 Software engineering – Guidelines for the application of ISO 9001:2000 to computer software.* Geneva:ISO.

Nemeth, E., Hein, T., Snyder, G., & Whaley, B. (2010). *Unix and linux system administration handbook*. Prentice Hall.

Puppet Labs. (2011). *Docs: Learning — Basic Agent/Master Puppet.* Author. Retrieved from http://docs.puppetlabs.com/learning/agent_master_basic.html#what-do-agents-do-and-what-do-masters-do

Puppet Labs. (2012a). *Docs: Learning — Manifests.* Author. Retrieved from http://docs.puppetlabs.com/learning/manifests.html

Puppet Labs. (2012b). *Puppet Labs.* Author. Retrieved from http://puppetlabs.com/

Rosin, L. (2011, March). *Open source systems management: Hits and misses.* TechTarget SearchEnterpriseLinux. Retrieved from http://searchenterpriselinux.techtarget.com/feature/Open-source-systems-management-Hits-and-misses

Shaw, N. C., DeLone, W. H., & Niederman, F. (2002, June). Sources of dissatisfaction in end-user support: an empirical study. *SIGMIS Database*, *33*(2), 41–56.

Steinglass, B., & Thomas, T. (2011). *Core Components.* Opscode. Retrieved from http://wiki.opscode.com/display/chef/The+Different+Flavors+of+Chef

Thomas, T., & OMeara, S. (2012). *Core Components.* Opscode. Retrieved from http://wiki.opscode.com/display/chef/Core+Components

Turnbull, J. and McCune, J. (2011). *Pro Puppet*. Apress.