Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Lauri Tirkkonen

Utilising configuration management node data for network infrastructure management

Master's Thesis
Espoo, May 4, 2016

Supervisor:	Professor Heikki Saikkonen
Advisor:	Jaakko Kotimäki M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Lauri Tirkkonen

Title:
Utilising configuration management node data for network infrastructure management

Date:	May 4, 2016	**Pages:**	38
Major:	Software Technology	**Code:**	T-106
Supervisor:	Professor Heikki Saikkonen		
Advisor:	Jaakko Kotimäki M.Sc. (Tech.)		

Configuration management software running on nodes solves problems such as configuration drift on the nodes themselves, but the necessary node configuration data can also be utilised in managing network infrastructure, for example to reduce configuration errors by facilitating node life cycle management. Many configuration management software systems depend on a working network, but we can utilise the data to create large parts of the network infrastructure configuration itself using node data from the configuration management system before the nodes themselves are provisioned, as well as remove obsolete configuration as nodes are decommissioned.

Keywords: configuration management, network infrastructure configuration, life cycle management

Language: English

Aalto-yliopisto
Perustieteiden
korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Lauri Tirkkonen		
Työn nimi:			
Konfiguraationhallinnan datan käyttö verkkoinfrastruktuurin hallintaan			
Päiväys:	4. toukokuuta 2016	**Sivumäärä:**	38
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	Diplomi-insinööri Jaakko Kotimäki		

Konfiguraationhallintajärjestelmien käyttö ratkaisee tietoliikenneverkon solmuilla (node) esiintyviä ongelmia kuten konfiguraation ajelehtimista, mutta konfiguraationhallintaan vaadittua tietovarastoa voidaan käyttää myös verkkoinfrastruktuurin hallinnassa, esimerkiksi vähentämään konfiguraatiovirheitä helpottamalla solmujen elinkaaren hallintaa. Useat konfiguraationhallintaohjelmistot vaativat toimivan verkon, mutta suuria osia verkkoinfrastruktuurin konfiguraatiosta voidaan luoda käyttäen konfiguraatiohallinnan tietovarastoa ennen kuin solmuja pystytetään, sekä voidaan varmistaa vanhentuneen konfiguraation poistuminen solmuja alas ajattaessa.

Asiasanat:	konfiguraationhalilnta, verkkoinfrastruktuurin hallinta, elinkaaren hallinta
Kieli:	Englanti

Acknowledgements

I would like to thank my manager and colleagues for bearing with me while I implemented and deployed the software presented in thesis into our production systems, and all of my friends and family for support during my studies.

Espoo, May 4, 2016

Lauri Tirkkonen

Abbreviations and Acronyms

API	Application Programming Interface
CARP	Common Address Redundancy Protocol
CIDR	Classless Inter-Domain Routing
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
IP	Internet Protocol
NDP	Neighbor Discovery Protocol
NTP	Network Time Protocol
RA	Router Advertisement (in NDP)
RR	Resource Record (in DNS)
SLAAC	Stateless Address Autoconfiguration
TCP	Transport Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual LAN
YAML	YAML Ain't Markup Language

Contents

Chapter 1

Introduction

This thesis is an attempt to improve the way in which network infrastructure configuration is managed at the Aalto University Department of Computer Science by utilising a data store normally used for configuration management software. In the first chapter, we present the research problem and the constraints which a solution should meet, as well as describe the structure of the rest of the thesis.

1.1 Problem statement

Managing network infrastructure configuration can quickly get cumbersome due to the high node churn in today's environments: virtual machines, containers or other nodes are provisioned and decommissioned regularly. Even with a simple layer 3 infrastructure, a newly provisioned node needs at least network addresses, default router and name server configuration, firewall rules and DNS records. There are existing solutions to some of these problems (eg. addressing, routing and name service through DHCP), but others require explicit configuration about the new node in the corresponding infrastructure service: name-to-address mappings in DNS records, address-to-rule mappings in firewall configuration, link layer-to-network layer address mapping in DHCP reservations, and so on.

To reduce the network administrators' maintenance burden and the configuration error rate, we should reduce redundancy in the node provisioning process as well as facilitate removing configuration specific to a node when it comes to the end of its life cycle. Additionally, in the interests of being able to understand and debug the system, it should interoperate with, not supersede, our current network configuration management solutions.

This general idea is quite common; for example, Carpenter and Jiang [5, Section 2.3] recommend usage of a “suitable configuration tool” to prevent drift between DNS records and DHCPv6 configuration.

We propose an authoritative repository-of-record for all of the network configuration metadata for a majority of nodes: link layer addresses, network layer addresses, firewall rules and DNS names. A natural place to store this information is the same data store we use for the configuration management data specific to each node, so that a single entry in that database fully describes a node. Having this data in one place, we can automatically generate and deploy the requisite network infrastructure configurations. However, we recognize that not every node fits one network configuration pattern, and thus our solution must be able to coexist with existing infrastructure instead of replacing it.

1.2 Requirements for a solution

Freeman [10] describes requirements that apply to network configuration management systems. While they are talking about managing routers in the context of service providers, many of the same principles do apply to managing configuration for software necessary in an IP network. For example: our system must have configuration error tolerance, so that a mistake does not affect the entire network; we should consider our system the authoritative source of truth for the desired state of the network; and the system should protect the network against misconfigurations as well as validate that the current state of the network matches the configuration (ie. the desired state).

In addition to the above, we have some local requirements. Most importantly, the system must support versioning and rollback, also of configuration changes that are not the most recent. Additionally, the system must not replace the existing network infrastructure management systems in place at our site but instead coexist with them.

1.3 Structure of the Thesis

This thesis is split into separate chapters, described below.

1.3.1 Background

In the Background chapter, we take a look at the concepts this thesis is built on, including configuration management systems and some network configuration basics. Additionally, we take a cursory look at some existing software solutions to managing network infrastructure configuration and discuss their suitability to the problem at hand.

1.3.2 Environment

In this chapter, we will take a look at the network infrastructure and previously used tools in the environment where we integrated our solution.

1.3.3 Implementation

We discuss the operating theory and technical details of our solution as it was implemented as well as provide examples of its function in the fourth chapter.

1.3.4 Evaluation

In this chapter, we discuss the degree to which our goals were met by the implemented software and present brief analyses of certain aspects of its performance.

1.3.5 Conclusions

In the final chapter, we present the conclusions reached by the work we've done and present some opportunities for future work.

Chapter 2

Background

In this chapter, we describe some of the systems and concepts necessary to understand the rest of the thesis, including some basic network infrastructure as well as configuration management software, and present some existing potential solutions to our problem and discuss their suitability.

2.1 Network infrastructure configuration

In the context of this thesis, when we say ”network infrastructure configuration”, we mean the software configuration and data files necessary for the operation and management of computer networks. More specifically we are interested in configurations concerning protocols operating on the data link, network and transport layers in the Open Systems Interconnection model [12]: we wish to configure name services provided by the Domain Name System (DNS), IP address assignment with the Dynamic Host Configuration Protocol (DHCP) as well as packet filtering based on IP or transport layer (eg. TCP/UDP) packet headers, ie. firewalling. We will describe the different types of configurations in more detail below.

2.1.1 DNS

DNS is a well-known distributed system that is used, among other things, to map host names to IP addresses and vice versa. It forms a global tree hierarchy of *zones*, each of which may contain *resource records* (RRs) that contain information about a domain name, as well as contain references to sub-zones (*delegations*). The *root zone* has an empty label and is thus simply called `.` (dot). It is defined by a well-known set of servers known as the *root name servers*, and it contains sub-zones like `com.` and `org.`, which are delegated to name servers other than the root name servers, and those are further split into sub-zones using the same delegation mechanism. [14] Generally organisations purchase a delegation for a second-level domain such as `example.com.` and run their own name servers which serve the resource records for that zone (eg. information about their web server under the domain name `www.example.com.`).

The configuration we are interested in is the configuration for the DNS server software that we run to serve the resource records in the zones which are delegated to us, ie. the zones we are authoritative for, as well as the *zone files* which contain the RRs themselves.

2.1.2 DHCP

DHCP is a protocol that used on IP networks to automatically assign IP addresses and other network configuration parameters, such as the default gateway and addresses of available recursive DNS resolvers, to devices on the network. Addresses may be allocated from a predetermined pool of addresses by the server software (dynamic allocation) or an administrator may assign each client's address (manual allocation). [9] Managing manual allocations, that is, link-layer address to assigned IP address mappings, is the configuration that we are interested in in the context of this thesis: each host should have stable addresses assigned to it so that we may use those addresses in DNS RRs and packet filter rules to refer to the correct host.

2.1.3 Packet filtering

Packet filtering, or firewalling, is a security mechanism that entails restricting the IP communications of hosts by blocking the delivery of packets to their destination based on some criteria defined by the administrator. Many network devices ranging from endpoints to routers have packet filtering features, but in this thesis we consider the filtering rules on the gateway routers on our networks and leave aside endpoints' own packet filters. We wish to use a "default deny" policy for communications, but be able to easily allow certain packets through the firewall to hosts in our networks: this means we need to be able to manage filter rules for incoming traffic based on the IP Destination header field, but for most cases we also need to filter on the Source field as well as information in protocol headers on higher layers of the OSI model, eg. TCP destination port.

2.2 Configuration management in system administration

In system administration, configuration management refers to the process of ensuring that a set of networked hosts are configured in the way that the administrator intends. Cfengine, presented by Burgess and College [4] in 1995, was the first published software tool intended to automate configuration tasks and prevent configuration drift, ie. ensure that configuration stays correct.

A large motivation for configuration management software has always been the increasing number of systems that administrators need to maintain. With the rise of cloud computing, virtualisation and containerisation technologies, this need has been emphasized even further: each virtual machine or operating system container is its own system, ie. runs its own software stack and network stack, which are separate from the host and other virtual machines or containers, and need to be managed separately.

In the context of cloud computing, many different people have characterized a difference in patterns for managing servers as "pets vs. cattle"; it's not completely clear where this term originates, but Bias [3, slide 20] attributes it to Microsoft's Bill Baker. The idea is that traditionally system administrators have kept their servers like pets, giving them names and carefully nurturing them back to health whenever they get sick, while in a cloud environment, servers are expendable and replaceable, like cattle. While servers falling under both patterns do actually benefit from using configuration management software, it is completely infeasible to have cattle without configuration management.

Due to these reasons, it is now increasingly rare that a system administrator logs into a particular pet of a server to make a configuration change with a text editor; instead many of his machines are now running configuration management software that tracks changes made into a configuration management repository and applies them to the machines.

Of course, in addition to configuration for its software stack, each virtual machine or operating system container (*host*) needs to be supported by the network infrastructure. A host may have multiple network interfaces, perhaps attached to different networks, but usually needs at least one. A host connected to a network is a *node* in that network. Since a host with a single network interface connected to a single network is one node, the terms are often used interchangeably.

If we assume the simple case where each node is a host, the amount of required network infrastructure configuration is similar to the amount of configuration required for the hosts' software stacks. Additionally, since hosts are provisioned and decommissioned often in virtualisation and containerisation environments, the node churn is high. This underscores the need to manage network infrastructure configuration with software, applying principles similar to those used in managing the configuration on the hosts themselves.

2.3 Existing solutions

Of course, our need for a solution to help manage the configuration of network infrastructure daemons with respect to each other and to node life cycles is not unique, and there are existing tools and solutions that can help with the problem. We take a look at a few and the specific problems they solve below, keeping in mind our own requirements and evaluating the solutions with respect to them.

2.3.1 Dynamic DNS updates

Instead of keeping resource records for all domain names in static zone files, there is a mechanism that may be used to automatically add records for names called DNS Update, proposed by Vixie et al. [16]. It is usually performed by the DHCP server in a configuration where there is a dynamic pool of addresses that may be assigned to DHCP clients: when the server assigns an address to a client for a duration (ie. gives the client a *lease* on the address), it will also request the DNS server to add a type A resource record for a domain name that is, for example, derived from the host name the client sent in a DHCP option and a configured domain suffix. This way, when a host joins the network, it will automatically be assigned a DNS name that can be used to connect to it. DHCP servers that implement this feature often refer to it as "Dynamic DNS update" or "DDNS update".

However, Dynamic DNS updates with ISC BIND are normally configured for only certain zones, which are then usually not managed by other means; one has to disable dynamic updates for a zone to modify the zone files manually [6, Section 4.2.1], and our current approach, detailed in the next chapter, of always generating all of our zone files from source data certainly presents a problem in that scenario. We could use a specific sub-zone that is dynamic-enabled and not edit that through our existing tooling, but that would mean the FQDNs of the managed nodes changing and becoming longer, which we do not want. The source of truth for the zone contents also becomes less clear when using dynamic updates: the current state may be queried from BIND, but it becomes more difficult to tell why a certain record holds a certain value.

2.3.2 `dhcpctl` in ISC DHCP

The ISC DHCP server software has a feature called `dhcpctl`, which may be used to query and modify the configuration and state of the DHCP server daemon at runtime. It is implemented on top of OMAPI (Object Management Application Programming Interface), which uses TCP to communicate with the server implementing the objects to manage. [7, 8]

We could conceivably use `dhcpctl` to add and remove host objects to the running DHCP server when adding nodes to our network, but there are similar problems in this approach as there are with Dynamic DNS updates - the source of truth is no longer our version controlled data, because the runtime configuration state can change by other means.

2.3.3 Foreman

Foreman is a web application that provides many features related to server life cycle management, eg. automatic provisioning of virtual machines, including creating DNS records and DHCP reservations. Further, it integrates with several commonly used configuration management tools: for example, it can process and store reports generated by Puppet agent runs as well as provide node classification (the process of selecting which *classes* of configuration should apply to each node). To manage the DNS zones and DHCP configuration, it has components called Smart Proxies that translate HTTP requests, initiated by the administrator using the web interface, to the actual operations on different components of network infrastructure. For ISC BIND and DHCP, the Smart Proxy implementations actually use Dynamic DNS Updates and `dhcpctl` (described above); the idea is that the web application sends a HTTP request to the Smart Proxy, which then modifies the state of the desired service through APIs specific to that service.

It is quite close to an acceptable solution to manage our infrastructure, but it fails on several key requirements we outlined in chapter 1. To use it for DNS record management, we would have to make it the sole authority for the zones it's in use with due to the problems with Dynamic DNS Updates described above, meaning that it does not coexist with our existing infrastructure. The DHCP Smart Proxy has a similar issue. Additionally, to actually simplify the management of our nodes and consolidate the configuration into one place, we would have to use the node classification features as well. Otherwise, decommissioning a node would require removing it from our current node classification data store as well as Foreman.

Lastly, even if we were to replace large parts of our existing network infrastructure management tools with Foreman, using its database as the sole source of truth for our network configuration, we would not get version control and rollback for that configuration (as of Foreman version 1.11), and are thus forced to conclude that Foreman is, in our environment, an unsuitable solution for the problem this thesis attempts to solve.

Chapter 3

Environment

In this chapter, we describe the environment where we integrated our solution: its network infrastructure as well as tools and methods used to manage it.

Layer 3 infrastructure configuration is managed using methods similar to the "Generate Everything" approach described by Schönwälder et al. [15]. Our solution will not change that, but consolidates the configuration for certain types of nodes into one place.

3.1 Usage patterns

Since our environment is the computer science department in a university, a common use-case is that the system administrators provide application platforms for research groups, projects and the like. Most often, this means a guest virtual machine with an operating system installed and co-maintained by the system administration, to which the customer has root access, but there are exceptions as well. These virtual machines all need network infrastructure configuration to function, but of particular interest is their life cycle. The applications rarely need to run indefinitely, so to conserve resources and avoid potential security issues, requests for such a platform come with an expiration date, after which the platform is to be decommissioned. However, since it is difficult to predict the future, this date is rarely correct, and when it arrives, the customer must be consulted again as to whether they still need it. Automating this process would save a considerable amount of time and effort.

3.2 Network segmentation

In our environment, there are 17 different link-layer (Ethernet) networks. Each link-layer network has exactly one corresponding network layer address block per network layer protocol, ie. one for IPv4 and one for IPv6. The networks are segregated using VLAN tagging on switches and routers.

3.3 Routing and firewall

Each host on every network has a default router on the same link, configured on clients through DHCP for IPv4 and NDP RAs for IPv6. This default router is at an address shared by two firewall machines using CARP for redundancy. For further fault tolerance, each shared address is on a VLAN tagged network link over two LACP-aggregated 10GbE links.

The router machines run OpenBSD and provide traffic shaping and firewalling through `pf`, a stateful packet filter shipped with OpenBSD. We have two machines for high availability reasons: firewall states are synchronized between the machines with `pfsync`, and CARP is used to manage which machine has the default gateway addresses on the network, thus providing automatic failover.

In addition the routers also implement DHCP relaying for DHCPv4, which is described below.

3.4 Address assignment

The general address assignment policy for our networks is that unless absolutely necessary, statically configured addresses are not used. Instead, we utilise link-layer addresses to generate or reserve persistent IP addresses for our nodes.

For IPv4, we have one DHCP server, which is only connected to one network. This means that we need relay servers in every other network since DHCP requires link-layer broadcast messages to function. The address assignment policy means that each node connecting to the network must have its MAC address in the DHCP server configuration, mapped to an address allocated for that node. The largest exceptions to this are workstation or ”office” networks, which need to support devices other than those managed by system administration, eg. BYOD (Bring Your Own Device).

For IPv6, we use stateless address autoconfiguration (SLAAC); the router machines advertise a specific /64 CIDR prefix on each link. We do not currently use DHCPv6, instead opting to fix DNS names to SLAAC addresses based on MACs or statically configuring addresses where necessary (for example, for failover scenarios).

3.5 Name service

Despite having seventeen networks, the environment only has two DNS name servers: one authoritative for all delegated zones (including reverse zones), and the other a slave for the same zones. These servers also provide a recursive resolver for hosts in our networks.

The DNS server software we use is ISC BIND. To configure it, there's an old in-house software solution which generates zone files from its own configuration format. This kind of solution is common and makes DNS easier to get right: for example, it automates generation of reverse (ie. PTR) records for all configured hosts and automatically updates SOA serial numbers.

3.6 Configuration management

The configuration management software system in use at our site is Puppet. With Puppet, we use Hiera, a hierarchical data store, as the storage for configuration data: while Puppet manifest files describe how to accomplish a certain configuration (ie. they are code, in the Puppet language), Hiera provides the parameters for that configuration (ie. it is data). Hiera contains information necessary to classify nodes (ie. assign Puppet classes to them) as well as customise certain aspects of configuration created by Puppet on the nodes. For example, we have a Puppet class which manages configuration for `ntpd`, the Network Time Protocol (NTP) daemon, on each node. The class is generalised such that we can apply it to every node, but modify the set of NTP servers added as peers by changing the Hiera data on a per-node or per-domain basis, based on the lookup hierarchy.

Hiera is able to use many different storage backends for its data, but because of the history and searchability features offered by version control systems as well as our administrators' familiarity with Unix tools for text manipulation, we are using a version controlled repository of YAML files as the backend.

3.7 Continuous integration

Currently, our network infrastructure configuration lives in version control repositories, separated by configuration type: we have one repository for DNS resource records and name server configuration, one for DHCP address assignments and DHCP server configuration, and one for firewall rules. The version control software provides us with the ability to roll back each individual change and helps to understand the existing configuration (eg. one can use 'blame' to help in figuring out why some configuration line exists).

To deploy configuration changes made by administrators into production, we use simple `git` hooks as a continuous integration tool. When an administrator pushes a change to the `master` branch of a network infrastructure repository, new configuration is built and syntax-checked by the system, and if the resulting configuration seems valid, it is automatically deployed to the production machine(s) running the corresponding service. This process is described in more detail in subsections for each infrastructure configuration type below.

It is evident that when using a system like this to manage network infrastructure configuration, provisioning a new node (or removing an old one) is an operation that requires many distinct configuration changes: assign IP addresses, create DNS resource records for them, add DHCP reservations and create new firewall rules. Our system aims to make node provisioning and decommissioning simpler for the most common use cases, while still keeping the flexibility of separate repositories to facilitate configuration for more specialized use cases.

3.7.1 DHCP repository

As described above, our address assignment policy necessitates link-layer (MAC) address to IPv4 address mappings in the DHCP server configuration. The configuration lives in a `git` repository, and is edited by administrators on their workstations. The configuration is split into two files: a base configuration file that defines basic operating parameters such as networks, domain names and address pools, and a supplementary file containing the IPv4 address to MAC mappings in a custom format: each line contains one one MAC address and one IPv4 address, separated by whitespace.

When an administrator pushes a change to the central `git` server, a `post-receive` hook executes to mirror the change the production DHCP server machine, using another push. The production machine executes a different `post-receive` hook, which transforms the IPv4-MAC -mappings into the host definition format expected by the DHCP server software (using an `awk` program), concatenates these host definitions with the base configuration file and syntax-checks the resulting configuration file using the server software's configuration parser (`dhcpd -t`). If the file is valid, it is installed into the appropriate path and the DHCP server software is restarted. If it is not valid, the server continues running with the old configuration and an error message is displayed so that the administrator who pushed the change knows they should fix it.

3.7.2 DNS repository

The in-house software solution we use for DNS zone file generation, mentioned above, is called simply `make-dns`, and is a `perl` program written over the span of many years. It supports other kinds of resource records as well, but most relevant for this thesis are the A and AAAA record types for host forward declarations, as well as the matching reverse (PTR) records. Without delving further into this program, its output is a configuration file for ISC BIND (`named.conf`) as well as zone files for all forward and reverse zones it is configured to be responsible for, which are referenced in the BIND configuration file.

The change deployment process is nearly identical to the one described above for DHCP: the `git` server's `post-receive` hook mirrors the change onto the production DNS server machine, which then executes its own hook to generate the configuration and zone files with `make-dns`. These files are then syntax checked with a tool shipped with the name server software, `named-checkconf -z`, and installed if there are no errors.

3.7.3 Firewall configuration repository

The firewall configuration is stored in `git` just like the other types of configuration, but does not contain a "build step"; the final `pf` configuration files are stored as-is in the repository and edited by hand by administrators.

Changes are deployed in a similar manner to the other configuration types, that is, the version control server pushes the new commits it receives to both of the production firewall machines using a `post-receive` hook. The production machines then run syntax checks on the configuration using `pfctl -n`, and load the configuration if the checks pass. We can do this because `pf` loads ruleset and configuration changes atomically without affecting its operation or existing states [11, slide 11].

3.7.4 Configuration change workflow

Our system administrator team has a strong Unix background, and thus our existing workflows for making configuration changes most often involve making the changes with a text editor (or suitable Unix utilities for text processing for bulk changes) to clones of the `git` repositories described above, and then pushing those changes to the version control server, which executes the hooks necessary to bring the changes into production. As we are quite comfortable with this process and because it lends itself well to bulk changes and history analysis through the version control software, we would like to preserve a similar workflow. However, with the environment as described, provisioning a new node takes multiple steps: the administrator must make changes in several separate configuration repositories and make sure they match the others. Decommissioning a node is similarly cumbersome, and forgetting some of the steps leads to configuration errors, so it is prudent that we find a better way to manage nodes with limited lifetimes.

Chapter 4

Implementation

In this chapter we discuss the implementation of our software solution and the methods with which it functions. We discuss each type of infrastructure configuration separately and describe how we integrated our software with the existing tooling. Some example listings in this chapter contain lines longer than would fit on the page, and are thus broken into multiple lines. Since line breaks are significant in eg. YAML, we annotate line breaks added for readability in listings with the ¬ symbol as the last character of the broken line, which is then continued on the next line.

Due to the difficulty of naming things, we call our software simply `hgen`, because it uses Hiera data files as input to generate configuration for other components. We refer to it using that name later in this thesis.

4.1 Integration with the configuration data store

As described in Chapter 3, we store configuration management data in Hiera, a hierarchical key/value data store. However, node network configuration should by nature be unique to each node: it makes no sense to assign one IP address to all nodes running a certain operating system, for example. Thus, since we do not have any notion of hierarchy for nodes, we do not actually query Hiera for the data necessary to generate our network infrastructure configuration. Instead we store the requisite data in the Hiera data store, but neither Hiera nor Puppet need to use it.

Since our configuration data store happens to be a version controlled repository of YAML files, we can apply the same type of continuous integration tooling here as we do for the different network infrastructure configuration types. When a commit changing a node's data is pushed to the version control service, we can run our software to generate new supplementary configuration files for each network infrastructure service, and if they differ from previously generated version, deploy them to production.

4.2 Example node data file

In the following sections, we will discuss how the different types of configuration are generated. We will give example outputs based on the example node data file shown in listing 4.1; it is not annotated, but we describe the function of each input item the relevant sections.

Listing 4.1: `flashman.niksula.hut.fi.yaml`, a YAML data file describing a node

```
mac: 00:19:99:e8:e8:8a
ip_addrs:
    - 130.233.41.137
slaac_prefix: 2001:708:20:e336::/64
pf_rules:
    - pass proto tcp from { 89.27.121.152 ¬
       2001:14ba:21ee:5601::/64 } to port ssh
```

4.3 IP address assignment and DHCP

For IPv4 addresses, we want to assign one address per MAC and configure it with DHCP for each node. In order to do this, we store two pieces of data for each node in Hiera: these are the link-layer address (MAC address) and the assigned IPv4 address for the node. For simplicity, each data file describes a node that has exactly one link-layer address – this is of course inaccurate for many nodes in the real world, but it strikes a good compromise in that it is true for a majority of the nodes we actually want to manage with this system, and simplifies the system considerably. Networked hosts may, of course, have multiple network interfaces and thus multiple link-layer addresses: in such cases we can create additional node files that only contain information relevant to `hgen` (that is, no Puppet classes or Hiera keys used in configuration management), or just use our existing systems instead.

In the case of IPv6, we don't actually need to generate any configuration for address assignment, since SLAAC is part of the protocol and does the job for us. It is possible to extend the system to generate DHCPv6 configuration in the same way as above for IPv4, though, but that was not a requirement in our environment.

When our software is run, it generates a supplemental configuration file for the ISC DHCP Server `dhcpd`, the DHCP server software we use. This configuration file contains a host definition mapping for each node, mapping one MAC address to an IP address. If the resulting supplemental configuration file differs from what is currently deployed, the integration system proceeds to run syntax checks on the new configuration and deploys it to production.

The DHCP server's main configuration file, `dhcpd.conf` is still owned by the previously existing system, and it has been modified to contain an `include` directive to support our new system's supplemental file. Changes to the continuous integration tooling or other supporting code of the DHCP configuration repository are not necessary, since the repository hooks generate different files (although syntax-checks are always run on the combination of both).

Using the example node data file from listing 4.1, `hgen` generates the lines shown in listing 4.2 into the supplemental `dhcpd` configuration file.

Listing 4.2: Example DHCP reservation in `dhcpd.hgen.conf`

```
host flashman.niksula.hut.fi. {
    hardware ethernet 0:19:99:e8:e8:8a;
    fixed-address 130.233.41.137;
}
```

4.4 DNS Resource Records

Each node data file is named after a fully qualified domain name, eg. *flashman.niksula.hut.fi.yaml*. Using this name, DNS records are generated by creating input lines to our existing system `make-dns`: `A` records for all IPv4 addresses of the node, `AAAA` records for all its IPv6 addresses (including the address derived from the MAC address using modified EUI-64 per SLAAC; the value of `slaac_prefix` is taken from the input data as the network prefix for this) as well as reverse `PTR` records for both address families. An example of the generated input to `make-dns` is shown in listing 4.3 - using this input, `make-dns` generates the forward `A` and `AAAA` records into the `niksula.hut.fi.` zone file, as well as reverse `PTR` records into the matching reverse zone files. As `make-dns` was already an existing system at our site before this thesis, we do not describe its operation further here.

Listing 4.3: Example `make-dns` input lines generated by `hgen`

```
fqdn flashman.niksula.hut.fi. 130.233.41.137
fqdn flashman.niksula.hut.fi. ¬
   2001:708:20:e336:219:99ff:fee8:e88a
```

4.5 Firewall rules

Firewall rules are a natural thing to want to manage with our new system, because it provides life cycle management for nodes: when a node is decommissioned, any firewall holes opened for its addresses should be closed.

Instead of making up a syntax of our own, we opted to use the excellent `pf` syntax directly. In the data store for each node, we have an array called `pf_rules`, where each entry must be a valid input line to `pfctl` in an anchor context. For example, consider the following additional configuration for the node `flashman.niksula.hut.fi` in Listing 4.4; we take the `pass` lines verbatim and place them inside an anchor that matches on packets to the node's addresses (which must be defined in the same file; we explicitly do not want to resolve names to generate the configuration or to load it). The resulting generated configuration will be as in Listing 4.5.

Listing 4.4: Firewall rules in YAML file

```
pf_rules:
    - pass proto tcp from { 89.27.121.152 ¬
       2001:14ba:21ee:5601::/64 } to port ssh
```

Listing 4.5: Generated lines in pf.hgen.conf

```
table <flashman.niksula.hut.fi.> const { ¬
   130.233.41.137 ¬
   2001:708:20:e336:219:99ff:fee8:e88a }
anchor "flashman.niksula.hut.fi." to ¬
   <flashman.niksula.hut.fi.> {
        pass proto tcp from { 89.27.121.152 ¬
           2001:14ba:21ee:5601::/64 } to port ssh
}
```

We do not place these lines directly in the main configuration file `pf.conf`, because that file is version controlled in a different repository, and it was one of our goals to interoperate with existing systems. Instead, we attach the generated file (called `pf.hgen.conf`) as a loaded anchor from our main configuration file (see Listing 4.6)

Listing 4.6: pf.conf lines to load our generated anchor

```
anchor "hgen" to <mynets>
load anchor "hgen" from "/etc/pf/pf.hgen.conf"
```

Our end result is thus an anchor that contains, for each node, a constant table and an anchor that is matched for packets destined for the addresses in the corresponding table. Using anchors in this fashion makes sure that we cannot break the entire ruleset with a configuration error, and that most errors in a single node's configuration will not affect other nodes (an exception being a misconfigured IP address). Additionally, this way, we can update the generated ruleset without having to reload the main configuration file by using the `-a` option for `pfctl` [2], further improving both the fault tolerance and the performance of applying configuration changes.

Once a node is decommissioned, the generated file will no longer refer to the table nor the anchor associated with it, causing `pf` to remove the table [1, `persist` flag].

4.6 Modifications to existing integration processes

Because our software runs syntax checks against all the separate repositories' configuration when changes are made, we now depend on the state of those repositories as well. This presents a problem in our existing environment, since the changes to separate configuration repositories are mirrored by the version control service to production nodes before being built. If there is a detectable configuration error, such as a syntax error, the production node will not deploy the change. However, the erroneous configuration will still exist in the repository on the version control server, and will cause syntax checks run by our software to fail even if there are no errors introduced by a later change to the configuration data.

To solve this, we improved our continuous integration processes for the repositories in question. Where previously the automated syntax checks were made on the production machines just before deploying, they are now executed by the version control service when a change is pushed using a `pre-receive` hook. This way, if a change causes the automated checks to fail, that change is not accepted by the version control service and thus the combined state of the repositories on the version control server is always valid.

Because the version control server runs the same operating system version as our production DNS and DHCP servers, implementing the `pre-receive` hook was quite trivial: we just install matching versions of the software on the version control server. However, we cannot check the validity of the firewall configuration on the version control server, because the firewall configuration parser does not run on that same operating system; it is integrated into OpenBSD. While ripping out just the parser feature from `pfctl` and porting that to run on other operating systems would technically be possible, it would likely require considerable effort, and such a port would have to be maintained to accommodate changes in `pf` delivered by future OpenBSD versions. Thus, to ensure compatibility with the production firewall, the firewall repository's `pre-receive` hook uses `ssh` to run the configuration checks on the actual production machine before accepting changes. In theory, this compromise introduces an undesirable dependency on the firewall machine being up to the version control service, but in practice it is not so bad because the firewall machines have failover redundancy (and in fact the client doing the push most likely contacted the version control server through the firewall because of how our network is segmented, so it is not likely that both are down).

Chapter 5

Evaluation

In Chapter 1, we defined the requirements that must be met for a solution to be acceptable. We will evaluate our system as implemented against each of those requirements below.

5.1 Error tolerance

Our system fails early when presented with a configuration that results in an error detectable by static checks run against the final generated configuration files. The extent of these checks depends on the consuming software, because we use their existing tools as described Chapter 4. If an error is detected, the whole configuration change is never applied, because the version control service rejects it.

Of course, syntactically valid configuration can be erroneous as well, for example if the administrator making the change makes a mistake, or if we encounter a bug. Such mistakes are not allowed to affect the entire network, however, so our solution makes sure that any misconfiguration of a node's firewall rules can only affect traffic destined for that node, because all the generated rules are contained in anchors that match only IP packets whose destination address is one of the node's addresses. Further, if `hgen` generates rules that affect the entire parent anchor, perhaps as the result of a bug, all nodes whose firewall rules are managed by `hgen` may have their connectivity affected, but critical network services remain online, because the parent anchor is loaded with a lower preference than the rest of the rules in the main ruleset (in `pf`, the last matching rule wins).

However, misconfigurations of MAC or IP addresses in nodes may well affect other nodes, even ones not managed by `hgen`. If an administrator sets a node's MAC or IP address in `hgen` to one that is already used in our network, both nodes' connectivity may fail. We mitigate the issue somewhat for IP addresses at run time in the DHCP server: it is configured to send an ICMP Echo Request (”ping”) to addresses before it gives a lease to clients, refusing to give the lease if someone responds to the ping. However, there will still be duplicate DNS PTR records for the address, and there is no mitigation in place against duplicate MAC addresses. In practice duplicate addresses have not been a problem in our environment, but if they ever become one, further work is required to skip nodes whose addresses conflict with others - it is trivial to check duplicates among the nodes we manage with `hgen`, but integration with the other existing systems with respect to duplicate detection presents a larger problem.

5.2 Source of truth

Our software is not the single source of truth about the desired network state, since the other extant management tools have just as much or more authority in the final configuration. However, the system we should evaluate is actually not just the software written for this thesis; rather, it is the combined system of the existing tools and the software we wrote that integrates and cooperates with them. This cumulative system is in fact authoritative for the DNS RRs, DHCP reservations and firewall rules in our environment - no inputs other than the `git` repositories described in Chapter 3 as well as `hgen` affect the final state of each program's configuration.

5.3 Versioning and rollback

Because our system's source of truth comprises only `git` repositories as described above, versioning changes, branching and merging, arbitrary rollback as well as history analysis is very simple: we can use `git` itself as well as any of the numerous extant tools that work with it.

5.4 Integration to existing systems

By design, our system does not replace the existing utilities in use for managing our network infrastructure configuration: even after deploying `hgen`, all of the previous tools still work like they used to (but with stricter error checking). This makes it easy to migrate any number of nodes under `hgen`'s management at any time – although we will certainly not do so for every node, because one size does not fit all.

5.5 Performance

As hgen only needs to be run when an administrator makes a configuration change, and as a result of a run, generates configuration to be used by other software, there are two aspects of performance we should consider: the impact on the runtime performance of the network infrastructure software caused by using configuration generated with hgen (compared to the previous situation, where configuration was either generated with the existing systems or hand-written by administrators), and the time it takes for our software to run once with respect to the number of nodes it is managing (ie. its scalability).

5.5.1 Runtime impact

Since hgen generates input for our DNS configuration generator, the output, ie. the input zone files for the ISC BIND software, are identical to what they would be had we not used hgen at all. That means there is no runtime cost for using hgen - we only incur a cost at configuration change time, when we have to regenerate the zone files.

The same principle applies to DHCP: hgen generates identical configuration directives to what we've been using in the past for ISC dhcpd, meaning there is no runtime cost there either.

The packet filter, pf, is a little different, however. Contrary to what we were doing without hgen, we now load an additional layer of anchors into the ruleset. Before, we were placing node-specific rules inside one anchor in the main ruleset, but with hgen we are actually using two layers of anchors: one in the main ruleset, and one per each node inside the first anchor. The per-node anchors have associated filtering rules, so we do incur a non-zero cost at ruleset evaluation - pf has to first evaluate the anchor's filter rule (to see if the traffic is destined for the node), and if so, then evaluate all rules inside that anchor.

However, ruleset evaluation only happens for packets that create a new state in the state table (eg. TCP SYN segments). In figure 5.1 we present on a logarithmic scale the rate of state searches versus the rate of state inserts in the state table. Every packet processed by the firewall will first cause a state search, so the number of state searches is equal to the number of packets processed by the firewall. If there is no state matching the packet, the ruleset needs to be evaluated to know whether or not allow the packet, and if it is allowed, it will create a state, ie. cause a state table insertion. Blocked packets and packets filtered with stateless rules (rules specifying no state) do not cause state table insertions, but we do not particularly care

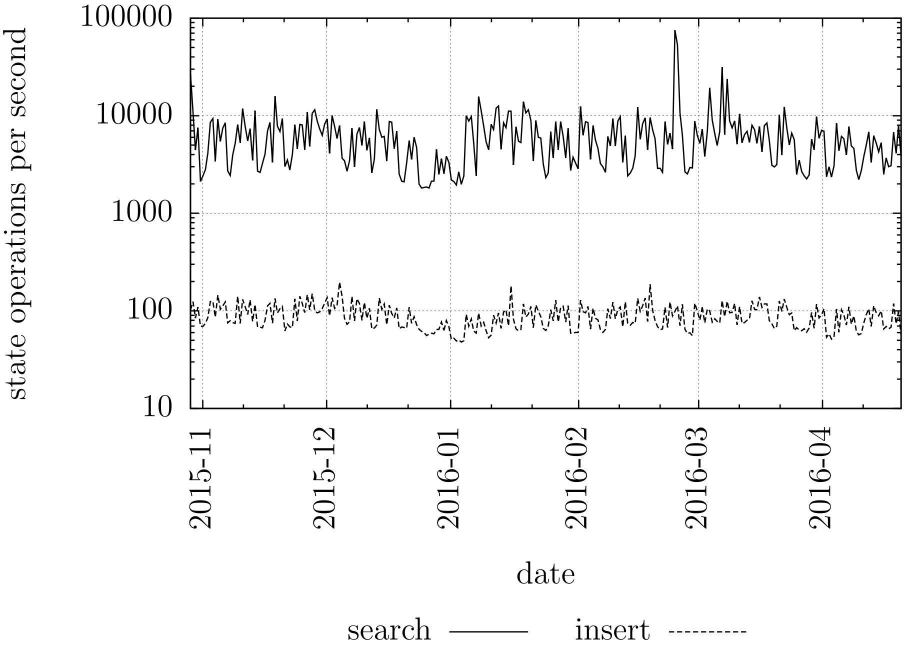

Figure 5.1: `pf` state table searches and insertions per second over a six-month period, log_{10} scale

if blocked packets get delayed. Statelessly passed packets do incur a penalty we might care about, but in practice we use them very rarely, and even then usually with the `quick` option, stopping ruleset evaluation before it hits the anchor generated by `hgen`. The graph tells us that only a very small minority of packets processed by our firewalls during a six-month period caused state insertions; the rest were either blocked, passed statelessly or found in the state table and thus short-circuited without needing to evaluate the ruleset; `pf`'s state table is a red-black tree, so lookup complexity is just $O(logn)$ for n existing states [1, 13], and using `hgen` does not affect the number of states created.

Given this, we can assert that the runtime cost of using `hgen` is marginal in our environment.

5.5.2 Scalability to a large number of nodes

While runtime cost is negligible, we should also consider the time it takes to process a configuration change made by an administrator. To that end, we measured our software's consumed CPU time in both user and kernel mode for various generated sets of node data files. The results are presented in figure 5.2, where the number of nodes in the dataset is on the X axis and the kernel and user CPU times are on the Y axis.

From the graph we can tell that the increase in node data files causes a linear increase in the amount of consumed CPU time, ie. the complexity of running our program is $O(n)$ where n is the number of node data files.

The wall-clock runtime of our program is very close to the sum of the user and kernel times. Since our program is single-threaded, this result means that the program is CPU-bound. If we wanted to improve its runtime, we could either attempt to parallelize it or optimize the hot paths. A quick analysis using `cProfile`, a profiler shipped with Python, reveals that 82 percent of the runtime of the program with 65536 input files was spent in `PyYAML`, the third party YAML parsing library we chose. However, our copy of `PyYAML` is using a pure-Python parser; it is also possible to build a copy with bindings to LibYAML, a C implementation of a YAML parser and emitter. We did not measure the impact of doing so, because the runtime of the program against our current production dataset is low enough to not be a problem, but if it ever becomes one, building a copy of PyYAML with LibYAML bindings is a low-hanging fruit to tackle.

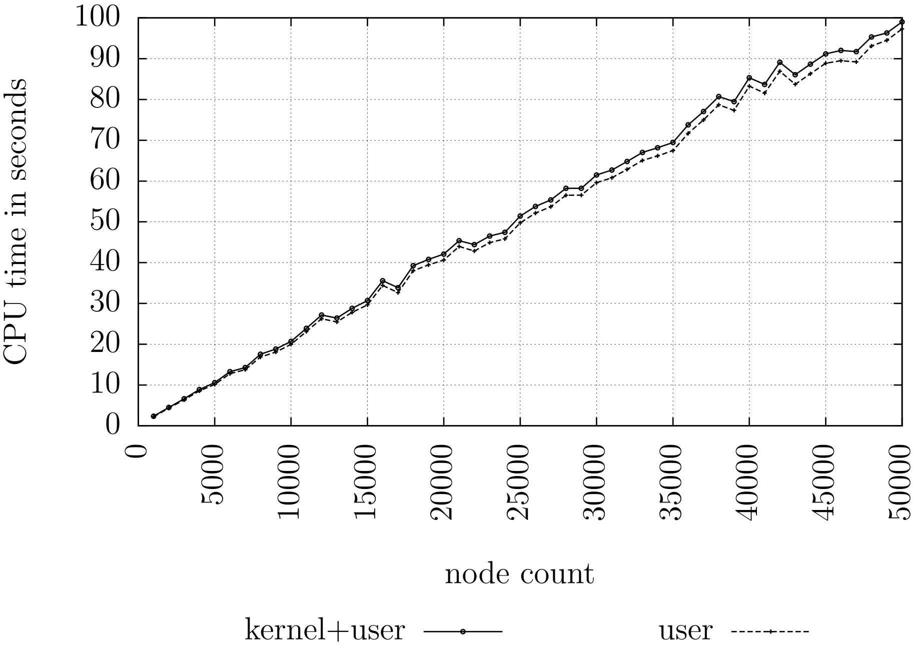

Figure 5.2: CPU time in relation to number of node data files

Chapter 6

Conclusions

In this thesis we presented a software system for managing network infrastructure configuration on a per-node basis, without replacing existing tools or infrastructure. This system greatly reduces the effort system administrators have to expend in several situations involving network nodes with limited lifetimes in our environment, in particular commissioning and decommissioning nodes: Previously these actions required several separate changes to different configuration repositories, as discussed in chapter 3. The workflow is similar and still familiar to our system administrators: we make changes using a text editor or Unix text processing tools, commit those changes to `git` and push them to the version control server in order to deploy them to production, but now we only need to do so once: all the information that comprises a node is stored in just one repository, for most nodes. Importantly, this prevents configuration errors caused by residual configuration concerning nodes that have been decommissioned.

In the previous chapter, we evaluated our solution against the criteria we set in chapter 1, and it passed all the requirements. It is reasonably error tolerant in that it confines many types of configuration errors in a node's configuration specifically to that node without affecting the others, it forms the authoritative source of truth for network infrastructure configuration when combined with our previously existing tools, it integrates well with those tools and it supports versioning, rollback and history analysis extremely well thanks to `git`.

We believe our implementation presents a suitable solution for the problem stated in chapter 1, and it does so with little to no runtime performance impact, as discussed in chapter 5. It fits our workflows and requirements better than existing solutions we evaluated, because as discussed in chapter 2, those solutions fail some of the requirements we set, unlike our implementation.

We have deployed the software and requisite version control hooks onto our production servers, and the system is working well.

6.1 Future work

While we consider our system suitable for production, there are some shortcomings with it. They are not critical, however, so we are not fixing them for this thesis but instead consider them avenues for further work.

A nice addition to our system would be to add a built-in method for IP address management; currently IPv4 addresses have to be reserved using other systems, and the node data files need to match. Optimally, node provisioning would not require that additional step either.

Configuration generation performance is acceptable, but it would likely not be terribly difficult to improve either. If the number of nodes we manage grows by an order of magnitude or more, this is definitely something we should do.

We currently store expiration dates for nodes in our data files, and `hgen` does warn about nodes that have expired when an administrator makes a configuration change, but periodically checking and reporting automatically would be a further improvement to our node life cycle processes.

Finally, while not directly related to the system presented in this thesis, we could move more nodes' configurations under `hgen` management to make managing their life cycle easier in the future. Because our system coexists with existing tools, this can be done whenever and in as large or small batches as is desired.

Bibliography

[1] *pf.conf(5): pf.conf – packet filter configuration file*, February 2015. URL `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-5.8/man5/pf.conf.5`. OpenBSD 5.8.

[2] *pfctl(8): pfctl — control the packet filter (PF) device*, June 2015. URL `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-5.8/man8/pfctl.8`. OpenBSD 5.8.

[3] Randy Bias. Architectures for open and scalable clouds, 2012. URL `http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds`.

[4] Mark Burgess and Oslo College. Cfengine: a site configuration engine. In *in USENIX Computing systems, Vol*, 1995.

[5] B. Carpenter and S. Jiang. RFC 6866: Problem Statement for Renumbering IPv6 Hosts with Static Addresses in Enterprise Networks, Feb 2013. Status: Informational.

[6] Internet Systems Consortium. *BIND 9 Administrator Reference Manual*, 2015. URL `http://ftp.isc.org/isc/bind9/cur/9.9/doc/arm/Bv9ARM.pdf`. BIND Version 9.9.8-P4.

[7] Internet Systems Consortium. *dhcpctl(3): dhcpctl library initialization*, 2016. URL `https://www.isc.org/downloads/`. `man` page shipped with ISC DHCP version 4.3.4.

[8] Internet Systems Consortium. *omapi(3): OMAPI - Object Management Application Programming Interface*, 2016. URL `https://www.isc.org/downloads/`. `man` page shipped with ISC DHCP version 4.3.4.

[9] R. Droms. RFC 2131: Dynamic Host Configuration Protocol, Mar 1997. Status: Draft Standard.

[10] Brian D Freeman. Network configuration management. In *Guide to Reliable Internet Services and Applications*, pages 255–275. Springer, 2010.

[11] David Gwynne. firewalling with OpenBSD's pf and pfsync, 2011. URL `http://www.openbsd.org/papers/lca2011-dlg.pdf`.

[12] ISO/IEC 7498-1:1994(E). Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, November 1994. URL `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip`.

[13] Ryan McBride. Introduction to pf, 2004. URL `http://www.openbsd.org/papers/bsdcan04-pf/index.html`.

[14] P. Mockapetris. RFC 1034: Domain Concepts and Facilities, Nov 1987. Status: Internet Standard.

[15] Jürgen Schönwälder, Martin Björklund, and Phil Shafer. Network configuration management using netconf and yang. *IEEE Communications Magazine*, 48(9):166–173, 2010.

[16] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE), Apr 1997. Status: Proposed Standard.