Technische Universität Berlin, Germany

Master's Thesis

# Evaluating Methods to Maintain System Stability and Security When Reversing Changes Made by Configuration and System Management Tools in UNIX Environments

May 2015

Markus Holtermann

# Acknowledgment

I want to express my gratitude to Prof. Dr. habil. Odej Kao from the faculty of Electrical Engineering and Computer Science at Technical University of Berlin, Germany, for encouraging me to pursue the topic of this thesis.

Many thanks go to all my colleagues at Flying Circus Internet Operations GmbH, especially Christian Kauhaus and Christian Theune who gave me the opportunity to work on my ideas in the field.

I thank Tim Graham for being a lector of my master's thesis, giving it a linguistic final touch.

I am forever grateful to my parents who supported me during my education in all my decisions and encouraged me to follow my dreams.

Without the Ubuntu Germany and international Django communities I would not have gained many skills that influenced this thesis.

# Contents

# Abstract

Having access to the Internet every second of our life is nowadays often taken for granted. The work and effort system administrators put into the infrastructure to keep services up and running smoothly is immeasurable. Fortunately there are tools like *Chef* and *Puppet* that allow them to simplify their work and reduce mistakes by automating repetitive tasks. Sometimes these automations do not work as expected and can result in critical situations. Undoing changes can result in data loss or inconsistencies, thus security and stability are a major concern.

This master's thesis explores common use cases of configuration and system management tools, and sheds some light on the risks faced by a system administrator when preparing rollback operations.

For this purpose the thesis categorizes various IT resources that are important in today's data centers. Finally, the thesis reveals risk reduction rules for rolling back operations to those resources.

# 1 Introduction

Configuration and system management is a part of systems engineering that allows system administrators to track changes to software and hardware configuration. Furthermore, it provides administrators with the possibility to conveniently and reliably apply the same changes to an infinite number of systems by simply defining what a particular configuration file on the target systems should include or be like, or by specifying what programs should be installed on the target system.

This master's thesis will examine the current status of configuration management for the reversibility of changes. This means that changes made in the past should be able to be rolled back. A rollback can involve various tasks – e.g. restoring a file to a previous state or revoking someone's access to a system – which are explained in depth in chapter 5 – *Resource Classification and Utilization*.

Keeping IT environments up to date and organized has been mentioned about 20 years ago, as presented in chapter 3.2 – *Historical Connections*, but has not received any major follow ups. Situations like "*Knightmare: A DevOps Cautionary Tale*" [Sev14] and "*Flying Circus RCA report #13266*" [The14b], on the other hand, reveal the need for configuration management tools to provide support for rollback strategies.

To explain the problem in a more practical way, imagine the following example: A hosting provider offers virtual machines that have an IP address that is assigned from a pool of free addresses when the machine is first set up. At some point a customer requests the server to be shut down and removed. Once the server is removed, its IP address is freed and goes back into the pool. However, a customer is able to request a restore of a server from a backup within a given time frame after its removal. This works perfectly fine as long as the original IP address is not in use. However, configuring the network setup will fail if the IP address has been recycled to a different machine.

This is a fairly simple example. It outlines the constraints (IP address, backup storage space) a resource (the virtual machine) has and demonstrates that a major problem is not yet solved. Changing a small part in a large system with many different kinds of resources involved may result in unexpected behavior if constraints are not taken into account.

Chapter 2 – *Basics* provides some prerequisite knowledge for understanding the main content of this master's thesis. The next chapter, 3 – *Related Work*, outlines others' work and its relationship to this thesis. The background presented in chapter 4 – *Background* provides the reader with essential knowledge that is required to understand the remaining chapters of the thesis. Chapter 5 – *Resource Classification and Utilization* develops an IT resource classification from which rules are derived that a system administrator should follow *to maintain system stability and security.* Furthermore, the chapter presents use cases that are discussed further in chapter 6 – *Maintaining System Stability and Security* where possible solutions are presented. The evaluation in chapter 7 – *Evaluation* gives two case studies where the rules and solutions are applied. The last chapter, 8 – *Conclusion and Prospects*, summarizes the research from the previous chapters and provides suggestions on how configuration management tools can improve their support of rollback operations.

# 2 Basics

This chapter provides basic information that a reader should know about in order to ease understanding of this master's thesis. System administrators are likely to already know the content. People who are new to the fields of configuration or system management will get the general knowledge they need to understand the content of this master's thesis.

## 2.1 What Is Configuration and System Management?

Configuration and system management is a part of systems engineering that allows system administrators to track changes to software and hardware configuration. Furthermore, it provides administrators with the possibility to conveniently and reliably apply the same changes to an infinite number of systems by simply defining what a particular configuration file on the target systems should include or be like or by specifying what programs should be installed on the target system.

Configuration and system management software can be used to keep these servers similarly configured. It can ensure *convergence* and *congruence* of the systems it is used on, which is explained in depth in chapter 3.1 – *Thematic Classification*.

## 2.2 What Is a Configuration Management Database?

A *configuration management database* (CMDB) provides a configuration and system management software with additional information that the software will include when applying changes to a system. This could, for example, be the list of customers, the virtual machines that belong to a customer and the IP addresses assigned to those virtual machines.

A configuration and system management software will use the information from the database to automatically generate the changes that need to be performed on each system.

As will be explained in a later chapter of this thesis, a CMDB can also be used to keep track of the state of each resource, e.g. mark availability of an IP address or the turn-down time of a virtual machine.

## 2.3 What Is ACID Behavior?

Databases are an important place to store information of whatever structure. A set of modifications that logically belong together can be grouped in a “transaction”. In order to continuously and concurrently provide “correct” data, in 1981, Jim Gray developed and defined some properties a system has to obey [Gra81]. Two years later, in 1983, Andreas Reuter and Theo Härder added the term *Isolation* to the already defined *Atomicity*, *Consistency* and *Durability*. This essentially marks the beginning of the term ACID.

The definitions of these four terms by their authors are:

**Atomicity** “*It either happens or it does not*” [Gra81].

**Consistency** “*The transaction must obey legal protocols*” [Gra81].

**Isolation** “*Events within a transaction must be hidden from other transactions running concurrently*” [HR83].

**Durability** “*once a transaction is committed, it cannot be abrogated*” [Gra81].

## 2.4 What Is a Package Manager and What Are Package Dependencies?

UNIX systems, and Linux systems in particular, have so-called package managers that are used to safely install programs and libraries on a system. There are a variety of package managers out there, differentiated by how they store meta data internally, among other aspects. Meta data for a package can be, but is not limited to, the package name, its version, licensing information, authors, project website and, especially important for this master’s thesis, dependencies on other packages. That last bit of meta data allows a package manager to know that to install package `A`, package `B` needs to be installed first.

The list of available packages is stored in repositories the package manager accesses. System package managers, such as `APT`[1], `Pacman`[2] and `Yum`[3] have a local copy of the software collection, allowing for faster queries. On the other hand, `npm`[4] and `pip`[5], the package managers for Node.js and Python packages, need to access the online software repositories each time they need to retrieve information.

Package managers keep track of which packages are installed. That way, the installation of `A` will only install `A` if `B` is already installed, and there is no need to install `B` again. By leveraging the meta data of the packages currently being installed and the meta data from the software repositories, package managers can also manage updates for packages. The details on how this is achieved are explained in chapter 6.1 – *Installation, Update and Removal of Programs – Dependency Problems.*

## 2.5 What Is a Virtual Machine and What Is a Linux Container?

A Virtual machine is a computer whose hardware is simulated by a software (hypervisor). The hypervisor provides a mapper around the underlying physical hardware (host) and allows to assign (parts of) the hardware capabilities to a virtual machine. That way a Linux host can be used to run multiple, possibly independent, virtual computers.

Virtual machines are often independent of each other, thus breaking one virtual machine does not imply that other machines on the same host are broken, too.

Linux containers such as LXC or `systemd-nspawn`, reuse parts of the host's operating system. By reducing the amount of virtualization, these approaches allow the virtualized guests to run faster.

[1]`https://wiki.debian.org/Apt`
[2]`https://www.archlinux.org/pacman/`
[3]`http://yum.baseurl.org/`
[4]`https://www.npmjs.com/`
[5]`https://pip.pypa.io/`

## 2.6 What Is a File System?

File systems are able to hold information about files and directories. This includes information such as where a file or directory is stored on a hard disk, what size it has, who the owner is and when the last access or modification happened to that file / directory.

In distributed environments file systems can be shared across servers by using file systems that provide network access, such as NFS.

# 3 Related Work

This chapter integrates the thesis into the thematic and historical context of other research in configuration and system management.

## 3.1 Thematic Classification

As with everything in computer science, configuration management evolved over the years. As a result, there are four major levels of configuration and system management:

1. No systematic configuration management
2. Divergent configuration management
3. Convergent configuration management
4. Congruent configuration management

Furthermore, one needs to differentiate between manual and automated configuration management. This refers to the way in which the configuration is applied on target systems.

Manual configuration management implies that an administrator is issuing commands by hand. This is error prone because it involves the human error factor. Automated configuration management, on the other hand, works autonomously by having programs doing the recurring tasks. Once set up, the management tools usually run recurrently. An administrator might start a configuration "run" manually, but the actual configuration of the target systems is done automatically.

Combining both categorizations, the following matrix arises. It shows whether or not the combinations of these orthogonal classifications make sense. For example, there is no way of having automated configuration management when there is no systematic configuration management:

| | No systematic | Divergent | Convergent | Congruent |
|---|---|---|---|---|
| Manual | Yes | Yes | Yes | No |
| Automatic | No | Yes | Yes | Yes |

Table 1: Matrix for configuration management classification and manual and automatic configuration management

### 3.1.1 No Systematic Configuration Management

In the very early stages of IT infrastructure, there was no systematic configuration management as we know it from today's data centers. Servers were like "*Snowflakes*" [Fow12], unique, non-reproducible and very hard to maintain. Administrators connected to each server individually and applied the changes that needed to be made. For small environments this is still a common way to maintain a server. The incentive for these manual changes arose from the short-term need. The time an administrator needed to become acquainted with a certain configuration tool was too long.

### 3.1.2 Divergent Configuration Management

To partly automate the deployment of projects and simplify their work, administrators often wrote their own scripts and tools. However, those scripts eventually broke and stopped working for a variety of reasons. The scripts left systems in an undefined state, possibly making another script fail and so on and so forth.

In their paper "*Why Order Matters: Turing Equivalence in Automated Systems Administration*" [TB02, Ch 4.1] Stephen Traugott and Lance Brown describe this behavior as "*divergence*".

> *Divergence is characterized by the configuration of live hosts drifting away from any desired or assumed baseline disk content.* [TB02, Ch 4.1]

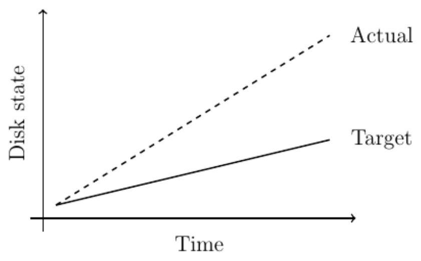

Figure 1: Divergence [TB02, fig. 4.1.1]

This practice is still common and allows arbitrary users to conveniently install software from the Internet. Examples of this are many installation files for Microsoft Windows programs, the installation of the "Heroku Toolbelt"[6] or the `ZSH` configuration "Oh My ZSH"[7]. The installation instructions essentially ask a user to download a file and execute it.

Depending on the amount of human interaction and the granularity of the single tasks, the configuration management can be seen as either manual or automatic, there is no real distinction. For manual management with a lot of human involvement, one could even classify this "configuration management" as not being configuration management at all.

### 3.1.3 Convergent Configuration Management

A much more deterministic way to maintain a system follows the "convergence" definition [TB02, Ch 4.1] by Traugott and Brown. They describe it as having an actual state and a target state. By reducing the difference between them, the actual state becomes more and more like the target state.

[6] `https://toolbelt.heroku.com/`
[7] `https://github.com/robbyrussell/oh-my-zsh#basic-installation`

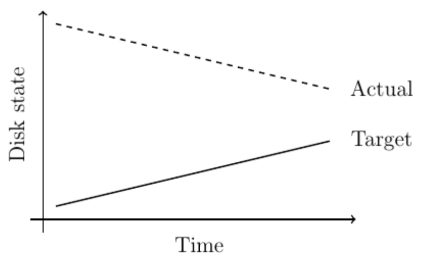

Figure 2: Convergence [TB02, fig. 4.2.1]

By synchronizing configuration files across multiple divergent hosts, administrators make the hosts converge for those files. A simple solution to do this are checklists. To reduce the human error factor, these checklists can also be automated.

### Manual Configuration Management

Checklists are used extensively to make the configuration and management of IT systems reproducible. The cloud hosting provider DigitalOcean, for example, provides a whole set of tutorials `https://www.digitalocean.com/community/tutorials` for various use cases. Guides like `http://wiki.nefarius.at/linux/der_perfekte_mail-server` allow beginners to create a safe and secure email server setup. Step-by-step instructions allow administrators to correctly perform complex tasks.

The "*BSI-100 Standards*"[8] by the German Federal Office for Information Security[9] are also checklists with "*recommendations ... on methods, processes, procedures, approaches and measures relating to information security.*".

In the end, companies that are using checklists are often also standardizing those processes to fit *Information Technology Infrastructure Library* (ITIL) requirements and allow ITIL certification. For the topic of this master's thesis, checklists are not relevant and will therefore not be taken into account.

[8] `https://www.bsi.bund.de/EN/Publications/BSIStandards/BSIStandards_node.html`
[9] Bundesamt für Sicherheit in der Informationstechnik (BSI)

### Automated Configuration Management

In 1998, Mark Burgess published the article *Computer Immunology* [Bur98]. In it, he proposes the idea of having tools in IT environments that behave like the human immune system and provide some kind of self-healing capabilities.

Burgess elaborates that the technique the human immune system uses to train itself to defend against threats is not possible in computer systems. The human body uses millions of cells that randomly check for threatening cells or molecules based on not only what the body has learned in the past but also on (random) mutations. Although training computer systems to recognize patterns is generally possible using e.g. neuronal nets, this would simply not be possible in the same way the human body does it:

> *Testing code at random places in random ways is hardly efficient, and while it might work with huge numbers ... in the body, it is not likely to be a useful idea ... Even the smallest functioning immune system ... consists of* $10^6$ *lymphocytes, which is several orders of magnitude greater than any computer system.* [Bur98, P. 291]

His idea already describes that there should be tools to automate configuration and system management. In fact, he developed “CFEngine” five years prior to this paper [Bur]. It provides automated configuration management but no “healing” features. However, the benefit of automation eliminates the human error factor, which eventually leads to more stable systems and predictable behavior.

The downside of automated configuration management, either manual or divergent, is the speed. While the scripts or installers for divergent systems simply perform one task after the other, tools for convergent systems need to gather the state of a target system has, before it can perform any actions.

An important behavior of convergent configuration management is the way in which parts of a system are excluded from being managed by the management tool. By not specifying anything about a resource, the management tool does not know the resource and thus cannot maintain it.

### 3.1.4 Congruent Configuration Management

The fourth level of configuration management is “*congruence*” [TB02, Ch 4.3]. Systems that essentially do not deviate more and more from the target system over time are in exactly the state a system administrator wants them to be.

This effectively gives an administrator full control over a system and full knowledge of the state of a system, which should be the goal of every system administrator according to Traugott and Brown:

> *We recognize that congruence may be the only acceptable technique for managing life-critical systems infrastructures, [...]* [TB02, Ch 4.3]

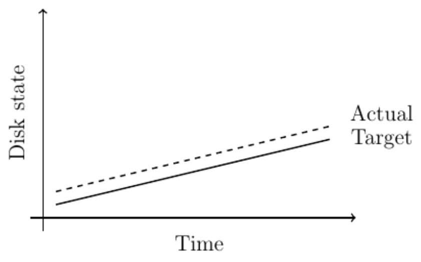

Figure 3: Congruence [TB02, fig. 4.3.1]

Unlike convergent configuration management, congruent systems have every resource defined. This makes it impossible to have changeable data. Today’s configuration and management systems work around this limitation by allowing integration of e.g. external storage systems that themselves are convergent.

The *Virtual Panel on Immutable Infrastructure* [PFBH14] by InfoQ outlines the opinions of the three experienced operations engineers Chad Fowler, CTO of Wunderlist, Mark Burgess, CTO and founder of CFEngine and Mitchell Hashimoto, CEO of HashiCorp, creator of Vagrant. The discussion deals with the question if immutable infrastructure “*is a step forward or backwards in effective infrastructure management*” [PFBH14].

While Burgess favors mutable infrastructure (hardly surprising due to his involvement in CFEngine) and describes the term “*immutable*” as a “*misnomer*”, Fowler defends immutable infrastructure, although he admits that his company 6Wunderkinder still relies on "*'cheats' with immutable infrastructure*". Hashimoto's view on immutable infrastructure straddles those of Burgess and Fowler. While Vagrant can be used for immutable infrastructure, it can also be used only for the initial provisioning of a system which then becomes a mutable system while being maintained with CFEngine.

Burgess argues that it is not possible to have an immutable server that gets its IP address via DHCP because this would be a change which is not possible by definition of immutability. Instead "*we should focus on ... what behaviours ... we want hosts to exhibit on a continuous basis. Or, in my language, what promises should the infrastructure be able to keep?*" [PFBH14, Question 1]

This thesis follows Burgess' opinion on mutable infrastructure. Immutable infrastructure, as explained later in this chapter, is not considered as part of the upcoming research. Immutable systems will not be considered because rolling back changes in those environments implies rebuilding the entire system. The newly build images will again be immutable.

## 3.2 Historical Connections

One of the first published papers on configuration management is from 1993 when Mark Burgess wrote about version 2 of his configuration management tool CFEngine [Bur93]: *CFEngine V2.0: A network configuration tool.* "*cfengine is intended first and foremost to be run as a batch job ...*" [Bur93, ch. 1]. Some of the tasks that can be done by CFEngine include the "*management of protection and ownership of files on any filesystem*" and "*autoconfiguration of the local area network device interface, netmask and broadcast addresses, as in ifconfig*" [Bur93, ch. 3]. CFEngine uses *a single file that describes the setup of all machines in a machine park* [Bur93, ch. 1]. Therefore, CFEngine is an automated and convergent configuration management tool.

In 1994, Paul Anderson published his paper "*Towards a High-Level Machine*

*Configuration System*" [And94] at LISA '94[10]. He describes the tool "*lcfg*" ("local configuration") that uses a central configuration database that has "*all information that is necessary to distinguish one machine from another*". The managed systems receive their configuration during initial setup and each time the system or one of its subsystems are restarted.

A few years later, in 1998, Steve Traugott and Joel Huddleston published their paper "*Bootstrapping an Infrastructure*" [TH98]. They describe a "*model [that] was developed during the course of four years of mission-critical rollouts and administration of global financial trading floors. The typical infrastructure size was 300-1000 machines, totaling about 15,000 hosts.*" [TH98, Abstract]. In 16 steps they explain how they "*were able to make a true migration from 'systems administrators' to 'infrastructure engineers'*" [TH98, step 2]. The important steps for the master's thesis that will be summarized here are "*Step 1: Version Control*" and "*Step 2: Gold Server*".

By using the version control tool *CVS*[11] Traugott and Huddleston found that they "*were able to do rollbacks or rebuilds of damaged servers and other components.*" [TH98, ch. 1]. This was a crucial discovery that allowed them to "*actually [be able to] destroy entire server farms and rebuild them with relative impunity during the course of development, moves, or disaster recovery. This also made it much easier to roll back from undesired changes*" [TH98, ch. 1].

Setting up a *Gold Server* – which is like a master server in today's terminology – they "*[made their] changes reproducible, recoverable, traceable*" [TH98, ch. 2]. By design, the gold server provides information to the clients but never pushes any information to them. Instead the clients pull updates. They decided to use this approach due to the odds that "*if you have more than 30 target hosts one of them will be down at any given time. Maintaining the list of commissioned machines becomes a nightmare*" [TH98, Push vs. Pull].

[10]8th USENIX Conference on System Administration
[11]Concurrent Versions System, `http://www.cyclic.com/cvs/info.html`

In the beginning of 2005, Puppet Labs, founded by Luke Kanies[12], started working on "Puppet"[13]. It is an agent-based configuration and system management tool that can achieve convergent system states. "Puppet" is one of the most commonly used configuration and system management tools in today's data centers according to Stephen O'Grady on RedMonk[14].

In January 2009, "Chef", founded by Jesse Robbins among others[15], announced [Rob09] their "systems integration framework" which is based on "Puppet". Due to its similarity to "Puppet" it holds many similar properties, such as being agent-based and providing convergent system states.

In addition to the aforementioned tools, "Ansible" and "Salt" are well known nowadays. They are inherently different to those tools mentioned before because they are agent-less. Therefore they implement – contrary to the recommendations of [TH98, Push vs. Pull] – push maintenance. This, of course, comes with the downside of having unreachable machines during a particular configuration run. However, not requiring agents on the target systems allows Ansible and Salt to maintain configuration for systems that are not capable of running those agents, such as managed switches.

A simple way to set up congruent systems and immutable infrastructure was possible when "*dotCloud, a PaaS provider, ... open sourced Docker*" [Avr13]. "*Docker*" uses modern virtualization techniques underneath an abstraction layer. The abstraction layer is a configuration file called a *Dockerfile*[16]. It is used to define a system layout, that is, which packages need to be installed, which network ports are exposed, etc. Docker uses `libcontainer` or Linux kernel features such as *LinuX Containers* (LXCs) to avoid requiring running the entire system.

[12]`https://www.linkedin.com/in/lukekanies`. Visited Feb 7, 2015

[13]Puppet's first public commit, Luke Kanies, Apr 13, 2005, `https://github.com/puppetlabs/puppet/commit/54e9b5e3561977ea063417da12c46aad2a4c1332`

[14] `http://redmonk.com/sogrady/2013/12/06/configuration-management-2013/`, Visited April 19, 2015.

[15]"Company Overview of Chef, Inc.", Bloomberg Businessweek, `http://investing.businessweek.com/research/stocks/private/snapshot.asp?privcapId=58274057`. Visited Feb 10, 2015

[16]`https://docs.docker.com/reference/builder/`

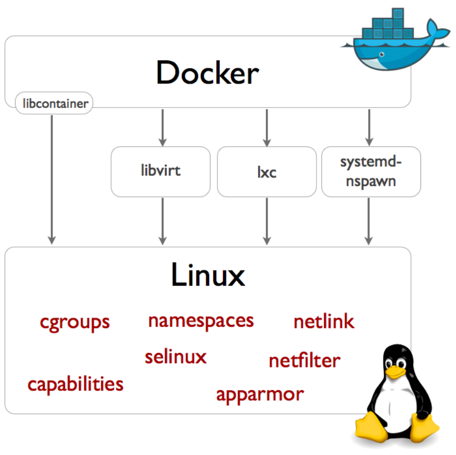

Figure 4: Illustration of the interfaces Docker uses to access virtualization features of the Linux kernel [The14a]

Another congruent system that has gained more and more traction is "Rocket" by CoreOS[17]. The team developed it because Docker became more of a "Docker platform" and not only a container format.

A "*unique approach to package and configuration management*" is the "NixOS" Linux distribution[18]. It can be seen as a combination of a convergent and a congruent system. Being able to make changes without the need to rebuild the entire system make NixOS a convergent system. Having consistent states that will never be modified, on the other hand, allows for stable rollback operations and gives the system congruent behavior.

[17]`https://coreos.com/blog/rocket/`
[18]`http://nixos.org/`

# 4 Background

The knowledge provided in chapter 2 – *Basics* only covers the information that non-system administrators need to understand. This chapter gives additional, in depth knowledge on topics every reader of this thesis needs to understand and be aware of.

## 4.1 Configuration Changes

In order to propose ideas of how to rollback or undo a modification done by a configuration management tool, it needs to be clear what a change is and how multiple changes relate to each other.

A change can be defined as the difference between two descriptions of statuses of software and virtualized hardware states, such as software installation, software configuration (e.g. IP address routing, mounted directories), hardware configuration (e.g. bound IP addresses), amount and type of virtual hardware (e.g. CPUs, memory, disk storage). It reflects what a configuration and system management tool needs to do to turn the configuration and setup of a target host into what is defined by the new description.

## 4.2 Differences Between Local and Distributed Configuration Changes

When looking into configuration management changes, one has to distinguish between *local* changes and *distributed* changes:

**Local configuration changes** are changes that do not influence other systems, neither directly nor indirectly. Those changes could be, for example, running a different kernel version or changing the disk size.

**Distributed configuration changes,** on the other hand, are changes that have a direct or indirect impact on other systems. An example would be changing an IP address or the port a specific service listens to.

There are no distributed changes in non-distributed systems simply due to the fact that local systems[19] are systems that do not interact with other systems.

However, whether or not the reverse holds true is not immediately obvious. Depending on the level of abstraction, local changes (to a single “node” or multiple “nodes”) in a distributed system may or may not influence the entire system in theory or in practice. While the above examples of running a different kernel version or changing the disk size do not influence other components of the distributed system in theory, changing a kernel in practice often implies performance, timing and feature changes that could result in other systems behaving differently. If changing the disk size includes replacing the hard drive, then the I/O performance will likely change.

Distributed configuration changes in distributed systems obviously imply different behavior between two or more nodes: if the port the database application is listening on changes, all applications that access the database must change the port to connect to as well.

## 4.3 Configuration and System Management Tools

Configuration and system management tools are used to keep the configuration files and resource settings on servers in a predefined state. Furthermore, they are used to keep the configuration and settings on multiple servers in sync. While computer systems’ configuration will diverge without usage of configuration management software, as explained by Traugott and Brown [TB02], such software can help turn a divergent system into a convergent one, which at some point can become a congruent system.

Comparing convergent management tools (as defined in chapter 3.1.3 – *Convergent*

[19]In this context “non-distributed systems” and “local systems” are used conterminously for clarification.

*Configuration Management*) like *Chef*, *Puppet* and *Ansible* shows, however, that they are quite similar in the way they represent a system's state: they do not actually store the differences between states like "*install software x*" but opt for a global state that is later compared to the actual state of a system.

This behavior results in a congruent system when bootstrapping a new system. When working with mutable systems, however, it is inevitable that it leads to some kind of divergence, for example, a program is installed and then removed. On a new system, the global state directly describes that this program must not be installed.

An example project that stores the actual tasks to be done between state changes is the database migrations framework of the Django web framework, although it is not a system management but database schema management tool.

> *The operations are the key; they are a set of declarative instructions which tell Django what schema changes need to be made. Django scans them and builds an in-memory representation of all of the schema changes to all apps, and uses this to generate the SQL which makes the schema changes.* [Dja15]

## 4.4 Problems to Consider

A common problem in distributed systems is the lack of a global state. Consider the example of a system receiving a status update. By the time the system receives the update, the status on the sending system might have already changed again[20]. In order to work around this issue, configuration and system management tools have to be aware of the problem and create applicable solutions that provide a certain level of consistency.

Aside from that assumption, this thesis will cover different problems that occur when rolling back changes. These problems can emerge for different reasons, one

[20]This is similar to Heisenberg's Uncertainty Principle from quantum mechanics which asserts that the precision of knowing the position and momentum of an object simultaneously is limited.

being that later changes depend on the change that is about to be rolled back. Most problems then need special handling that takes other tasks into account in order to proceed with a successful rollback.

A few things that will be explained in depth in chapter 5 – *Resource Classification and Utilization* are situations related to the installation, update and removal of programs (5.3.1 – *Installation, Update and Removal of Programs*), how to handle the removal of a user (5.3.2 – *Modification of User Accounts*) and how to safely remove files and directories where mounted file systems are involved (5.3.3 – *Modifications Involving the File System*).

Furthermore, problems that occur when changes affect multiple resources, or when the order in which changes are applied is important, will be discussed in chapters 5.3.4 – *Network Setup Modifications* and 5.3.5 – *Ordering of Changes in Distributed Systems.*

By taking the entire state into account during the "runs" of the configuration and system management tools, these tools can guarantee that all the facts it knows about the system are configured in the way they should be. On the downside, those runs can easily take minutes until all servers have been completed. In contrast, knowing the different operations between states and the last applied state, as in Django's migration framework, allows much faster runs when the last applied state is known.This strategy has the drawback of not being able to ensure a congruent system but still yields convergent systems.

## 4.5 System Stability and Security

### System Stability

The term *system stability* is often used in a colloquial fashion when it describes a user's experience of how often a program or operating system stops working. This could be as simple as the "*Program is not responding.*" error message in Microsoft Windows or as bad as a kernel panic with immediate reboot, in each case giving no indication of what went wrong. In a regular user's understanding, system stability is solely based on their experience, feelings, and expectations, and not guided by any numbers or measurements.

When looking at the system stability from a technical and mathematical perspective, it defines the percentage of time a system works without crashing and how often a system has been used at all:

$$r = \frac{s}{s+f} \qquad f, s \in \mathbb{N} \tag{1}$$

$$\lim_{\substack{s\to\infty \\ f\to 0}} \frac{s}{s+f} = 1 \qquad \text{stable system} \tag{2}$$

$$\lim_{\substack{s\to 0 \\ f\to\infty}} \frac{s}{s+f} = 0 \qquad \text{unstable system} \tag{3}$$

where $s$ is the number of successful usages and $f$ defines the amount of crashes. The higher $s$ and the lower $f$, the more stable a system is. A completely stable system has a stability ratio of 1. A high value for $f$ and a low value for $s$ on the other hand, describes a unstable system.

Looking at Jim Grey's "*Why Do Computers Stop and What Can Be Done About It?*" [Gra85, p. 3], the formula defined above is the same as the formula describing the availability based on the *Mean Time Between Failure* (MTBF) and *Mean Time To Repair* (MTTR):

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \tag{4}$$

MTBF is effectively the time a system is operational and stable ($s$), and MTTR the time a system is unstable or faulty ($f$).

### System Security

Unlike system stability, *system security* does not concern the usage experience of a system but relates to the security of of the data provided or used by the system.

According to *A Dictionary of Computing* by the *Oxford University Press*,

> *[an] (operating) system is responsible for controlling access to system resources, which will include sensitive data. The system must therefore include a certain amount of protection for such data, and must in turn control access to those parts of the system that administer this protection. System security is concerned with all aspects of these arrangements.* [Dai04]

In other words, system security, or in particular *computer security*, maintains CIA:

- *Confidentiality* by preventing unauthorized disclosure of information.
- *Integrity* by preventing unauthorized modification of information.
- *Availability* by preventing unauthorized withholding of information.

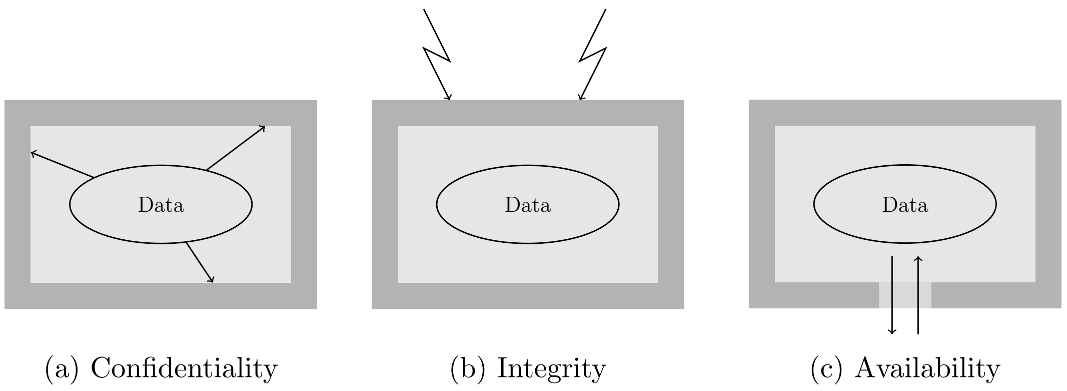

Figure 5: Schematic representation of computer security attributes [Fel13]

This chapter introduced the ideas behind configuration and system management tools, how they work in general and what problems occur when they are used in distributed environments. With the explanation of mutable and immutable infrastructure and the knowledge of what system stability and security is, the next chapter will go into how different kinds of changes can be handled.

# 5 Resource Classification and Utilization

Following the introduction to configuration and system management tools, the thesis will continue with a detailed differentiation of types of configuration changes. The chapter will state which general rules one should consider in order to make rolling back changes possible and finish with a set of actual use cases.

## 5.1 Classification of IT Resources

This section defines a taxonomy to classify resources in IT infrastructure. If a resource is managed by a configuration or system management tool, the operations needed to rollback a resource to a previous state can be derived from the rules generated from the taxonomy. The classification is furthermore intended to be used to create rules for operations that are not considered as part of this thesis. Hence the taxonomy is more generic than would be needed to describe only the considered changes.

### Accessibility

The ACCESSIBILITY of a resource defines which permissions regular users have to that resource and which permissions are limited to administrators. A RESTRICTED resource might allow *usage* permissions to regular users while *management* access is only allowed for privileged users. Resources that are only accessible by a subset of all regular users are also considered restricted, although the respective users could have all permissions. UNRESTRICTED resources can be used in any way by all users.

Based on this definition, a network interface is an example of a restricted resource. Regular users can use it, but the ability to manage the interface remains restricted to root or a set of administrators.

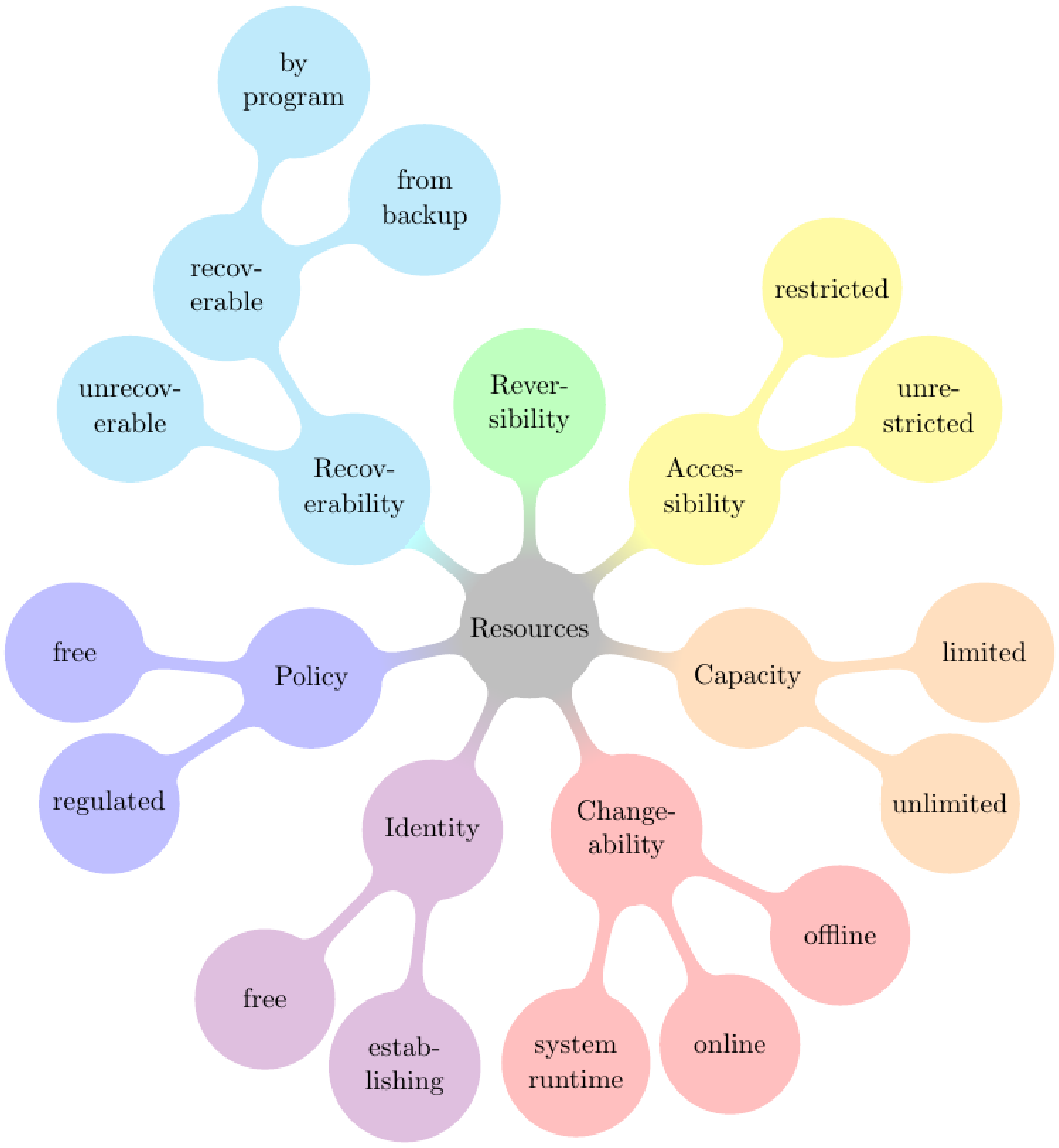

Figure 6: Taxonomy for resources in IT environments

### Capacity

Even though many resources in IT environments are almost UNLIMITED, administrators still need to consider the CAPACITY of LIMITED resources. Anything that can be generated or assigned on the fly can be considered unlimited. This could be SSL / TLS certificates, persistent storage or memory of a virtual machine or IPv6 addresses. On the other hand, UNIX user IDs (originally limited to $2^{15} - 1 = 32767$), IPv4 addresses (due to their widespread use) or the computation time of a CPU should be considered limited in systems.

### Changeability

The CHANGEABILITY of resources can have serious impact on the availability of a system setup. When a resource is changed ONLINE, there are no interruptions of other resources. The changes to the resource are applied while the remaining resources of the system are regularly working. Conversely, an OFFLINE change requires a service interruption. This could be a kernel update or a changing of the number of CPUs. Changing a single service on a system can happen during the SYSTEM RUNTIME and may only require a service restart.

### Identity

With security considerations in mind, resources can provide some kind of IDENTITY in which case they are IDENTITY ESTABLISHING while other resources are IDENTITY FREE. Resources that provide identities are e.g. SSL / TLS certificates, IP addresses, user IDs, usernames or email addresses. Other resources, such as a CPU or a software, are identity free.

### Policy

The existence of a resource can be dependent on outside information that is not handled by configuration and system management tools. In this case, the resource is POLICY-REGULATED. This is often seen in corporate environments where the existence of a user account is defined by the user's employment status. For email providers, the existence of an email address could e.g. be subject to the payment status of a user's account. Resources that don't rely on outside information are POLICY-FREE.

### Recoverability

Since every computer system fails at some point[21], the RECOVERABILITY of resources is important in order to minimize the risk of data loss or service interruption. Basically all files that are not modified by a third party, that is, only changed by the package manager or the configuration management tool, can be considered RECOVERABLE BY A PROGRAM. Files that are part of a working backup strategy are RECOVERABLE FROM A BACKUP. Files that are neither auto-generated nor part of a backup are lost during a crash and are thus UNRECOVERABLE.

### Reversibility

The REVERSIBILITY of a resource is particularly important with regards to the topic of this master's thesis. If a resource is not reversible, the change introducing the resource cannot be (automatically) undone. An example could be removing user data that is unrecoverable. Unless the user itself has a way to reconstruct the data, removing it is not reversible.

## 5.2 Rules Derived From the Taxonomy

### Accessibility

Denying users or groups a certain access in a forwards change reduces the risk of exposing that resource. This comes with the downside that programs running

[21]It is practically impossible to write software that does not have bugs.

with the permissions of previously permitted users might stop working and fail with "*Permission denied*" errors when trying to access the resource.

Increasing the access level or the set of users that have access to a resource increases the probability of exposure of sensitive data. If resources contain sensitive data this is likely to be known beforehand. Thus this change is unlikely to occur except for intended changes.

However, rolling back a permission change or increasing the set of users having access to a resources can often result in exposure of secret information. This is especially true when granting access to more users. Furthermore, files may not have been supposed to be accessible by those users in the first place or the resources' confidentiality may not have been assured before.

**Rule 1: Be careful when changing resource accessibility**

When changing the permissions or the set of users that have access to a resource, the resource's confidentiality, integrity and availability must be re-evaluated based on its content and use case.

### Capacity

There are a lot of resources in today's IT environments that are still limited to a maximum number of items. One of the problems larger IT data centers are faced with is the lack of free IPv4 addresses for public use[22].

Although a limited resource may not be in use at a given time, this does not imply it can be used for another service. When deallocating the resource and putting it back into a pool of unused resources, the resource can be used again for a different service. However, rolling back the deallocation then leads to problems due to an unavailable resource.

[22]According to the *IANA IPv4 Special-Purpose Address Registry* [IET14], about 7.55% of all IPv4 addresses are reserved and should not be publicly routable.

**Rule 2: Be careful when reusing limited resources**

> A resource must only be reused once all references, including historic ones, to it are removed. Reusing a resource with historic references must mark the historic changes including the references as not reversible.

Depending on how user accounts are managed for a larger amount of users, the original restriction of UNIX user IDs to a total of $2^{15} = 32768$ (including the root user) without duplicating the ID, could pose a problem. Newer UNIX or UNIX-like systems allow up to $2^{32}$ user IDs. The concrete limit can be found by looking at the type definition for `__UID_T_TYPE` in `/usr/include/bits/typesizes.h` which mostly corresponds to `__U32_TYPE`. This in turn is defined as `unsigned int` in `/usr/include/bits/types.h`. `/usr/include/limits.h` holds the definition of `UINT_MAX` which is defined as `4294967295U` on a 32 and 64 bit GNU/Linux. Thus this particular restriction is not a problem anymore.

### Changeability

Categorizing the various components of a modern IT system, one can see three different situations with respect to a service's or file's lifetime as described above with the keyword CHANGEABILITY. While updating a PHP application can happen while the web server is running and does not require a restart of the underlying PHP interpreter, a WSGI container running a Python web application needs to be restarted. The key difference is the availability of the respective application: While the PHP application might be unavailable during the time the new file is written but mostly feels like instantaneously being updated, the restart of the Python application needs a short time to boot-up again.

Another example that describes a change during system runtime can be an update of a database application: the database and all services depending on its availability[23] will not work. On the other hand, services that do not depend on the database or any of its dependents, e.g. a DNS service, will continue working.

[23] Assuming there are is no replication set up or the clients are not configured to fall back to the replications.

**Rule 3: Estimate downtime and check dependencies**

Changing data files, configuration files or the respective programs themselves requires knowledge of how long the system or component will be unavailable as well as a careful dependency checking to list other services that will be affected.

#### Identity

In order to restore services after a system crash, backups are made. These backups have to be in a working state in order to prevent misbehaving systems after a successful restore. However, if these backups include identifying resources, the whole backup needs to be protected. The access to a backup should only be granted to those who can see all included information on the live system as well. If a backup e.g. includes the file `/etc/shadow`[24], only users with root access on the respective server should be able to access the backup.

**Rule 4: Keep identifying resources secret**

Keep identifying information in a separate backup that can be applied back after the "main" backup has been applied, to minimize the risk to expose secret data.

As soon as an identifying resource is not in use anymore, this resource should be invalidated and further and future usage should be prevented. For SSL / TLS certificates this would imply putting them on CRLs and e.g. publishing the revocation information via OCSP responders. Email addresses should not be reused but marked as "not available anymore" to prevent a second owner gaining access to resources the first owner had access to.

Similar to the deallocation of limited resources, one has to consider whether revoking an identifying resource is an option. If the resource is still inside a backup, where it might be retrieved from to get a service running again, revoking that resource can lead to problems when a service is restored from the backup.

[24]The file containing all user account passwords on UNIX system

**Rule 5: Revoke identifying resources**

As soon as an identifying resource is not in use anymore, neither directly nor as part of a backup, the resource should be revoked and access and future usage should be declined. Resources with limited capacity may be reused if need be.

Identifying resources must not exist more than once by the very nature of being unique in a certain context. In case these contexts overlap due to some environmental changes resources may not be unique anymore. Thus one should make sure to make identifying resources globally unique.

**Rule 6: Use environment-wide unique identifiers**

In order to prevent possible problems with clashing identifiers at a later point in time, make sure identifiers are environment-wide unique from the beginning.

### Policy

POLICY-FREE resources should exist upon the question "Is the resource used?". If the answer is "yes" the resource obviously has to stay and can therefore not be rolled back. If the response to the question is "no" the resource *can* be removed, though. For POLICY-REGULATED resources, however, the "outside world" defines whether the resource may exist or must be removed. The question would rather be "Is the resource allowed to exist?". In case a resource must not exist, all other resources depending on it must either be removed or be reassigned to another resource providing the service.

**Rule 7: Review depending resources**

When removing resources all depending resources are either deleted or reassigned. The deletion should only happen with recoverable resources according to *Rule 9: Prune files and folders with package manager* and *Rule 10: Remove files managed by a configuration and system management tool*

Since removing policy-regulated accounts always comes with the risk of accidentally removing too much data, resources that are independent of a particular employee, e.g. a database with data of a company's service offering, should be bound to its own role account which will have its own policy.

**Rule 8: Bind production resources to role accounts**

To reduce the chance of accidental removal of organizational data, bind resources to role accounts with their own policies. This way the removal of a departing employee's account does not jeopardize that data.

### Recoverability

Before one should remove files their RECOVERABILITY needs to be evaluated. Build upon the above explanation about recoverability, the following rule for files and folders managed by the system package manager can derived:

**Rule 9: Prune files and folders with package manager**

Files and folder structures that are created by the system's package manager can be removed if they match the files or folders from their respective package and are thus not modified. For folder structures, the additional requirement that these folders must be empty and not contain user data applies.

Files that are handled by a configuration and system management tool can also be removed if they are not modified. If these files were modified, one already has a divergent system which should not be the case in the first place. Furthermore, these files are generally either copied to the system or generated on-the-fly by the configuration management tool based on the information the tool knows about the system.

**Rule 10: Remove files managed by a configuration and system management tool**

Files that are managed by a configuration and system management tool can be considered unmodified as the next run of that tool would override the files content and permissions anyway. Defining the absence of a file effectively will remove the file from the system.

All files that are part of a backup could be recovered if they are deleted. Depending on the size of the backups and whether they are full backups or differential backups, restoring from a backup can take a long time, however, so it's better not to make a mistake which requires restoring from a backup.

Unrecoverable files cannot be restored once they are deleted[25]. Therefore their removal must be cautiously planned and a removal of files not to be removed must be prevented.

This also indicates that file removal – or data removal in general – of unrecoverable resources is not reversible.

### Reversibility

The reverse operation of a remove operation can also be ambiguous: a software that is being uninstalled can either be marked as "never been installed" or "uninstalled" in the database of the underlying package manager.

[25]This does not consider the usage of forensic software to reconstruct removed files by reconstructing the appropriate *inodes* on a file system, as this is not part of this thesis.

**Rule 11: Check reversibility of resource changes and specify outcome**

Changes made to resources can be one-way operations and thus do not have an unambiguous way to undo, in which case the result of the reverse operation must be defined.

## 5.3 Use Cases of Configuration and System Management Tools

The variety of tasks that configuration and system management tools were originally developed for is rather small compared to what those tools are used for in today's IT infrastructure. It is not just the management of users or the list of installed programs anymore, but includes more complex processes like generating SSL / TLS certificates, moving virtual machines between nodes while they are still online or deploying applications from a source code repository.

Based on the classification of IT resources in chapter 5.1 and under due consideration of the rules developed in chapter 5.2, this chapter will highlight common use cases of these tools in modern IT environments. Along the different use cases and their implementation, this chapter will also point out the problems arising when a rollback operation happens.

### 5.3.1 Installation, Update and Removal of Programs

#### Program Installation

Installing new programs or libraries on a system, in either local or distributed environments, usually does not affect other parts of the system. However, especially in UNIX environments, installing additional libraries often allows other programs to expand their capabilities by dynamically loading these libraries at startup. While these additional features can create an unexpected behavior in another system, this principle follows *The UNIX Philosophy* by Mike Gancarz "Make Each Program Do One Thing Well" [Gan95, Tenet 2, p 19].

In terms of reversibility, a program installation mostly has a direct affect on other resources in the environment. If the program is required by other resources, these resources will change their behavior and could stop working entirely until the program is installed again.

### Program Update

Contrary to the installation of a program, updating to a newer version of a program may result in serious behavioral changes. If a program follows the *Semantic Versioning* concept [PW], updates changing the major version number introduce backwards incompatible changes and thus often lead to severe problems in a production environment if not explicitly taken care of. Patch releases, on the other hand, should be fully backwards compatible with the previous minor release.

A program downgrade, as the reverse operation for an update, follows the same rules as the update process. Depending on the versions the process should not have any influence on other resources, or needs to be taken care of explicitly.

### Program Removal

Unlike the program installation, a program removal implies that a certain functionality is not available anymore. Although this seems obvious, the changes to the system along with removing a library often result in errors.

One needs to take into account when possible problems may manifest themselves. Programs dynamically linked to shared objects, that only load them when available, still have that library in memory as long as the process is running. After a restart of the program this library is not loaded anymore and a change in the behavior of that program is the result. On the other hand, programs that explicitly invoke another program that has just been uninstalled will fail[26].

Installing new programs also often adds some default configuration files. These

[26]That is, if the calling program does not catch the error and continues working as if nothing happened, like when the called program would still exist.

files will remain on the disk when a program is removed if they have been modified and thus are not identical to the files from the package manager. Hence installing a program after it has been removed before may result in different behavior than anticipated if a preexisting configuration file is still around.

| Classification | Property |
|---|---|
| Accessibility | Modifications of system-wide installed packages are restricted to administrators |
| Capacity | Unlimited; only limited by the amount of packages bundled by the vendor |
| Changeability | Depends on the program and might need program restart or server reboot |
| Identity | Free |
| Policy | Free |
| Recoverability | Programs and libraries are installed through a system management tool and are therefore recoverable by program |
| Reversibility | The reversibility is only guaranteed if all dependencies are known |

Table 2: Classification of programs / libraries installed via a system's package manager

### 5.3.2 Modification of User Accounts

#### User Creation

Creating new users on a local system is straight forward: if the user (identified by a certain name and/or id) does not exist, create the user. Related tasks, such as creating the user group and user's home directory, are not considered as part of the user creation process as they can be abstracted to their own operations. Creating the home directory is handled in chapter 5.3.3 – *Modifications Involving the File System* section *Directory Modifications*. The user creation would then depend on this operation.

In distributed systems the process is more complex. If the environment has a central user database (e.g. LDAP), the user has to be created in that database if it does not exist. Potential dependencies need to be resolved before and may involve the entire environment.

If no central user database exists in a distributed system, the creation process becomes even more complex. Two users with the same identifier on different systems may be completely unrelated but should still be the same across a different set of systems. As long as their identifying part is unique throughout the entire system, this does not pose a problem. According to *Rule 6: Use environment-wide unique identifiers* identifiers, should be unique in the entire environment.

As an example: there are two clients ClientA and ClientB that each have an employee John with a user account "john". While this account is unique on all servers belonging to either ClientA or ClientB (but not both) the account identifier is not unique across the entire system.

Figure 7 shows 6 servers that are divided in 3 sets (red / vertical pattern, blue / horizontal pattern, yellow / diagonal pattern). The violet / cross pattern colored server is part of the red and blue set. In case there are two users with the same identity shared across the red and blue set, this will result in an identity clash on the violet system since the two users are not distinguishable.

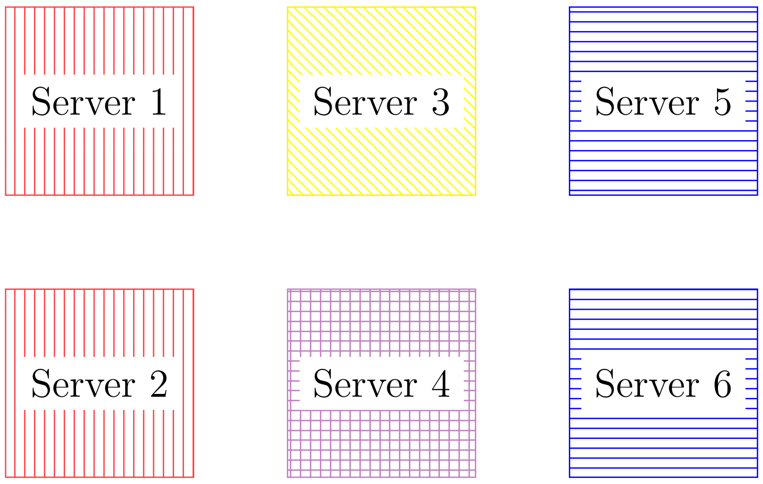

Figure 7: Clashing user identities from servers 1, 2 and 5, 6 on 4 in distributed systems without global user database.

### User Removal

When removing a user account, one has to review if the account is policy- regulated or policy-free (see chapter 5.1 – *Policy*). As already elaborated above, removing a policy-regulated account implies a removal or reassignment of dependent resources, while removing a policy-free account must be prohibited until no other resource depends on the account. This can also be seen by the direction of the arrows in figure 8. They are effectively 180° turned, illustrating the dependency direction.

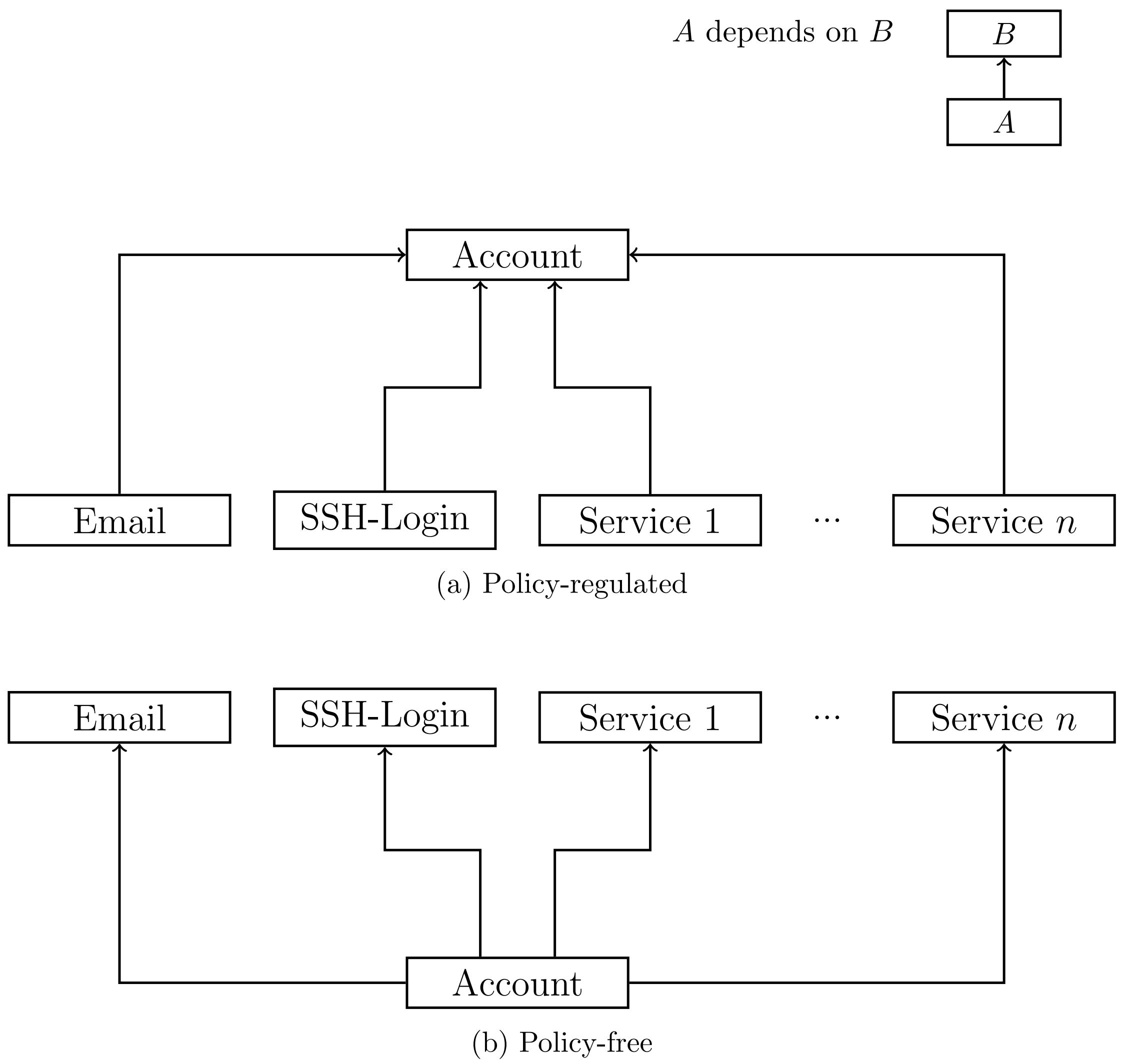

Figure 8: Dependencies for policy-regulated and policy-free resources

**Policy-free** Removing a policy-free user account, not just a permission or access to a specific system but also the user data, is not trivial in terms of recoverability. By the very nature of not knowing if all user data is backed up by its owner – and thus manually recoverable by them – one needs to evaluate if the user data is relevant to be included in backups.

Email addresses are often closely related to user names and sometimes are even the user name with an appended domain (`@example.com`). Thus removing a user account implies that the email address is not linked to a specific user account, anymore and can be removed. When removing the account the contained emails may have to be preserved for tax or legal reasons. This normally happens outside the email account as part of a backup strategy.

However, since the email address is unused it could be reused by somebody else. Reusing a previously used email address – and identifying resources in general – raises the questions, though, if this is a good idea with respect to data privacy protection. Especially for corporate email addresses, statutory data privacy policies may apply and have to be taken into account. Furthermore, when an email address has been used for signing up on third party services, reusing an address could allow the new owner to gain unprivileged access to the third party service. Thus a full revoke according to *Rule 5: Revoke identifying resources* should be considered.

**Policy-regulated** On the other hand, removing a policy-regulated user account is rather deterministic in what is going to happen with resources belonging to it. The common action is the deletion of those resources. This is fine for programs and recoverable configuration files. However, removing a production database owned by that user is not an option. Although the database will be backed up, the latest updates probably occurred after the latest backup. Thus, the deletion must be avoided. Instead the database should be reassigned to another account, following *Rule 8: Bind production resources to role accounts.*

Taking both the user creation and removal process into account, the following properties for user accounts according to the taxonomy stated above can be derived:

| Classification | Property |
|---|---|
| Accessibility | Creation and deletion of user accounts is restricted to administrators; changing the password to the user itself. |
| Capacity | Unlimited, at least on newer UNIX systems over 4 billion user accounts can be created locally |
| Changeability | The identifying part of identifying resources is not changeable though additional information about a user account can be changed without causing problems. |
| Identity | Identity establishing |
| Policy | Free or regulated; depending on the IT environment |
| Recoverability | The accounts themselves are recoverable by a program; the user's data not necessarily. |
| Reversibility | Reversing a user creation or removal operation is generally possible, though not recommended. |

Table 3: Classification of user accounts

### 5.3.3 Modifications Involving the File System

#### File Modifications

A common use case for configuration and system management tools is the modification of configuration files and keeping these files in sync across multiple systems. Modifying a file does not only mean changing the file's content, but also takes changes to the owner or group and other permission attributes into account. Hence *Rule 1: Be careful when changing resource accessibility* should be closely taken care of.

When changing a file's content, which is part of an installed package that was installed through the system's package manager, the package manager has to consider this file user-modified and thus not delete it when removing the program. This can eventually lead to several problems:

1. The file remains on the file system and is still there when the program is installed again at a later time. The package manager will likely not complain

about the pre-existing file since the file was modified by a user before. This can then result in unexpected behavior. Instead of a default configuration, an old configuration is loaded which might expose the system to an insecure network and make it vulnerable.

2. Since a file the package manager is going to install already exists on the file system but does not match the file from the package, the package manager could bail out and prevent an installation due to a pre-existing file that does not match the file from the package.

To prevent those problems, the configuration and system management tools should remove the files explicitly as mentioned in *Rule 10: Remove files managed by a configuration and system management tool* due to them being modified by the management tool.

| Classification | Property |
|---|---|
| Accessibility | Most files are readable by every user on a system; access to configuration files is often limited to the user running the program |
| Capacity | Unlimited; only limited by the size of the hard drive |
| Changeability | Online |
| Identity | Files are normally identify free, though cryptographic files (SSL / TLS certification / keys, SSH keys) are likely identity establishing |
| Policy | Mostly policy-free. Files in `/etc/skel` are copied to home directory of local users |
| Recoverability | Only recoverable if files are in a backup; not guaranteed for user data |
| Reversibility | Only reversible when files are managed by a configuration management tool or the package manager |

Table 4: Classification of files

### Directory Modifications

When modifying directory permissions, the files and folders contained inside that directory potentially get exposed to a wider group of people. Thus following *Rule 1:*

*Be careful when changing resource accessibility* is required.

Recursive modifications of a directory have always to be handled with care and can lead to serious damage of the directory structure and may even result in data loss. If a directory is a mount point itself or if any of its subdirectories are mount points of currently mounted devices, recursive operations should be avoided.

| Classification | Property |
|---|---|
| Accessibility | Most directories are readable by every user on a system; access to the home directories is often limited to the respective owner |
| Capacity | Unlimited; only limited by the size of the hard drive |
| Changeability | Online |
| Identity | Free |
| Policy | Free |
| Recoverability | Only recoverable if in a backup; not guaranteed for user data |
| Reversibility | Only reversible when managed by a configuration management tool or the package manager |

Table 5: Classification of directories

### Directory Mounting

Mount points are defined in the file `/etc/fstab`. Configuration and system management tools modify this file to permanently define mount points of local devices and remote resources into the root file system. Unless specified otherwise, the mount points are automatically mounted upon system boot. Especially for remote resources, their changeability has to be kept in mind. In case of e.g. a NFS storage server, one has to follow *Rule 3: Estimate downtime and check dependencies*. Before updating the server one should unmount the NFS shares on all servers in order to prevent access locks. Otherwise applications trying to read or write from the remote file system could hang without timing out.

When a directory is going to be mounted at a later point after system start and thus all services are already running, the services depending on the not yet mounted directory must not be started automatically. Otherwise they can end up using the file system containing the mount point directory. For backup processes this can especially be a problem when, due to missing previous backup data, a full backup is performed filling the remaining disk space.

#### Directory Unmounting

Unmounting a directory is only possible if no running process is accessing any files inside that mount point. Otherwise the unmount process fails. Periodic tasks, however, that are running a main process and spawn subprocesses to perform the actual tasks, either in random intervals or on a regular basis, potentially suffer from a problem mentioned above: If a mount point is unmounted after the main process has been started but before a subprocess starts (again), the subprocess will try to access files and directories from the mount point's file system and not from the previously mounted device or remote resource. Thus a e.g. backup job would write to the main file system and not to the NFS. The server's hard drive will fill up and eventually the backup process will fail once no space is left on the device.

A possible work around to counteract the problem could be monitoring access requests to mounted resources and only unmount them once no access happened for the interval of regular tasks or the maximum interval between two randomly scheduled tasks.

### 5.3.4 Network Setup Modifications

Among assuring the installation state for programs, the existence or absence of user accounts and keeping track of file content and file permissions, configuration and system management tools are heavily used to manage the network setup of single systems and entire data centers. This includes assigning, changing and releasing IP addresses on a server as well as updating its routing setup.

Before adding an IP address to a server, one has to make sure the IP address is not in use. Otherwise the routing of incoming and outgoing packages to and from that server is not guaranteed to work. Depending on the actual setup, adding an IP address can even make other servers unavailable.

Unlike adding an IP address, removing one does not only come with the same risk of rendering (parts of) the network unavailable. It will also make other services depending on a particular host to be unavailable.

| Classification | Property |
|---|---|
| Accessibility | Changes to network interfaces is limited to administrators |
| Capacity | Limited; there is a physical limit of devices per host though virtual interfaces can be created as on demand |
| Changeability | Changing an IP address or routing configuration can happen during the system runtime; no reboot required |
| Identity | IP addresses or MAC addresses are identity establishing |
| Policy | Free |
| Recoverability | The setup is performed by configuration and system management tools and thus recoverable |
| Reversibility | Depends on the entire network setup |

Table 6: Classification of network interfaces

### 5.3.5 Ordering of Changes in Distributed Systems

All use cases explained above are similar in that the actual changes required have only local affects. In larger environments, however it is not uncommon to have dependencies between services running on otherwise unrelated dedicated hosts.

As an example, a proxy connects to a server on port 8000. For some reason the port the server listens on must be changed to 8080. Recalling that changes can never be guaranteed to happen synchronously, there are two approaches to achieve the change:

1. Change the proxy settings to point to the new port, restart the proxy, change the port the server listens on, restart the server
2. Change the port the server listens on, restart the server, change the proxy settings to point to the new port, restart the proxy

Both approaches inevitably result in timeouts of requests as the proxy either tries to connect to a port that is *not in use yet* (1) or *not in use anymore* (2).

Depending on the uptime requirements of the service, this may not be acceptable.

This chapter provided seven classification criteria and formulated 11 rules which were combined with several use cases to show some problems system administrators have to take care of. The next chapter will work on these use cases and provide examples how to solve them.

# 6 Maintaining System Stability and Security

Based on the use cases given in chapter 5.3 – *Use Cases of Configuration and System Management Tools*, this part of the thesis will follow the rules compiled in 5.2 – *Rules Derived From the Taxonomy* from the IT resource taxonomy to design and *evaluate methods to maintain system stability and security when reversing changes made by configuration and system management tools.* These methods can then be used to act accordingly in the given cases. The methods will also outline possibilities to handle similar use cases and give system administrators guidelines to consider.

## 6.1 Installation, Update and Removal of Programs

One of the main actions performed through configuration and system management tools is the management of installed packages, i.e. programs and libraries. As outlined in chapter 5.3.1, while the solution to many problems may be obvious and may even seem trivial to take care of, there are circumstances – especially taking care of package updates that would break other packages – that need careful consideration and testing[27] before rolling them out to production.

### Dependency Problems

A common problem faced by administrators is dependency errors. A package `foo` in version `1.0` requires package `bar>=1.0,<2.0`. Updating `bar` to `2.0` should be prohibited if `foo` must still be installed. Packages that are part of the official distribution package repositories are not considered to be a problem when it comes to dependency problems and maintaining system stability, because their maintainers make sure all requirements are up to date. The problem is programs and libraries that are e.g. self-compiled and are manually put into appropriate places to make them look like system packages. If `foo` from above is such a self compiled package,

[27]In fact, all changes should be tested on a staging system before being rolled out on a production system!

the system package manager would not know about it and thus would not reject updating `bar` to `2.0`. Using `foo` will eventually fail; tracing the error can be time and cost expensive.

A logical step to prevent the respective package from updating is to tell the package manager that a particular version is used / required; in other words, that a particular state of a package must not change. The equivalent command to do that on a Debian-based Linux distribution for `bar` is `apt-mark hold bar` [Arc15]. While this solution works, packages depending on `bar` evolve and at some point pinning the version of a particular package can prevent other packages from being updated due to unmet dependencies.

By using *APT pinning* on Debian-based OSs, a package that is available in multiple versions (e.g. `bar` in both `1.0` and `2.0`) can be pinned to a specific version or version range[28]:

```
Package: bar
Pin: version 1.*
Pin-Priority: 1001
```

This would allow the package manager to use the latest *possible* version of that package that is supported by other installed packages. However, this behavior is not true for APT as it tries to update to the latest *available* version of a package if no other version has a higher pinning (priority). Hence, all packages that allow `bar >=2.0` would need to be pinned to their latest version requiring `bar >=1.0,<2.0`.

A better solution that can dynamically cope with possible updates of a dependency is the development of an actual package that can be installed by the package manager. If package `buz==1.0` depends on `bar>=1.0,<2.0` and an updated version `buz==2.0` on `bar>=1.5,<2.0`, a *hold*, due to the above restrictions, of `bar==1.4` would prevent an update of `buz` from `1.0` to `2.0`. Having `foo` as a package that is installed through the package manager would allow the package manager to update `bar` to e.g. `1.9` as this is supported by both `foo==1.0` and `buz==2.0`.

[28]Debian man page for `apt_preferences`: `http://manpages.debian.org/cgi-bin/man.cgi?sektion=5&query=apt_preferences&apropos=0&manpath=sid&locale=en`

Rolling back to a previous version of a program can be seen as an "update" to a specific version which was just released earlier than the currently installed one. If all programs and libraries on a system are installed through the package manager, removing a package with still installed dependencies will fail, unless those dependencies are explicitly removed, too. However, if there are "manually compiled" programs or libraries on the system – e.g. users compiling programs in their home directory – those programs can break if their dependencies are not *Application Binary Interface* (ABI) compatible or even unavailable.

| Advantages | Disadvantages |
|---|---|
| **`apt-mark hold`** | |
| • No surprising updates | • Not flexible<br>• Not an option if multiple custom packages need different versions |
| **APT pinning** | |
| • Allows for version ranges<br>• Easy to work with multiple packages | • Becomes confusing when multiple packages are involved |
| **regular packages** | |
| • Package manager is made aware of dependencies for custom packages<br>• Works with multiple conflicting versions | • Overhead to build and maintain the package |

Table 7: Advantages and disadvantages of different approaches to prevent breaking custom programs

### Stale Configuration Files

While Linux distributions ship ready-to-use configuration files for packages for an average user, administrators normally have special requirements and thus update them to their needs. Changing those configuration files should be done by configuration and system management tools, in order to preserve convergence and keep those files in sync across multiple systems. The general handling of files is covered in chapter 6.3 – *Modifications Involving the File System*.

When removing a package, however, a package manager may or may not remove configuration files. Their behavior depends mainly on two factors:

1. **The file state:** A file that matches the file from a package installed by the package manager is likely to be removed. A file that has been changed and therefore does not match the corresponding file from an installed package, on the other hand, is more likely not to be removed.
2. **The command that is run:** Some package managers have multiple ways of uninstalling / removing a package from a system. Some of those commands leave configuration files behind even though they did not change. Other commands also remove configuration files that have been modified.

The Linux distribution implicitly defines which package manager is being used. In the case of Debian and Ubuntu `APT` is the default. To remove a package `apt-get remove somepackage` or `apt-get purge somepackage` can be used. The former does not touch configuration files at all and keeps them in place. Hence modifications to a configuration file are still there after the package has been removed. `apt-get purge`, on the other hand, cleans all files associated with that package and also removes files that have been modified and do not match the file that is included in a package.

| Advantages | Disadvantages |
|---|---|
| Removal of unmodified files | |
| • Keeps the system clean | *None* |
| Removal of modified files | |
| • Prevents surprises when reinstalling a package | • Might remove information that cannot simply be recovered |

Table 8: Advantages and disadvantages of removing modified and unmodified files

A package manager that works differently is `Pacman` used by Arch Linux: instead of removing a modified configuration file, the file is renamed to include an additional `.pacsave` extension. This allows an administrator to have a look at differences between a configuration file after a package has been reinstalled.

| Advantages | Disadvantages |
|---|---|
| • Allows administrators to look up configurations after a program has been removed | • Clutters file system; content should be part of configuration management tools |

Table 9: Advantages and disadvantages of renaming modified files

## 6.2 Modification of User Accounts

### 6.2.1 Creation

#### Local User Accounts

Creating local users with or without configuration and system management tools is simple. In the end the creation boils down to a call to `useradd`[29]. This command can take care of the user's name, ID and groups, creation of the home directory and setting a password, amongst other things.

[29]Man page `man 8 useradd`: `http://linux.die.net/man/8/useradd`

With reference to chapter 6.3, it should be said that the actual creation of the home directory will not be done through the `useradd` command but by leveraging the management tools as stated in 5.3.2 – *Modification of User Accounts*. This firstly allows for better granularity and consistency when it comes to rolling back a user creation, e.g. "is the user data still used?", "is it part of a recent backup?" and secondly, does not restrict the home directory to be created with settings supported by `useradd`. Instead the home directory could be created on a SAN and later mounted into the system.

Adding a user in non-distributed environments without login capabilities managed through `useradd` (e.g. without password, invalid password or inactive) does not pose any security issues iff[30] there is no other way for the user to login, e.g. through SSH public key authentication.

The same holds for local users in distributed systems. This particular situation is more complex, though. In case `/home` is mounted from a central storage and the particular user already has a valid SSH setup, creating the underlying user account on a new system automatically gives the user access to that server. Hence, even though this seems obvious, *creating a user account on a system a user should not have access to should never be done*!

When creating local user accounts in distributed environments, one should ensure the global uniqueness of the user's ID, as this is used on the underlying file system among other things. Keeping the user IDs globally unique makes ID mapping[31] in services like *NFS* and *Samba* needless and follows *Rule 6: Use environment-wide unique identifiers*.

If the usernames are not used for anything else than logging in to a system, they do not have to be globally unique. Uniqueness on a per system level is sufficient. As soon as the usernames are used in a global service though, e.g. as the local part in an email address, they have to be unique. Thus, to be future-proof, usernames should also be globally unique.

[30] *if and only if*

[31] Tools like `rpc.idmap`: `https://www.kernel.org/doc/Documentation/filesystems/nfs/idmapper.txt` or `https://www.samba.org/samba/docs/man/Samba-HOWTO-Collection/idmapper.html`

### Central User Accounts

Central user directories such as LDAP handle the storage of user accounts differently than local users. Users always have an environment-wide unique identifier which is the *Distinguished Name* (DN) in the context of LDAP. Creating another user with this DN is prohibited. Access to a resource can be given based on the location of the user in the directory tree or by the user's properties.

While, by the very nature of being a central system, there is no way to have two distinct users with the same identifier, its setup is more complex and requires more thought. However, there are some benefits in using directory services when it comes to larger environments: when a user wants to change its password, the password can be updated in a single place and is then available to all services that rely on it. For local users, the respective user would have to login to every service and change the password manually. In case of central settings protected from a user's edits, the configuration and system management tool needs to update every service, possibly taking a long time to propagate.

Since practically all major services used in today's IT environments already support LDAP out of the box, there is no real downside of using it instead of local users in distributed systems. Services that do not support LDAP on its own can rely on PAM which itself can use LDAP.

### 6.2.2 Removal

While removing a user does not pose any security risks – it reduces the amount of people with access to a system – it can easily lead to a less stable environment by increasing the number of failing usages $f$, according to the ratio defined in equation 1, or even result in data loss.

Considering the differences between policy-regulated and policy-free accounts, the latter must only be removed when all other resources depending on them are rolled back. The former involves the danger of removing data which could not be recovered because it is not part of a backup.

### Local User Accounts

The process for removing local user accounts in local and distributed environments should *not* delete the home directory when using the underlying `userdel`[32] for the very same reasons that `useradd` does not create the directory.

As a first safeguard, `userdel` will reject a request to remove a user that is currently running at least one process[33]. This does not, however, protect one from breaking a service that is currently inactive. Once the service gets started as a specific user or wants to drop privileges to a specific user, the service fails and stops working.

### Central User Accounts

The removal of a user account from a directory service is rather simple and will only briefly be explained through the example of LDAP again. Other directory services may behave differently. Thus the actions have to be adopted to match the respective directory service in use.

LDAP provides the command `ldapdelete`[34] to remove leaves or entire subtrees. Similar to the removal of directories as explained in 6.3 – *Modifications Involving the File System*, removing nodes that are not leaves, i.e. *Organizational Units* (OUs), will result of the deletion of all nested nodes. Hence OUs must only be removed iff all nodes in its subtree are allowed to be removed.

[32]Man page `man 8 userdel`: `http://linux.die.net/man/8/userdel`
[33]On Ubuntu 14.04 `userdel` fails with an error like "`userdel:  user testuser is currently used by process 31775`".
[34]Man page `man 1 ldapdelete`: `http://linux.die.net/man/1/ldapdelete`

| Advantages | Disadvantages |
|---|---|
| **Local user accounts** | |
| • Easy to use<br>• Works out of the box<br>• All programs support them | • Not an option for larger environments<br>• Expensive maintenance with many users |
| **Local user accounts in distributed systems** | |
| • Works out of the box<br>• All programs support them | • No practical way to synchronize user settings across systems<br>• Easy to expose servers to unprivileged users<br>• Hard to maintain<br>• May result in collisions when merging networks |
| **Central user accounts in distributed systems** | |
| • No need for synchronization of user settings<br>• Scales for tens of thousands of users<br>• No collisions when merging networks | • Complex initial setup<br>• Overhead in maintenance and setup |

Table 10: Advantages and disadvantages of different kind of user management

## 6.3 Modifications Involving the File System

As outlined in chapters 5.1 – *Classification of IT Resources* and 5.2 – *Rules Derived From the Taxonomy* in section "Recoverability", there are three kinds of files:

1. Files created/tracked/updated through the system package manager or through resources installed through the system package manager
2. Files tracked through the configuration and system management tool
3. Files created/changed/deleted by a user

Following this classification, changing files that fall under (2) back to a previous state can be accomplished by updating the configuration and system management tool's configuration. On the next run, the respective files will be updated.

Changing files that are tracked through the system package manager (1) can of course be performed by using the configuration tool. Another, often more appropriate solution to roll back a system to a previous state, can involve a downgrade of the respective packages. Which, in turn, needs to follow the guidelines mentioned in 6.1 – *Installation, Update and Removal of Programs*.

User data, i.e. files that may or may not fall under (1) and all files that a user created or modified (3) need to be part of a backup before any changes to these files are made. This, in practice, is way more complex than any other case targeted in this thesis. While user data is normally kept in a user's home directory, users may have permissions to place files outside their home directory. As the existence of those files is unknown to a backup tool, unless coincidentally placed inside a folder that is backed up, there will not be a backup of those files.

As one can simply derive from those explanations, reversing files matching (1) or (2) is not a problem in general. Reversing user data is only possible when the files are placed inside a backed up location, otherwise user data is unrecoverable and thus not reversible.

To maintain the system stability from a user's perspective, a user must not be able to modify any files that are not backed up. Files that are not part of a backup cannot be required to safely and securely run a system. Furthermore, in order to preserve congruence of "identical" systems and not turn them into divergent systems, users must not be able to manually modify any files that are required to run the system and its services. In other words, all files that are required to keep a system running must either come from the system package manager or the configuration management tool.

Directories follow the same rules. There is one important consideration when rolling back a directory creation, i.e. removing a directory again: A directory must only be removed when it is empty and does not contain any files or other directories. The former requirement can be disregarded when a directory and contained files have been created in a single step. Removing that directory may also remove that particular file. However, recursively removing a directory is dangerous and can lead to severe data loss:

- Another file inside that directory that is not part of a backup would be removed
- Another directory inside that directory could be a mount point with a currently mounted file system. Recursively descending inside that directory would remove all files unless e.g. `--one-file-system` is given to `rm`[35]

Apart from the unintended data loss when removing files that are not recoverable, adding or changing files can easily result in a broken service setup. A common problem are syntax errors in those files. Some programs, e.g. Apache and Nginx, provide a tool to check their configuration for obvious errors[36]. On Nginx for example, linking to a non-existing `ssl_certificate_key` results in a "No such file or directory" error. If the Nginx service is restarted, the starting procedure would fail. An easy way to protect against this is making the configuration management tool check for the configuration test and revert the configuration file to its prior state and either raise a warning or bail out.

## 6.4 Network Setup Modifications

When modifying the network configuration of resources, one can easily not only break the setup of that particular resource – and may even loose control of it – but also render parts of the network unusable. This is especially true when adding and removing IP addresses from resources or changing routing configuration.

A countermeasure to prevent duplicate IP address assignments can be adding all IP addresses to a central database like the CMDB which will then be queried about the availability of a specific IP address before effectively adding the IP address to the server's configuration. This way addresses cannot be reused as long

[35]Man page `man 1 rm`: `http://linux.die.net/man/1/rm`
[36]For Apache the command is `/usr/sbin/apache2ctl configtest`; for Nginx the command is `/usr/bin/nginx -t`

as the CMDB has ACID behavior.

Adding an IP address is not necessarily reversible: if another resource depends on this system, removing the address would make the other resource fail and hence breaks the system stability. Therefore dependent systems must express their dependency on that host and IP address and an IP address must not be removed until all dependencies have been resolved.

Furthermore, the address removal process must not directly mark the IP address as unused. If it did, undoing the deleting can fail because the respective IP address is already in use by another system. It should rather mark the address as "to be deleted" for a grace period in which it may only be used when undoing the deletion process. There may also be services that depended on the address which have already been shut down. In case they are restarted they would fail.

This boils down to the fact that deallocating (not only identifying) resources must only be done once all resources that have or have had dependencies to it are not available anymore. A practical approach to accomplish this can be found on the flyingcircus.io blog: *Automatically deleting things – safely and reliably* [The15].

## 6.5 Ordering of Changes in Distributed Systems

Referring to the use case presented in chapter 5.3.5 – *Ordering of Changes in Distributed Systems*, there are two obvious ways to change the setup according to the requirements. Both approaches, however, result in a short period of unavailability of the server for the proxy.

There are two slightly more complex solutions to tackle the problem. They both require the proxy and server to gracefully reload their configuration without terminating current and rejecting new incoming requests:

1. Bind server to multiple ports
2. Redundant server setup

### 6.5.1 Bind Server to Multiple Ports

The idea behind this approach is to give the server multiple ports to listen on. That way the proxy can connect to both the old and the new port. The steps in order to achieve a downtime free change are:

1. Add the new port to the listening ports of the server
2. Gracefully reload the server process
3. Change proxy setup to connect to the new port
4. Gracefully reload the proxy service
5. Remove the old port from the listening ports of the server
6. Gracefully reload the server process

### 6.5.2 Redundant Server Setup

This approach offers a similar solution that is more appropriately for larger infrastructures where a proxy may also acts as a load balancer. It can also allow for using server services that do not offer the possibility to bind the service to multiple ports. It requires though that a subset of servers can be used to serve the service for a short time while the transition is in progress. For simplicity the servers are split into sets $s_1$ and $s_2$.

1. Remove the servers from $s_1$ from the proxy's configuration
2. Gracefully reload the proxy service
3. Change the port of the servers in $s_1$ to the new port (delete the old port)
4. Gracefully reload the server processes in $s_1$
5. Change proxy setup to exclude the hosts in $s_2$ (only connect to hosts in $s_1$)
6. Gracefully reload the proxy service
7. Change the port of the servers in $s_2$ to the new port (delete the old port)

8. Gracefully reload the server processes in $s_2$
9. Change proxy setup to include the hosts in $www_2$ again
10. Gracefully reload the proxy service

In case the page load cannot be handled by only $s_1$ or $s_2$, the web servers can be split into smaller groups. In that case the steps (5) - (8) need to be repeated with the sets $s_i$.

As one can see, the overall change performed in this example is reversible and does not require a specific explanation on how to roll it back. The reversibility of the change is due to all lower level changes (i.e. changing the configuration files, reloading the processes) being reversible. Other complex examples, however, are not necessarily reversible. This can be due to destructive operations such as removing unrecoverable files.

| Advantages | Disadvantages |
|---|---|
| **Bind Web Server to Multiple Ports** | |
| • Easy to accomplish<br>• Works only for single host systems | • Not an option for all services (e.g. PostgreSQL and MySQL do not support to bind to multiple ports) |
| **Redundant Web Server Setup** | |
| • Easy to accomplish<br>• Scales for large systems<br>• Might be an option for services not supporting binding to multiple ports | • Obviously not an option for single host systems |

Table 11: Advantages and disadvantages of approaches to maintain service uptime

# 7 Evaluation

This master's thesis first defined a taxonomy to classify resources in a modern IT environment and continued with rules derived from the classification that a system administrator should follow. Based on the methods covered in 6 – *Maintaining System Stability and Security*, one can conclude what a configuration and system management tool has to provide *to maintain system stability and security when reversing changes*: dependencies between resources.

In order to integrate automated configuration management into an existing or new IT environment one should follow these steps:

1. Identify the resources that are part of the environment. This includes but is not limited to the hardware components, software, accounts and account data.
2. Classify all resources according to the taxonomy described in chapter 5.1 – *Classification of IT Resources* on page 23. This helps to find requirements and limitations in the considered environment setup.
3. Work out the dependencies between the resources. What has to be done before something else? What requires another resource to be set up?

The following two case studies show examples of how Puppet and Ansible can be used to maintain a package installation status, files and directories, service status, among other aspects of a system.

## 7.1 Case Study: Puppet

The code for this Puppet case study of how to define dependencies between files, a package installation status and the status whether a service should be running or not, can be found in D – *Case Study: Puppet*. It is a simplified version taken from the production environment[37] at Flying Circus Internet Operations GmbH.

[37]Source: `https://bitbucket.org/flyingcircus/fc.platform/src/d8f01d8f41bf/puppet/modules/sys_cluster/manifests/`

The Puppet manifest describes the actions Puppet uses to take care of the Linux package "Consul".

What one can see in `sys_cluster/manifests/consul.pp` lines 8 to 10 are dependency definitions that make Puppet first follow the descriptions in `sys_cluster/manifests/consul/install.pp`, then takes care of the configuration by following `sys_cluster/manifests/consul/config.pp` and finally starting the service as defined in `sys_cluster/manifests/consul/service.pp`.

The file `config.pp` writes a file with an encryption key on the hard disk of the server (lines 12-15). By additionally specifying the "mode", Puppet complies with *Rule 4: Keep identifying resources secret* and effectively sets the access permissions to the file to be read-write for its owner and no access for anybody else. However, not specifying a file owner here does not pose a problem, since the system-wide default sets the owner and group to root. However, if there were not a default value, the file owner or group could be changed manually and Puppet would not change it back.

Both `install.pp` and `service.pp` inherit from `sys_cluster/manifests/consul/absent.pp` which defines the items Puppet should remove or deactivate. By defining the absence of the packages "consule" and "consulate", and the absence of the directories `/etc/consul.d` and `/var/lib/consul` (`absent.pp` lines 2-16) the Puppet manifest follows rules *Rule 9: Prune files and folders with package manager* and *Rule 10: Remove files managed by a configuration and system management tool*. In lines 18-21 of `absent.pp`, an explicit dependency for the service "consul" is defined to be disabled *before* the package "consul". This is an application of *Rule 7: Review depending resources*. Hence Puppet makes sure the service is not running before the installation itself. The `service.pp` file, on the other hand, revokes the dependency in line 6.

Without explicit dependencies between the "consul" service and the package for the installation and uninstallation part, Puppet may either try to start the service before it is installed or remove the package before the service is stopped.

When the trigger to restart the consul daemon in `service.pp` lines 9-11 was written, the administrator followed *Rule 3: Estimate downtime and check dependencies*. The consul service is set up as a clustered service, thus taking down a single node in the cluster is not a problem.

## 7.2 Case Study: Ansible

This case study shows how Ansible can be used to configure and manage the web server "Nginx". The configuration code is shown in E – *Case Study: Ansible* and was originally taken from the author's own server setup.

The first code snippet, `roles/nginx/tasks/main.yml`, is the role's main configuration and the entry point for Ansible. In lines 2-3, Ansible ensures that Nginx is installed on all target systems (the uninstallation is not part of the excerpt). As outlined in 6.2.1 – *Local User Accounts*, the creation of directories should be done by the parts of the configuration management tool that need these files / directories. Hence lines 8-12 create some specific folders used by the included (lines 13-14) declarations which will be explained in the next paragraphs. If Nginx is supposed to be enabled by default, it is activated in lines 15-16. Since Ansible processes all tasks chronologically, having the "autostart activation" at the end of the role ensures that Nginx is not started upon next reboot if the server crashes during the setup. Instead the server can be rebooted and Ansible can continue to set up the Nginx installation without interfering with an already running Nginx service.

The file `roles/nginx/tasks/vhosts.yml` takes care of the setup of all virtual hosts defined for a server. Prefixing the names of the configuration files with a number, as one can see in `hostvars/server.example.com/vars.yml` lines 18-21, ensures the configuration files are environment-wide unique (*Rule 6: Use environment-wide unique identifiers*). The tasks in lines 2-8 and 9-13 of `vhosts.yml` ensure the presence and absence of the virtual hosts defined in the variables `nginx_vhosts` and `nginx_vhosts_absent` respectively. The latter maintains *Rule 10: Remove files managed by a configuration and system management tool.* The remaining two tasks (lines 14-21 and 22-24) control the root directories for each virtual host. It is worth mentioning that as of Ansible version 1.9 there is no native way to prevent recursive deletions of directories. Therefore the task follows *Rule 10: Remove files managed by a configuration and system management tool*, but would violate *Rule 11: Check reversibility of resource changes and specify outcome* if the content of the variable `nginx_root_absent` is not defined manually.

In `roles/nginx/tasks/ssl.yml` lines (2-19), Ansible creates a fallback SSL / TLS certificate and key for the current host if none is present. This way Ansible does not reuse an identifying resource by accident (*Rule 2: Be careful when reusing limited resources*) and works in accordance with *Rule 1: Be careful when changing resource accessibility.* The remaining lines of the file define the creation of SSL / TLS certificates and keys from the content of the variable `nginx_ssl_data`. Furthermore, the permissions to the certificates and keys are set to read-only by root, following *Rule 4: Keep identifying resources secret.* The lines 29 and 40 additionally ensure that the content of the `nginx_ssl_data` variable does not show up in logs.

Although the actual revocation of the SSL / TLS certificates does not happen as part of the Ansible playbook, the removal of the certificates and private keys from the server are closely related to *Rule 5: Revoke identifying resources.*

Instead of running Nginx as root or a particular user, the `roles/nginx/files/nginx.conf` file binds the service to the user and group "http" (see *Rule 8: Bind production resources to role accounts*).

These case studies demonstrate that Ansible has a much coarser dependency granularity than Puppet. This is due to the sequential execution of the tasks inside the roles which implies that many dependencies are already resolved by ordering the tasks appropriately. Puppet, on the other hand, allows for much finer dependencies.

# 8 Conclusion and Prospects

This master's thesis shows that not carefully following the dependencies between resources can lead to severe damage not only to single resources but to the entire environment. A detailed example has been described by my colleague Christian Theune on the flyingcircus.io blog: *Automatically deleting things – safely and reliably* [The15].

Modern IT systems consist of hundreds and thousands of resources. Keeping all of them functioning in the way they are supposed to is a challenge administrators are facing every day. The usage of configuration management tools – regardless of whether they are agent-based or agent-less – helps them a lot. Using tools that help to maintain at least convergent systems is required. The risks of creating divergent systems by not using automated configuration and system management are ubiquitous. Divergent systems will inevitably fail sooner than later, possibly with data loss or a long system downtime.

Two of the initially mentioned configuration and system management tools – "CFEngine" and "Puppet" – provide an interface to specify dependencies between resources. In the "CFEngine" context they are called *Promises* [AS, ch 5.3f]. In "Puppet" environments they are *Metaparameters* and are used in the context of *Resource References and Ordering* [Lab].

"Ansible" only allows specifying dependencies between different roles [Ans15]. This can be enough when the granularity of the different roles is fine enough. Though most roles provide a self-contained set of tasks that are fully applied, dependencies can exist between those roles.

"Chef" does not allow any kind of dependencies which is justifiable due to its deterministic behavior [Aru12] as also argued by Traugott and Brown in [TB02].

Checklists like the "BSI-100 Standard" or process documentation specifications like ITIL are not feasible in large IT environments from an administrator's point of view, at least as long as they describe processes executed by people. The human error factor can lead to a company's bankruptcy as presented in "*Knightmare: A DevOps Cautionary Tale*" [Sev14]. Instead these standards provide solid starting points of how to reduce system faults.

Future development in the area of configuration and management systems should focus on easier, more administrator focused, integration of rollback strategies. Referring back to Mark Burgess' *Computer Immunology* [Bur98] idea, software components providing "self-healing" capabilities to IT systems would extensively help administrators *to maintain system stability*.

Although neither "Docker" nor "Rocket" nor "NixOS" are the focus of this thesis, it is worth looking at what role they can play in the future with respect to configuration and system management.

Docker and Rocket provide inherently congruent system environments. They can easily be used to deploy the same service a thousand times. The lack of reverting changes from the perspective of this thesis does not automatically make them more stable or secure. Although the changes to, for example, the `Dockerfile` are tracked in a version control system as described by Traugott and Huddleston [TH98, ch. 1], administrators still need to think about the consequences of e.g. removing a package or removing the access restrictions on a certain file.

NixOS introduces some innovative features with its functional dependency approach. Its usage is currently being evaluated at Flying Circus Internet Operations GmbH. From a system stability perspective, NixOS seems to be far ahead of aforementioned convergent and congruent management tools due to its atomic upgrades and reproducible system configurations.

# A. List of Abbreviations

| | |
|---|---|
| **ABI** | Application Binary Interface |
| **ACID** | Atomicity, Consistency, Isolation, Durability |
| **CEO** | Chief Executive Officer |
| **CIA** | Confidentiality, Integrity, Availability |
| **CMDB** | Configuration management database |
| **CRL** | Certificate Revocation List |
| **CTO** | Chief Technology Officer |
| **DHCP** | Dynamic Host Configuration Protocol |
| **DN** | Distinguished Name |
| **DNS** | Domain Name System |
| **IANA** | Internet Assigned Numbers Authority |
| **ITIL** | Information Technology Infrastructure Library |
| **I/O** | Input / Output |
| **LDAP** | Lightweight Directory Access Protocol |
| **LXC** | LinuX Container |
| **MTBF** | Mean Time Between Failure |
| **MTTR** | Mean Time To Repair |
| **NFS** | Network File System |
| **NTP** | Network Time Protocol |
| **OCSP** | Online Certificate Status Protocol |
| **OS** | Operating System |
| **OU** | Organizational Unit |
| **PAM** | Pluggable authentication module |
| **SAN** | Storage Area Network |
| **SSH** | Secure Shell |
| **SSL** | Secure Sockets Layer |
| **TLS** | Transport Layer Security |
| **WSGI** | Web Server Gateway Interface |

# B. List of Figures

# C. List of Tables

# D. Case Study: Puppet

```
class sys_cluster::consul {
    class { 'sys_cluster::consul::install': } ->
    class { 'sys_cluster::consul::config': } ~>
    class { 'sys_cluster::consul::service': }

    # work around dependency problem: Service['consul'] is originally
    # declared in install.pp and thus, it does logically not belong
    # to service.pp
    Class['sys_cluster::consul::config'] ~> Service['consul']
}
```

sys_cluster/manifests/consul.pp

```
class sys_cluster::consul::install inherits sys_cluster::consul::absent {
    Package['consul'] { ensure => present }

    Package['consulate'] { ensure => present }

    File['/etc/consul.d'] { ensure => directory }

    File['/var/lib/consul'] { ensure => directory }

    sys_portage::conffile { '/etc/init.d/consul': seen => '0.5.0' }

    sys_portage::conffile { '/etc/conf.d/consul': seen => '0.5.0' }
}
```

sys_cluster/manifests/consul/install.pp

```
class sys_cluster::consul::config {
    $encryption_key = consul_key()

    file { '/etc/conf.d/consul':
        content => template('sys_cluster/consul/conf.d_consul.erb')
    }

    file { '/etc/consul.d/00basic.json':
        content => template('sys_cluster/consul/basic.json.erb')
    }

    file { '/etc/consul.d/01encryption.json':
        content => '{"encrypt": "${encryption_key}"}\n',
        mode => 0600,
    }

    file { '/etc/local/consul': ensure => directory }
}
```

sys_cluster/manifests/consul/config.pp

```
class sys_cluster::consul::service inherits sys_cluster::consul::absent {
    include sys_process::restarter

    Service['consul'] {
        provider => 'gentoo', enable => true,
        ensure => true, before => undef,
    }

    sys_process::restarter::service { '/run/consul.pid':
        restart => '/etc/init.d/consul restart',
    }
}
```

sys_cluster/manifests/consul/service.pp

```
class sys_cluster::consul::absent {
    package { 'consul':
        category => 'sys-cluster', ensure => absent,
    }

    package { 'consulate':
        category => 'dev-python', ensure => absent,
    }

    file { '/etc/consul.d':
        ensure => absent, force => true,
    }

    file { '/var/lib/consul':
        ensure => absent, force => true,
    }

    service { 'consul':
        enable => false, ensure => false,
        provider => gentoo_absent, before => Package['consul'],
    }
}
```

sys_cluster/manifests/consul/absent.pp

# E. Case Study: Ansible

```
---
- name: Install Nginx
  pacman: name=nginx state=present
- name: Set base configuration
  copy: src="nginx.conf" dest="/etc/nginx/nginx.conf"
  notify:
    - restart nginx
- name: Create directories
  file: path="/etc/nginx/{{ item }}" state=directory
  with_items:
    - sites
    - ssl
- include: vhosts.yml
- include: ssl.yml
- name: Enable service
  service: name=nginx enabled={{ nginx_enabled }}
```

roles/nginx/tasks/main.yml

```
---
- name: Create vhosts
  copy:
    src: "vhosts/{{ item }}"
    dest: "/etc/nginx/sites/{{ item }}"
  with_items: nginx_vhosts
  notify:
    - restart nginx
- name: Remove old vhosts
  file: dest="/etc/nginx/sites/{{ item }}" state=absent
  with_items: nginx_vhosts_absent
  notify:
    - restart nginx
- name: Create vhost roots
  file:
    dest: "{{ item.path }}"
    owner: "{{ item.owner|default('root') }}"
    group: "{{ item.group|default('root') }}"
    mode: "{{ item.mode|default('0755') }}"
    state: directory
  with_items: nginx_root
- name: Remove old vhost roots
  file: dest="{{ item.path }}" state=absent
  with_items: nginx_root_absent
```

roles/nginx/tasks/vhosts.yml

```
---
- name: Generate master key
  command: openssl genrsa -out /etc/nginx/ssl/local.key 1024
           creates="/etc/nginx/ssl/local.key"
  notify:
    - restart nginx
- name: Set master key permissions
  file: dest="/etc/nginx/ssl/local.key" owner=root group=root mode=0400
- name: Generate master certificate
  command: openssl req -new -nodes -x509 -subj
           "/C=DE/ST=Berlin/L=Berlin/O=private/CN=localhost"
           -days 3650 -key /etc/nginx/ssl/local.key -out
           /etc/nginx/ssl/local.crt -extensions v3_ca
           creates="/etc/nginx/ssl/local.crt"
  notify:
    - restart nginx
- name: Set master certificate permissions
  file: dest=/etc/nginx/ssl/local.crt owner=root group=root mode=0400
- name: Add SSL certificates
  file:
    dest: "/etc/nginx/ssl/{{ item.key }}.crt"
    content: "{{ item.value.crt }}"
    owner: root
    group: root
    mode: 0400
  with_dict: nginx_ssl_data
  notify:
    - restart nginx
  no_log: true
- name: Add SSL keys
  file:
    dest: "/etc/nginx/ssl/{{ item.key }}.key"
```

```
      content: "{{ item.value.key }}"
      owner: root
      group: root
      mode: 0400
    with_dict: nginx_ssl_data
    notify:
      - restart nginx
    no_log: true
  - name: Remove old SSL certificates
    file: dest="/etc/nginx/ssl/{{ item }}.crt" state=absent
    with_items: nginx_ssl_absent
    notify:
      - restart nginx
  - name: Remove old SSL keys
    file: dest="/etc/nginx/ssl/{{ item }}.key" state=absent
    with_items: nginx_ssl_absent
    notify:
      - restart nginx
```

roles/nginx/tasks/ssl.yml

```
---
- name: restart nginx
  service: name=nginx state=restarted
```

roles/nginx/handlers/main.yml

```
---
nginx_root:
  - path: /srv/http/example.com
    owner: http
    group: http
nginx_root_absent:
  - path: /srv/http/staging.example.com
nginx_ssl_data:
  example.com:
    crt: |
      -----BEGIN CERTIFICATE-----
      Domain and intermediate certificates data
      -----END CERTIFICATE-----
    key: |
      -----BEGIN RSA PRIVATE KEY-----
      Private key data for certificate above
      -----END RSA PRIVATE KEY-----
nginx_vhosts:
  - 01-example.com.conf
nginx_vhosts_absent:
  - 02-staging.example.com.conf
```

host_vars/server.example.com/vars.yml

```
user  http  http;
worker_processes  2;

error_log  /var/log/nginx/error.log;

events {
    worker_connections  1024;
}

http {
    include  mime.types;
    default_type  application/octet-stream;
    access_log  /var/log/nginx/access.log;

    ssl_ciphers  ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256: \
                 ECDH+AES128:DH+AES:ECDH+3DES:DH+3DES:RSA+AESGCM: \
                 RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS;
    ssl_prefer_server_ciphers  on;
    ssl_protocols  TLSv1 TLSv1.1 TLSv1.2;
    ssl_session_cache  shared:SSL:50m;
    ssl_session_timeout  5m;

    server {
        listen  [::]:80 default_server ipv6only=off;
        listen  [::]:443 default_server ipv6only=off ssl;
        server_name  localhost;
        error_page  500 502 503 504  /50x.html;

        ssl_certificate  /etc/nginx/ssl/local.crt;
        ssl_certificate_key  /etc/nginx/ssl/local.key;

        location = /50x.html {
            root  /usr/share/nginx/html;
        }

        location / {
            root  /usr/share/nginx/html;
            index  index.html index.htm;
        }
    }

    include  /etc/nginx/sites/*.conf;
}
```

roles/nginx/files/nginx.conf

```
server {
    listen  [::]:443 ssl;
    server_name  example.com;
    add_header  Strict-Transport-Security  "max-age=31536000; preload";

    ssl_certificate  /etc/nginx/ssl/example.com.crt;
    ssl_certificate_key  /etc/nginx/ssl/example.com.key;

    location / {
        root  /srv/http/example.com/;
        index  index.html;
    }
}
```

roles/nginx/files/vhosts/01-example.com.conf

# F. Glossary

**APT** Advanced Packaging Tool – A system package manager on Debian based Operating Systems (OSs) that uses dpkg.

**atime** The time of the last *a*ccess of a file's content.

**Configuration File** A file that a service on a system uses to retrieve and store settings in.

**ctime** The time of the last *c*hange of a file's attributes.

**DEB** The file format used by APT and dpkg.

**DN** Distinguished Name – The full, unique name of an item in LDAP.

**dpkg** The base for Debian package managers.

**Home Directory** A directory on a file system that is only usable by its owner.

**LDAP** Lightweight Directory Access Protocol – A directory protocol to handle identities in an IT environment.

**MTBF** Mean Time Between Failure – The time a system runs correctly between two failures.

**mtime** The time of the last *m*odification of a file's content.

**MTTR** Mean Time To Repair – The time a system is faulty between to successful periods.

**NTP** Network Time Protocol – A protocol to synchronize the global time across the Internet..

**OU** Organizational Unit – The groups a user is part of in LDAP.

**PAM** Pluggable authentication module – An API to locally authenticate a user.

**POSIX** A standard to maintain interoperability between OSs.

**Process ID** Each time a process starts on UNIX systems they get a, to that time, unused ID.

**RPM** RPM Package Manager, mainly used on RedHat based OSs.

**SSH** Secure Shell – Protocol and program to login to remote systems.

**SSL / TLS** Secure Sockets Layer / Transport Layer Security – Protocols to provide confidentiality and integrity of data.

**User ID** A system-wide unique identifier for a user.

# G. References

[And94] Paul Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th USENIX Conference on System Administration*, LISA '94, Boston, MA, USA, 1994. USENIX Association. `https://www.usenix.org/legacy/publications/library/proceedings/lisa94/full_papers/anderson.a`, Visited May 8, 2015.

[Ans15] Inc. Ansible. Playbook roles and include statements, April 2015. `http://docs.ansible.com/playbooks_roles.html#role-dependencies`, Visited Apr 27, 2015.

[Arc15] Arch Linux Wiki. Pacman rosetta, April 2015. `https://wiki.archlinux.org/index.php?title=Pacman_Rosetta&oldid=366964`, Visited Apr 4, 2015.

[Aru12] John Arundel. Puppet versus chef: 10 reasons why puppet wins, December 2012. `http://bitfieldconsulting.com/puppet-vs-chef`, Visited Apr 27, 2015.

[AS] CFEngine AS. Cfengine 3 concept guide. `https://auth.cfengine.com/archive/manuals/cf3-conceptguide`, Visited Apr 27, 2015.

[Avr13] Abel Avram. Docker: Automated and consistent software deployments. March 2013. `http://www.infoq.com/news/2013/03/Docker`, Visited May 12, 2015.

[BC11] Mark Burgess and Alva Couch. On system rollback and totalized fields: An algebraic approach to system change. *The Journal of Logic and Algebraic Programming*, 80(8):427 – 443, 2011.

[Bur] Mark burgess biographical information. `http://markburgess.org/bio.html`, Visited Jan 24, 2015.

[Bur93] Mark Burgess. Cfengine v2.0: A network configuration tool, September 1993. `http://www.iu.hio.no/~mark/papers/cfengine_history.pdf`, Visited Apr 27, 2015.

[Bur95] Mark Burgess. A site configuration engine, 1995. `https://www.usenix.org/legacy/publications/compsystems/1995/sum_burgess.pdf`, Visited May 8, 2015.

[Bur98] Mark Burgess. Computer immunology. In *Proceedings of the 12th USENIX Conference on System Administration*, LISA '98, pages 283–298, Berkeley, CA, USA, 1998. USENIX Association. https://www.usenix.org/legacy/publications/library/proceedings/lisa98/full_papers/burgess/burgess.pdf, Visited Jan 18, 2015.

[Dai04] John Daintith. *A Dictionary of Computing.* Oxford University Press, 2004. http://www.encyclopedia.com/doc/1O11-systemsecurity.html, Visited Mar 27, 2015.

[Dja15] Django Software Foundation. Migrations in django, 03 2015. https://docs.djangoproject.com/en/dev/topics/migrations/, Visited Mar 13, 2015.

[Fel13] Anja Feldmann. Lecture: Internet/computer security, summer term 2013, 2013. Part I: General – Terminology.

[Fow12] Martin Fowler. Snowflakeserver, July 2012. http://martinfowler.com/bliki/SnowflakeServer.html, Visited May 8, 2015.

[Gan95] Mike Gancarz. *The UNIX Philosophy.* Digital Press, Newton, MA, USA, 1995.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Databases*, Cupertino, CA, USA, June 1981. Tandem Computers. http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf, Visited May 6, 2015.

[Gra85] Jim Gray. Why do computers stop and what can be done about it. Technical Report 85.7, Tandem Computers, June 1985. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf, Visited May 6, 2015.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[IET14] IETF. Iana ipv4 special-purpose address registry, December 2014. http://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml, Visited Jan 20, 2015.

[Lab] Puppet Labs. Learning puppet — resource ordering. https://docs.puppetlabs.com/learning/ordering.html, Visited Apr 27, 2015.

[PFBH14] Manuel Pais, Chad Fowler, Mark Burgess, and Mitchell Hashimoto. Virtual panel on immutable infrastructure. March 2014. http://www.

infoq.com/articles/virtual-panel-immutable-infrastructure, Visited Jan 18, 2015.

[PW] Tom Preston-Werner. Semantic versioning 2.0.0. http://semver.org/spec/v2.0.0.html, Visited Feb 19, 2015.

[Rob09] Jesse Robbins. Announcing chef, January 2009. https://www.getchef.com/blog/2009/01/15/announcing-chef/, Visited Jan 24, 2015.

[Sev14] Doug Seven. Knightmare: A devops cautionary tale, April 2014. http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/, Visited May 8, 2015.

[Spi07] Diomidis Spinellis. *"Another level of indirection" in "Beautiful Code: Leading Programmers Explain How They Think"*, chapter 17, pages 279–291. O'Reilly and Associates, 2007.

[TB02] Stephen Gordon Traugott and Lance Brown. Why order matters: Turing equivalence in automated systems administration. In *Proceedings of the 16th USENIX Conference on System Administration*, LISA '02, Philadelphia, PA, USA, 2002. USENIX Association. http://www.infrastructures.org/papers/turing/turing.html, Visited Jan 18, 2015.

[TH98] Steve Traugott and Joel Huddleston. Bootstrapping an infrastructure. In *Proceedings of the 12th USENIX Conference on System Administration*, LISA '98, Boston, MA, USA, 1998. USENIX Association. http://www.infrastructures.org/papers/bootstrap/bootstrap.html, Visited May 8, 2015.

[The14a] The Docker Team. Illustration of the interfaces docker uses to access virtualization features of the linux kernel, March 2014. http://blog.docker.com/wp-content/uploads/2014/03/docker-execdriver-diagram.png, Visited May 6, 2015.

[The14b] Christian Theune. Flying circus rca report #13266, March 2014. http://flyingcircus.io/postmortems/13266.pdf, Visited Apr 27, 2015.

[The15] Christian Theune. Automatically deleting things – safely and reliably, April 2015. http://blog.flyingcircus.io/2015/04/24/automatically-deleting-things-safely-and-reliably/, Visited Apr 27, 2015.